第15章 面向对象程序设计

OOP：概述

- 面向对象程序设计(*object-oriented programming*)的核心思想：
  - 数据抽象：可以将类的接口与实现分离
  - 继承：可以定义相似的类型并对其关系建模
  - 动态绑定：可以在一定程序上忽略相似类型的区别，而以统一的方式使用它们的对象

继承

通过继承联系在一起的类构成了一种层次关系

- 基类：层次关系的根部
  - 负责定义在层次关系中所有类共同拥有的成员
- 派生类：直接或间接从基类继承而来
  - 每个派生类定义各自特有的成员

动态绑定

支持同一段代码分别支持处理基类和派生类

> **后续介绍原因**

- **当我们使用基类的引用（或指针）调用一个虚函数时，将发生动态绑定**

  > 在运行时选择函数的版本，所以又被称为运行时绑定

定义基类和派生类

定义基类

基类的两种成员函数

- 基类希望派生类直接继承而不要改变的函数
- 基类希望派生类进行覆盖的函数——**虚函数**

```
virtual double net_price(...)..     // 用声明时，用关键字 virtual 来表名该函数是一个虚函数
```

  - 当我们使用指针或引用调用虚函数时，该调用将动态绑定，即根据对象类型不同调用不同的版本

    > 动态绑定的实现

  - 任何构造函数之外的非静态函数都可以是虚函数
  - `virtual`关键字只能出现在类内的声明语句之前而不能用于类外部的函数定义
  - 如果基类把一个函数声明为虚函数，则该函数在派生类中隐式地也是虚函数

    > 影响 多层继承关系

- 基类通常都应该定义一个虚析构函数

  > 因为每个类都是有默认的虚构函数，所以基类中的虚析构函数一定会被重定义

- 被用作基类的类，必须已经定义而非仅声明
  - 一个类不能派生本身

定义派生类

派生类必须通过 类派生列表 来明确指出它是从哪一个基类继承而来的

```
class Bulk_quote : publie Quote { }          // Bulk_quote 以public的方式继承了Quote
```

- 派生类的声明不用包含它的派生列表

  > 这里可以理解一下声明的意义，只是让程序知道有这个对象即可

- 派生类经常（但不总是）**重定义**它继承的虚函数；

  如果没有重新定义，派生类则会直接继承其在基类的版本。
- 对于一个派生类，自己的对象和从基类中继承的对象在内存中不一定是连续分布的
- 派生类不能直接初始化基类的成员，需要调用基类的构造函数来初始化它的基类部分

  > 每个类控制它自己的成员初始化过程。

> 要与类对象交互必须使用该类的接口；尽管我们可以在派生类构造函数内给它的公有或保护的基类成员赋值，但是也不建议这样做

- 除非我们特别在基类的初始化函数指出，否则基类部分会执行默认初始化
- 首先向初始化基类部分，再初始化派生类的成员

> **这个顺序是必须的吗**

防止继承

```
class A final {     // 类后跟着final关键字，表示该类不能被继承

}
```

类型转换

因为派生类对象中含有基类对应的组成部分，所以我们可以把派生类对象当作基类对象来使用，也能将基类的指针或引用绑定到派生类对象中的基类部分（动态绑定）

> 仅支持指针和引用之间的转换，对象之间不能转换

- 属于隐式类型转换，编译器会自动进行

  不存在基类想派生类的隐式类型转换

```
Quote *p = &bulk_quote;
Bulk_Quote *p2 = p;              // 即使一个基类的指针或引用绑定在一个派生类对象上，也不能执行基类向派生类的转换
```

> 编译器通过检查指针或引用的静态类型来推断改转换是否合法 **??**

- 智能指针也支持派生类向基类的转换
- 因为类的构造函数都是引用自己，所有可以用派生类来初始基类

静态类型与动态类型

表达式的静态类型在编译是总是已知的，它是变量声明时的类型或表达式生成的类型；

动态类型则是变量或表达式表示的内存中的对象类型，直到运行时才知道

```
void print(const Quote& item){      // item的静态类型是Quote&，但是当传递一个Bulk_Quote后，它指向的内存区域的对象类型是Bulk_Quote类型
    ...
}
```

- 对于指针和引用，其静态类型和动态类型可能不一致；

  其他对象两种类型一定一致

虚函数

派生类中的虚函数

- 如果我们在派生类中覆盖了某个虚函数，可以再一次使用 `virtual` 指出该函数的性质

  > - 非必须。基类的虚函数在所有派生类中都是虚函数

- 覆盖基类的虚函数，形参类型、返回类型必须与对应虚函数相同

  > - 例外：当类的虚函数返回的是类本身的指针或引用时，重定义返回类型只要可以访问就行
  >
  > ```
  > 虚函数 返回D*;
  > 重定义的虚函数 返回B*;
  > B是D的派生类，从D到B的类型转换是可以访问的
  > ```

