CS100 Recitation 11

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
- 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

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

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

```
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，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。

合成的默认构造函数的行为？

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，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。

合成的默认构造函数：先调用父类的默认构造函数，再 ……

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);
    // ...
    return *this;
  }
};
```

合成的拷贝控制成员（不算析构）的行为？

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

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

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`。

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)。

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

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)。

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*。

* 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**）

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。

运行时类型识别 (RTTI)

`dynamic_cast` 可以做到"向下转型"：

- 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。

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**）。

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（选读）。

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` 。