CS100 Recitation 12

modified from GKxx

Contents

- 继承、多态
  - 知识点梳理
  - 一个例子

继承 (Inheritance)、多态 (Polymorphism)

继承

- 一个子类对象里一定有一个完整的父类对象*
- 禁止“坑爹”：继承不能破坏父类的封装性

继承

- 一个子类对象里一定有一个完整的父类对象*
  - 父类的**所有成员**（除了构造函数和析构函数）都被继承下来，无论能否访问
  - 子类的构造函数必然先调用父类的构造函数来初始化父类的部分，然后再初始化自己的成员
  - 子类的析构函数在析构了自己的成员之后，必然调用父类的析构函数

* 除非父类是空的，这时编译器会做“空基类优化 (Empty Base Optimization, EBO)”。

空基类优化 (Empty Base Optimization, EBO)

- 一般来说，任何对象或者成员对象的大小至少占一字节，即使是空对象（没有非静态数据成员的类或结构体）
- 但如果空类作为基类被其他类继承，且子类不是空对象，则空类占用的内存会被优化掉
- 应用：使类无法拷贝且节省空间（before C++11）(example1)

继承

- 禁止“坑爹”：继承不能破坏父类的封装性
  - 子类不可能改变继承自父类成员的访问限制级别
  - 子类不能随意初始化父类成员，必须经过父类的构造函数
    - 先默认初始化后赋值是可以的

子类的构造函数

必然先调用父类的构造函数来初始化父类的部分，然后再初始化自己的成员

```
class DiscountedItem : public Item {
 public:
  DiscountedItem(const std::string &name, double price,
                 int minQ, double discount)
      : Item(name, price), m_minQuantity(minQ), m_discount(discount) {}
};
```

可以在初始值列表里调用父类的构造函数

- 如果没有调用?

子类的构造函数

必然先调用父类的构造函数来初始化父类的部分，然后再初始化自己的成员,可以在初始值列表里调用父类的构造函数

- 如果没有调用，则自动调用父类的默认构造函数
    - 如果父类不存在默认构造函数，则报错。

```
//example 2
class Base{
public:
    Base() : val_Base(0) {cout<<"**Base default construction**"<<endl>;}
    int val_Base;
};
class A : public Base{
public:
    A() : val_A(0) {cout<<"**A default construction**"<<endl;}
    int val_A;
};
```

子类的析构函数

析构函数在执行完函数体之后：

- 先按成员的声明顺序倒序销毁所有成员。对于含有 non-trivial destructor 的成员，调用其 destructor。
- 然后调用父类的析构函数销毁父类的部分。

子类的拷贝控制

自定义：不要忘记拷贝/移动父类的部分

```
class Derived : public Base {
 public:
  Derived(const Derived &other) : Base(other), /* ... */ { /* ... */ }
  Derived &operator=(const Derived &other) {
    Base::operator=(other);// 当前在子类的作用域中，要指定为父类作用域下的operator=
    // ...
    return *this;
  }
};
```

合成的拷贝控制成员（不算析构）的行为？

- 拷贝控制成员：构造函数、析构函数、拷贝构造函数、拷贝赋值运算符、移动构造函数、移动赋值运算符

子类的拷贝控制

合成的拷贝控制成员（不算析构）：先父类，后子类自己的成员。

- 如果这个过程中调用了任何不存在/不可访问的函数，则合成为 implicitly deleted

动态绑定

- 静态对象类型和动态对象类型
  - 静态对象类型：对象在声明时的类型，编译时决定。

```
int a=1;    double b=1.0;   char c='a'; //.......
```

  - 动态对象类型：当前所指对象类型，运行时决定。

```
int *p=new int;  //.......
```

  - 静态绑定：（指针或引用）绑定的是静态对象类型，发生在编译时。
  - 动态绑定：（指针或引用）绑定的是动态对象类型，发生在运行时。

动态绑定

- 动态绑定需要的三个条件：
  - 有指针或者引用
  - 指针或引用向上转型
  - 虚函数

向上转型

- 一个 `Base *` 可以指向一个 `Derived` 类型的对象
  - `Derived *` 可以向 `Base *` 类型转换
- 一个 `Base &` 可以绑定到一个 `Derived` 类型的对象
- 一个 `std::shared/unique_ptr<Base>` 可以指向一个 `Derived` 类型的对象
  - `std::shared/unique_ptr<Derived>` 可以向 `std::shared/unique_ptr<Base>` 类型转换
- 例如:

```
class Base;
class Derived : public Base; //必须是public继承
Base *p = new Derived;
```

简单点说就是用父类的指针或引用指向子类的对象，将子类转换为父类，此时属于子类的成员变量和成员函数（非虚函数）就不能访问了。

虚函数

继承父类的某个函数 `foo` 时，我们可能希望在子类提供一个新版本（override）。

我们希望在一个 `Base *`，`Base &` 或 `std::shared/unique_ptr<Base>` 上调用 `foo` 时，可以根据**动态类型**来选择正确的版本，而不是根据静态类型调用 `Base::foo`。

虚函数

在子类里 override 一个虚函数时，函数名、参数列表、`const`ness 必须和父类的那个函数**完全相同**。

- 返回值类型必须**完全相同**或者随类型*协变* (covariant)。（父类中返回父类指针，子类返回子类指针）

```
class Base{
    virtual Base* foo() {cout<<"Base"};//返回父类指针
};
class Derived : public Base{
    virtual Derived* foo() {cout<<"Derived"};//返回子类指针
};
```