---

override和final说明符

- 如果在派生类中，定义一个与虚函数同名但形参不同的成员函数，是合法行为；这个函数是一个独立的函数

  C++11新标准，允许派生在显式地注明它使用某个成员函数覆盖它继承的某个虚函数

  - 在形参列表后，或在const成员函数的const关键字后，或在引用成员函数的引用限定符后，添加一个关键字 `override` ；
  - 编译器会自动帮我们检查，该函数是否真的覆盖了一个虚函数；
- 同样，我们可以在虚函数后注明 `final` 来声明该虚函数不允许重定义；

  任何尝试覆盖该函数的操作都将引发错误

  ```
  virtual void f() const fianl { }
  ```

虚函数与默认实参

虚函数的默认实参由本次的静态类型决定

> 即，默认实参一定是由基类决定的

回避虚函数的机制

某些时刻，希望强迫虚函数的调用不要动态执行，而是使用特定的版本

- 使用作用域运算符

```
1 double undiscounted = base->Quote::net_price(43);        // 不用管base的类型，强制调用Qutote的版本
```

- 通常情况，只有成员函数（或友元）中的代码需要使用作用域运算符来回避虚函数的机制
- 如果想在派生类的重定义函数中调用对应的虚函数，那么就必须使用作用域运算符

> 否则则认为是调用自身的递归行为

抽象基类

纯虚函数

在声明语句的分号之前，书写 `=0` 就可以将一个虚函数说明为纯虚函数

```
1 virtual double net_price(xx, xx) const = 0;
```

> =0没有什么意义，只是告诉编译器而已

- =0只能出现在类内部的虚函数声明语句处
- 纯虚函数无须定义

> 无须定义不表示不能定义。
>
> 如果为纯虚函数提供定义，则函数体必须定义在类的外部。
>
> **感觉定义纯虚函数没有意义呀**

- 如果派生类需要实例化就得重写纯虚函数，否则派生类就是一个抽象类，不能被实例化

抽象基类

含有（或未经覆盖直接继承）纯虚函数的类是抽象基类

- 抽象基类负责定义接口，而后续的派生类可以覆盖该接口
- 抽象基类无法实例化

> 因为无法为纯虚函数分配空间

访问控制与继承

受保护的成员：`protected`

- 对于类的用户是不可访问的；
- 对于派生类的成员和友元来说是可以访问的
- 派生类中，只能通过派生类的对象来访问基类中受保护的成员；

  派生类对基类受包含的成员没有任何访问特权

```
class Base {
    protected:
        int mem;
}
class base:Base {
    void get (base b) {
        b.mem;      // 正确
    }
    void get (Base b) {
        b.mem;      // 错误
    }
}
```

派生访问说明符

所谓的派生访问说明符，并不会影响派生类D的**成员**对基类B的成员的访问权限，因为这个权限已经由B中成员的访问说明符**全权规定**

派生访问说明符其实定义的是，在派生类将基类的成员继承过来之后，这些成员在D中的**新的访问权限**

- 如果以public来继承B，则B中成员的访问权限在D中具有相同的访问权限；
- 如果以protected来继承B，则D中成员的访问权限的变化是public成员改变成protected成员，其余不变；

- 如果以private来继承B，则D中的所有成员都变成了private成员
- 大多数类都只继承一个类，称为"单继承"

默认的继承级别

- 使用class关键字定义的派生类是私有继承
- 使用struct关键字定义的派生类是公有继承

派生类向基类转换的限制

- 如果**D公有地继承B**，**用户代码**才能使用派生类到基类的转换

```
class Base {
protected:
    int mem;
};
class base: protected Base {
    int t;
};

int main(){
    base t;
    Base *p = &t;   // 报错
}
```

- 无论D以什么方式继承B，**D的成员函数和友元**都能使用派生类向基类的转换
- 如果D继承B的方式是公有的或者受保护的，则**D的派生类成员和友元**可以使用D向B的类型转换；

  反之，则不能使用

友元与继承

- 不能继承友元关系
- 每个类负责控制各自的成员的访问权限

```
calss Base {
    friend class Pal
}
class Pal [
    int f(Base b) { return b.mem; }
    inf f2(base b) { return b.t; }  // 错误，Pal不是base的友元，也就不能访问base内的成员
    int f3(base b) { return b.mem; } // 正确，base中mem的权限仍然由Base控制
]
```

