第15章 面向对象程序设计

15.1 OOP：概述

1. **面向对象程序设计**(object-oriented programming)的核心思想是：**数据抽象**、**继承**和**动态绑定**。
2. 数据抽象：可以将类的接口与实现分离。
3. 继承（inheritance）：可以定义相似的类型并对其相似关系建模。
4. 动态绑定(dynamic binding)：可以在一定程度上忽略相似类型的区别，而以统一的方式使用它们的对象。
5. 动态绑定有时又被称为**运行时绑定(run-time binding)**
6. 在C++中，当使用基类的引用（或指针）调用一个虚函数时将发生动态绑定。

15.2 定义基类和派生类

15.2.1 定义基类

1. 基类通常都应该定义一个**虚析构函数**，即使该函数不执行任何实际操作也是如此。
2. c++中，基类必须将它的两种成员函数区分开：一种是基类希望其派生类进行覆盖的函数（虚函数）。一种是基类希望派生类直接继承而不要改变的函数。
3. 成员函数如果没被声明为虚函数，则其解析过程发生在**编译**时而非**运行**时。

15.2.2 定义派生类

1. **访问说明符**（public、protected和private）的作用是：控制派生类从基类继承而来的成员是否对派生类的用户可见。
2. 如果派生类没有覆盖其基类中的某个虚函数，则该虚函数的行为类似于其他的普通成员，派生类会直接继承其在基类中的版本。
3. c++标准并没有明确规定派生类的对象在内存中如何分布。一个对象中，继承自基类的部分和派生类自定义的部分在内存中不一定是连续存储的。
4. 尽管在派生类对象中含有从基类继承而来的成员，但是派生类并不能直接初始化这些成员。和其他创建了基类对象的代码一样，派生类也必须使用基类的构造函数来初始化它的基类部分。
5. 如果基类定义了一个静态成员，则在整个继承体系中只存在该成员的唯一定义。不论从基类中派生出多少个派生类，对于每个静态成员来说都只存在唯一的实例。

6. 如果我们想将某个类用作基类，则该类必须已经定义而非仅仅声明。
7. 一个类不能派生它本身。
8. 最终的派生类将包含它的直接基类的子对象以及每个间接基类的子对象。
9. c++11新标准提供了一种防止继承发生的方法，即在类名后跟一个关键字 `final` 。

15.2.3 类型转换与继承

1. 我们可以将基类的指针或引用绑定到派生类对象上。当使用基类的引用（或指针）时，实际上我们不清楚该引用（或指针）所绑定对象的真实类型。
2. 智能指针类也支持派生类向基类的类型转换，可以将一个派生类对象的指针存储在一个基类的智能指针内。
3. 当使用存在继承关系的类型时，必须将一个变量或其他表达式的**静态类型（static type）**与该表达式表示对象的**动态类型（dynamic type）**区分开来：
   (1) 表达式的静态类型在编译时总是已知的，它是变量声明时的类型或表达式生成的类型。
   (2) 动态类型则是变量或表达式表示的内存中的对象的类型。动态类型直到运行时才可知。
   (3) 如果表达式既不是引用也不是指针，则它的动态类型永远与静态类型一致。
4. 不存在从基类向派生类的**隐式**类型转换。

```
class Quote {...}
class Bulk_quote : public Quote {...}

Quote base;                    // 基类对象
Bulk_quote derived;            // 派生类对象

Quote* baseP1 = &derived;      // 正确：基类指针绑派生类对象
Quote& baseRef = derived;      // 正确：基类引用绑派生类对象

Bulk_quote* bulkP1 = &base;    // 错误：不能派生类指针绑基类对象
Bulk_quote& bulkRef = base;    // 错误：不能派生类引用绑基类对象

Quote* baseP2 = &derived;      // 正确：基类指针绑派生类对象
Bulk_quote* bulkP2 = baseP2; // 错误：不能将基类转换成派生类
```

5. 编译器在编译时无法确定某个特定的转换在运行时是否安全，因为编译器只能通过检查指针或引用的静态类型来推断该转换是否合法。但我们可以使用**显式**的方式进行类型转换：
   (1) 如果在基类中含有一个或多个虚函数，我们可以使用 `dynamic_cast` 请求一个类型转换，该转换的安全检查将在运行时执行。
   (2)如果我们已知某个基类向派生类的转换是安全的，则可以使用 `static_cast` 来强制覆盖掉编译器的检查工作。
6. 当我们用一个派生类对象为一个基类对象初始化或赋值时，只有该派生类对象中的基类部分会被拷贝、移动或赋值，它的派生类部分将被忽略掉。

```
Bulk_quote bulk;     // 派生类对象
Quote item(bulk);    // 使用Quote::Quote(const Quote&)构造函数
item = bulk;         // 调用Quote::operator=(const Quote&)
```