虚函数

加上 `override` 关键字：让编译器帮你检查它是否真的构成了 override
例如：

```
class Base{
    virtual void foo() {cout<<"Base";}
};
class Derived : public Base{
    virtual void foo(int x) override {cout<<"Derived";}//error,没有重写父类的foo函数
};
```

```
class Base{
    virtual void foo() {cout<<"Base";}
};
class Derived : public Base{
    virtual void foo(int x) {cout<<"Derived";}//不会报错
};
```

- 子类的虚函数前的 `virtual` 可省略，但是父类的 `virtual` 不能省略

虚函数

除了 override，不要以其它任何方式定义和父类中某个成员同名的成员。

- 阅读以下章节，你会看到违反这条规则带来的后果。

《Effective C++》条款 33：避免遮掩继承而来的名称

《Effective C++》条款 36：绝不重新定义继承而来的 non-`virtual` 函数

《Effective C++》条款 37：绝不重新定义继承而来的缺省参数值

纯虚函数

通过将一个函数声明为 `=0`，它就是一个**纯虚函数** (pure virtual function)。

```
class Base{
    virtual void foo() = 0;
};
```

一个类如果有某个成员函数是纯虚函数，它就是一个**抽象类**。

- 不能定义抽象类的对象，不能调用无定义的纯虚函数*。

```
Base b; //error
Base *p = new Base; //error
class A:public Base{
    void func(){
        Base::foo();//error
    }
};
```

* 事实上一个纯虚函数仍然可以拥有一份定义，阅读《Effective C++》条款 34（必读，HW7 的客观题会涉及此条款的内容）。

纯虚函数

纯虚函数通常用来定义**接口**：这个函数在所有子类里都应该有一份自己的实现。

如果一个子类继承了某个纯虚函数而没有 override 它，这个成员函数就仍然是纯虚的，这个类仍然是抽象类，无法被实例化。

```
class Base{
    virtual void foo() = 0;//纯虚函数，Base是抽象类，不能实例化
};
class A:public Base{};//没有重写foo函数，A任是抽象类，不能实例化
class B:public Base{
    void foo() override {} //B不是抽象类，可以实例化
};
```

动态绑定

- 动态绑定就是指在运行时根据对象的动态类型来选择正确的成员函数版本。

```
//example3
class Base{
public:
    virtual void foo() {cout<<"Base"<<endl;}
};
class Derived : public Base{
public:
    virtual void foo() {cout<<"Derived"<<endl;}
};
int main(){
    Base *p = new Base;
    p->foo();//Base
    delete p;
    p = new Derived;
    p->foo();//Derived
    delete p;
}
```

运行时类型识别 (RTTI)

`dynamic_cast` 可以做到“向下转型”（即将 `Base*` 转为 `Derived*`）。

- 它会在运行时检测这个转型是否能成功
- 如果不能成功，`dynamic_cast<T*>` 返回空指针，`dynamic_cast<T&>` 抛出 `std::bad_cast` 异常。
- **非常非常慢**，你几乎总是应该先考虑用一组虚函数来完成你想要做的事。

`typeid(x)` 可以获取表达式 `x`（忽略顶层 `const` 和引用后）的动态类型信息

- 通常用 `if (typeid(*ptr) == typeid(A))` 来判断这个动态类型是否是 `A`。

https://quick-bench.com/q/E0LS3gJgAHlQK0Em_6XzkRzEjnE

根据经验，如果你需要获取某个对象的动态类型，通常意味着设计上的缺陷，你应当修改设计而不是硬着头皮做 RTTI。（oop有三大特征:封装,继承,**多态**）

设计

`public` 继承建模出“is-a”关系：A discounted item **is an** item.

但有些时候英语上的“is-a”具有欺骗性：

- Birds can fly. A penguin is a bird.
- A square is a rectangle. 但矩形的长宽可以随意更改，而正方形不可以。

阅读《Effective C++》条款 32（必读，HW7 的客观题会涉及此条款的内容）。

- `public`继承意味着“is-a"。适用于`base class`的每一件事情也一定是用于derived class身上，因为每一个`derived class`对象也都是一个`base class`对象。

继承的访问权限

这个 `public` 是什么意思?

```
class DiscountedItem : public Item {};
```

继承的访问权限

```
class DiscountedItem : public Item {};
```

继承的访问权限：**这个继承关系（“父子关系”）是否对外公开。**

如果采用 `private` 继承，则在外人眼里他们不是父子，任何依赖于这一父子关系的行为都将失败（有哪些？）。

继承的访问权限

```
class DiscountedItem : public Item {};
```

继承的访问权限：**这个继承关系（“父子关系”）是否对外公开。**

如果采用 `private` 继承，则在外人眼里他们不是父子，任何依赖于这一父子关系的行为都将失败。

- 访问继承而来的成员（本质上也是向上转型）
- 向上转型（包括动态绑定等等）、向下转型

`private` 继承：建模“is-implemented-in-terms-of”。阅读《Effective C++》条款 38、39。

继承的访问权限

```
struct A : B {}; // public inheritance
class C : B {};  // private inheritance
```

`struct` 和 `class` 仅有两个区别：

- 默认的成员访问权限是 `public` / `private` 。
- 默认的继承访问权限是 `public` / `private` 。

一个 OOP 的例子

抽象语法树的各种结点