改变个别成员的可访问性

通过在类中是使用 `using` 声明语句，我们可以将类的直接或间接基类中的任何可访问成员标记出来；

using声明语句中的名字的权限由该using声明语句之前的访问说明符决定

继承中的作用域

- **派生类的作用域嵌套在基类作用域之内。**

  对于派生类来说，使用基类成员和派生类成员没有区别、

  > - 如果一个名字在派生类的作用域中无法被解析，则编译器继续在外层的基类作用域中寻找该名字的定义
  > - 定义派生类中的函数不会重载基类中的成员：只会覆盖
  >   - 可以同时**声明作用域**来主动调用被派生类覆盖的基类函数

- 对于基类中静态成员，在整个继承体系中只存在该成员的唯一定义

类成员的调用解析过程

```
p->mem()
```

1. 确定p的静态类型
2. 在p的静态类型对应的类中查找mem。如果查找不到，则一次在直接基类中不断查找到达继承的顶端。

   如果还是找不到，则编译器报错

3. 一旦找到mem，就进行常规的类型检查以确认对应当前找到的mem，本次调用合法
4. 假设调用合法，则编译器将根据调用的是否是虚函数而产生不同的代码：
    - 如果mem是虚函数且我们通过引用或指针进行调用，则编译器产生的代码将在运行时，根据对象的动态类型确定到底运行该虚函数的哪个版本
    - 如果mem不是虚函数或者我们是通过对象（而非引用或指针）进行的调用，则编译器将产生一个常规函数的调用

在编译时进行名字查找

一个对象、引用或指针的静态类型决定了该对象的哪些成员是可见的；

即使静态类型和动态类型不一致，但我们能使用该对象的哪些成员仍是由静态类型决定的·

```
Bulk_Quote bulk;
Bulk_Quote * bulk_p = &Bulk;
Quote * item_p = &Bulk;
bulk_p -> discount();        // 正确，discount()在Bulk_Quote中定义
item_p -> discount();        // 错误，尽管item_p的动态类型是Bulk_Quote，当时其静态类型决定它不包含discount对象
```

覆盖重载的函数

如果基类中的虚函数存在重载的形式，我们在派生类中可能只想覆盖其中一种形式，但是按上面的描述，只要我们重定义了一个同名的函数，

就会覆盖所有的函数

- 解决方法：

  在派生类中使用 `using [函数名]`，将基类该函数的所有重载实例都添加到派生类的作用域中，这样在派生类中定义其特有的函数就可以了

构造函数与拷贝控制

虚析构函数

```
class base {
    public:
        virtual ~Quote() = default;
}
```

为什们基类都要定义虚析构函数?

因为当我们使用delete删除指针时，可能出现指针的静态类型和删除对象的动态类型不符的情况;

通过在基类中将析构函数定义为虚函数以确保执行正确的析构函数版本

> 如果基类的析构函数不是虚函数，则delete一个指向派生类对象的基类指针将产生未定义的行为

- 即使基类定义了析构函数，它不一定需要拷贝和赋值操作

  > 即不符号三/五法则

- 虚析构函数将阻止合成移动函数（包括基类和派生类）

> 当我们不定义虚构函数的时候，编译器会默认生成一个什么都不做的析构函数，但是注意了默认生成的析构函数就是普通函数不是虚函数

合成拷贝控制与继承

类中合成的函数负责使用直接基类中对应的操作对一个对象的直接基类部分进行初始化

> 派生类的构造函数，和调用直接基类的构造函数，.....;
>
> 无论基类成员是合成的版本还是自定义的版本都可以

- 要求相应的成员可访问，并且不是一个被删除的函数
- 某些定义基类的方式可能导致派生类成员成为被删除的函数:
  - 如果基类中默认构造函数、拷贝构造函数、拷贝赋值运算符或析构函数是被删除的或不可访问的，则派生类中对应的合成成员函数将是被删除的

    > 因为编译器无法使用基类成员来执行派生类对象基类部分的相关操作

  - 如果基类中有一个不可访问或删除的析构函数，则派生类中合成的默认和拷贝函数将是删除的

    > 编译器无法销毁派生类对象的基类部分

> **析构函数不能是删除成员？？P450**

- 编译器将不会合成一个删除掉的移动操作

> **??**

> 个人总结：如果基类中实现了这些成员函数（无论是合成的还是自定义的），在派生类中都会合成相应的成员函数

派生类的拷贝控制成员

定义派生类的拷贝或移动构造函数

- 默认情况下，基类的默认构造函数初始化派生类对象的基类部分；

  如果想拷贝（或移动）派生类的基类部分，则必须在派生类的构造函数（多种类型）初始值列表中**显式地**使用基类的拷贝（或移动）函数

```
1 D(const D &d) { }        // 默认使用基类的默认构造函数初始化派生类的基类部分
2 D(const D &d): Base(d) {}    // 显式声明使用基类的拷贝构造函数
```