15.3 虚函数

1. OOP的核心思想是**多态性（polymorphism）**。引用或指针的静态类型与动态类型不同这一事实正是c++支持多态性的根本所在。
2. 当且仅当对通过指针或引用调用虚函数时，才会在运行时解析该调用，也只有在这种情况下对象的动态类型才有可能与静态类型不同。
3. 基类中的虚函数在派生类中隐含地也是一个虚函数。当派生类覆盖了某个虚函数时，该虚函数在基类中的形参必须与派生类中的形参严格匹配。返回值也是如此。
4. 上述规定有个例外：当类的虚函数返回类型是类本身的指针或引用时，上述规则无效。也就是说，如果D由B派生得到，则基类的虚函数可以返回B*而派生类的对于函数可以返回D*，只不过这样的返回类型要求从D到B的类型转换是可访问的。
5. 派生类如果定义了一个函数与基类中虚函数的名字相同但是**形参列表不同**，这是合法的。这时派生类的函数并不会覆盖掉基类中的版本。但是就实际编程习惯来说这样做往往意味着该情况是不小心将虚函数的**形参列表搞错**了。
6. 我们可以使用 `override` 关键字标记了某个虚函数。但该函数并没有覆盖已存在的虚函数，此时编译器将会报错。
7. 我们还能把某个函数指定为 `final` ，之后任何尝试覆盖该函数的操作将引发错误。

```
struct B {
    virtual void f1(int) const;
    virtual void f2();
    void f3();
    virtual f4() final;
}
struct D1 : public B {
    void f1(int) const override; // 正确：f1与基类中的f1匹配
    void f2(int) override;       // 错误：B没有形如f2(int)的函数
    void f3() override;          // 错误：f3不是虚函数
    void f4() override;          // 错误：f4已经被声明成final
    void f5() override;          // 错误：B没有名为f5的函数
}
```

8. 如果虚函数使用默认实参，则基类和派生类中定义的默认实参最好一致。**如不一致**，无论是基类还是派生类调用该虚函数，**都只会使用基类的默认实参。**

9. 使用**作用域运算符**可以实现对虚函数的调用不进行动态绑定，而是强迫其执行虚函数的某个特定版本。

```
double undiscouted = baseP->Quote::net_price(42);
```

10. 通常情况下，只有成员函数（或友元）中的代码才需要使用作用域运算符来回避虚函数机制。
11. 如果一个派生类虚函数需要调用它的基类版本，但是没有使用作用域运算符，则在运行时该调用将被解析为对派生类版本自身的调用，从而导致无限递归。

15.4 抽象基类

1. 通过在函数体的位置（即在声明语句的分号之前）书写 `=0` 就可以将这个虚函数说明为**纯虚函数（pure virtual function）**，这样就清晰的告诉用户当前这个虚函数没有实际的意义。
2. 我们无须为纯虚函数提供实现。当然我们也能为纯虚函数提供实现，不过函数体必须定义在类的外部。
3. 含有（或者未覆盖直接继承）纯虚函数的类是**抽象基类（abstract base class）**。抽象基类负责定义接口，而后续的其他类可以覆盖该接口。我们**不能（直接）创建一个抽象基类的对象**。

15.5 访问控制与继承

1. 一个类使用 `protected` 关键字来声明那些它希望与派生类分享但是不想被其他公共访问使用的成员。
   (1) 和私有成员类似，受保护的成员对于类的用户来说是不可访问的。
   (2) 和公有成员类似，受保护的成员对派生类的成员和友元来说是可访问的。
   (3) 派生类的成员或友元只能通过派生类对象来访问基类的受保护成员。派生类对于一个基类对象中的受保护成员没有任何访问特权。

```
class Base {
protected:
    int prot_men;    // 受保护的成员
};
class Sneaky : public Base {
    friend void clobber(Sneaky&);    // 能访问Sneaky::prot_mem
    friend void clobber(Base&);      // 不能访问Base::prot_mem
    int j;                           // j默认是private
};
// 正确：clobber能访问Sneaky对象的private和protected成员
void clobber(Sneaky& s) { s.j = s.prot_mem = 0; }
// 错误：Clobber不能访问Base的Protected成员
void clobber(Base& b) { b.prot_mem = 0; }
```

2. 某个类对其继承而来的成员的访问权限受到两个因素的影响：
   (1) 在基类中该成员的访问说明符。
   (2) 在派生类的派生列表中的访问说明符。
3. 其中 **派生访问说明符** 对 **派生类的成员（及友元）** 能否访问其直接基类的成员没有什么影响。对基类成员的访问权限只与**基类中的访问说明符**有关。

```
class Base {
public:
    void pub_mem();
protected:
    int prot_men;
private:
    char priv_mem;
};
struct Pub_Derv : public Base {
    // 正确：派生类能访问public成员
    void e() { pub_mem(); }
    // 正确：派生类能访问protected成员
    int f() { return prot_mem; }
    // 错误：private 成员对于派生类来说是不可访问的
    char g() { return priv_mem; }
};
struct Prot_Der : protected Base {
    // proteced 不影响派生类的访问权限
    // 同上... ...
};
struct Priv_Der : private Base {
    // private 不影响派生类的访问权限
    // 同上... ...
};
```