派生类赋值运算符

派生类的赋值运算符必须显式地为其基类部分赋值

```
1 D &D::operator=(const D &rhs) {
2     Base::operatro=(rhs);        // 对应下面问题，是否合成的类型 是用 默认构造函数实现？
3     //处理派生类部分
4     return *this;
5 }
```

> 不显式声明使基类赋予拷贝函数，派生类也会生成合成的赋值运算符

派生类析构函数

派生类的析构函数只负责销毁由派生类自己分配的资源；

派生类对象的基类部分和调用基类的析构函数销毁

- 顺序：派生类析构函数->基类

在构造函数或析构函数中调用虚函数

编译器把对象的类型和构造函数（析构函数）的类看作同一个；

- 如果构造函数或析构函数调用了某个虚函数，则我们应该执行与构造函数或析构函数所属类型向对象的虚函数版本
- 这种绑定对间接调用虚函数也有用

> 派生类中调用基类构造函数，基类构造函数再调用虚函数

继承的构造函数

一个类只能继承其直接基类的构造函数

```
class SubClass : public MainClass {
    public:
        using MainClass:MainClass;          // 继承基类的构造函数
    ...
}
```

- 作用于构造函数时，using声明语句将令编译器产生代码；

  对于基类中的每个构造函数，编译器都在派生类中生成一个形参列表完全相同的构造函数

  - ```
    SubClass(parm):MainClass(args);          // args是将派生类构造函数的形参传递该基类的构造函数
    ```

  - 派生类中自己的数据成员将被默认初始化
- 与using作用于基类成员不同，构造函数的using声明不会改变构造函数的访问级别；
- 一个using声明语句不能指定为explicit或constexpr的。

> 派生类中继承的构造函数会基类构造函数的所有属性(explicit 或 constexpr)

- 当基类构造函数含义默认实参，这些实参并不会被继承；

  相反，派生类将获得多个继承的构造函数，其中每个构造函数分别省略一个含有默认实参的形参

> **测试一下**
>
> 假设基类中三个形参 其中两个有默认实参，那么派生类会产生3个吗?

- 默认、拷贝和移动构造函数不会被继承；
    ```

> 继承的构造函数不会被派生类当作默认构造函数来使用

- 派生类可以继承一部分构造函数，而为其他构造函数定义自己的版本；
  - 如果派生类中定义的构造函数与基类的构造函数具有相同的参数列表，则该构造函数不会被继承

    > 如果派生类含有自己的成员，如果它的构造函数需要处理自己的成员，那么不就永远不会和基类构造函数相同了？

容器与继承

当用容器存放继承体系中的对象时，通常必须才去间接存储的方式

> 因为引用不是一个对象，所以间接的方式特指指针方式

- 不能把具有继承关系的多种类型的对象存放在容器中

```
vector<Bulk_Quote> vt;      // 无法保存Quote对象
vector<Quote> vt;       // 虽然可存放Bulk_Quote对象，但是这些对象已经不是Bulk_Quote对象了
```

- 当派生类对象被赋值给基类对象时，其中的派生类部分将被“切掉”；

  因此容器和存在继承关系的类型无法兼容

在容器中放置（智能）指针而非对象

- 允许将派生类的智能指针转换为基类的智能指针

对于C++面向对象的编程来说，一个悖论是我们无法使用对象进行面向对象编程。

相反，我们必须使用指针和引用

假设一个类，需要保存Bulk_Quote和Quote，我们必须得类定义指针成员才能处理这两个继承关系的类。

```
Basket bsk;
bsk.add_item(make_shared<Quote>(...));
bsk.add_iter(make_shared<Bulk_Quote>(.....));
```

我们考虑如何隐藏指针——模拟虚拷贝

```
class Quote {
    public:
```

```
3         virtual Quote *clone() const & { return new Quote(*this); }
  // 拷贝
4         virtual Quote *cloen() const && { return new
  Quote(std::move(*this)); }     // 移动
5 };
6 class Bule_Quote {
7     public:
8         Bulk_Quote *clone() const & { return new Bulk_Quote(*this);}
9         Bulk_Quote *clone() const && { return new
  Bulk_Quote(std::move(*this)); }
10 };
11
12 class Basket {
13     public:
14         void add_item (const Quote& sale) {
15             items.insert(std::shared_ptr<Quote>(sale.clone()));
16         }
17         void add_item (Quote&& sale) {
18             items.insert(std::shared_ptr<Quote>
  (std::move(sale.clone()));
19         }
20 };
21   bsk.add_item(qt);
```

- `sale.clone()` 保证返回正确类型的指针

  利用基类的引用，将区分指针类型的工作封装起来，用户只需传入对象即可