4. 派生访问说明符的目的是控制**派生类用户（包括派生类的派生类在内）** 对于基类成员的访问权限：

```
// 派生访问说明符 控制 派生类用户 的访问权限：
Pub_Derv d1;     // 继承自Base的成员是public的
Prot_Derv d2;    // 继承自Base的成员是protected的
Priv_Derv d3;    // 继承自Base的成员是private的
d1.pub_mem();    // 正确：pub_mem在派生类中是public的
d2.pub_mem();    // 错误：pub_mem在派生类中是protected的
d3.pub_mem();    // 错误：pub_mem在派生类中是private的

// 派生访问说明符 控制 派生类的派生类 的访问权限：
struct Derived_from_Public : public Pub_Derv {
    // 正确：Base::Prot_mem在Pub_Derv中仍然是protected的
    int use_base() { return prot_mem; }
}
struct Derived_from_Protected : public Prot_Derv {
    // 正确：Base::Prot_mem在Prot_Derv中仍然是protected的
    int use_base() { return prot_mem; }
}
struct Derived_from_Private : public Priv_Derv {
    // 错误：Base::prot_mem在Priv_Derv中是Private的
    int use_base() { return prot_mem; }
};
```

5. 派生访问说明符 对 **派生类用户** 的影响总结：
   (1) `public` 继承时，基类的成员的权限不受影响。
   (2) `protected` 继承时，基类的 `public` 成员会转为 `protected` 成员，其余的权限不变。
   (3) `private` 继承时，基类的 `public` 和 `protected` 成员均会转为 `private` 成员，也就是说所有成员都成 `private` 的了。
6. 对于代码中的某个给定节点来说，基类的公有成员是可访问的，则派生类向基类的类型转换也是可访问的，反之则不行。
7. **类**有三种用户：
   (1) 普通用户：普通用户编写的代码使用类的对象，这部分代码只能访问类的公有（接口）成员；
   (2) 类的实现者：负责编写类的成员和友元的代码，成员和友元既能访问类的公有部分，也能访问类的私有（实现）部分。
   (3) 派生类：基类把它希望派生类能够使用的部分声明成受保护的。普通用户不能访问受保护的成员，而派生类及其友元仍旧不能访问私有成员。
8. 和其他类一样，基类应该将其接口成员声明为**公有**的，同时将属于其实现细节的函数部分分成两组：
   (1) 一组可供派生类访问，应该声明为**受保护**的，这样派生类就能在实现自己的功能时使用基类的这些操作和数据；
   (2) 另一组只能由基类及基类的友元访问，应该声明为**私有**的。
9. 友元关系不能传递，同样友元关系也不能继承。
10. 每个类负责控制各自成员的访问权限（*基类的访问权限由基类本身控制，也就是说即使对于派生类的基类成员的权限，也不由派生类控制而由基类控制）。

11. 通过在类的内部使用 `using` 声明语句，可以将该类的直接或间接基类中的**任何可访问成员（非私有成员）** 标记出来。 `using` 声明语句中名字的访问权限由该 `using` 声明语句之前的访问说明符来决定:

```
class Base {
public:
    std::size_t size() const { return n; }
protected:
    std::size_t n;
private:
    int i;
}

// 注意是private继承
class Derived : Private Base {
// 正确
public:
    using Base::size;
protected:
    using Base::n;

// 错误：只能为非私有成员提供声明
public:
    using Base::i;
protected:
    using Base::i;
}
```

12. 使用 `class` 关键字定义的派生类是私有继承的，使用 `struct` 关键字定义的派生类是公有继承的。
13. `class` 关键字和 `struct` 关键字**唯一**的差别就是默认成员访问说明符及默认派生访问说明符。除此之外再无其他不同之处。

```
class Base {};
struct D1 : Base {};    // 默认public继承
class D2 : Base {};     // 默认private继承
```

15.6 继承中的类作用域

1. 每个类定义自己的作用域，在这个作用域内我们定义类的成员。
2. 当存在继承时，派生类的作用域嵌套在其基类的作用域之内。

3. 如果一个名字在派生类的作用域内无法正确解析，则编译器将继续在外层的基类作用域中寻找该名字的定义。
4. 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。即使静态类型与动态类型可能不一致（当使用基类的因引用或指针时会发生这种情况），但是我们能使用哪些成员仍然是由静态类型决定的。
5. 派生类的成员将隐藏同名的基类成员。
6. 可以通过作用域运算符来使用一个被隐藏的基类成员。
7. 除了覆盖继承而来的虚函数之外，派生类最好不要重用其他定义在基类中的名字。
8. 声明在内层作用域的函数并**不会重载**声明在外层作用域的函数，因此，定义在派生类中的函数**不会重载**其基类中的成员。
9. 如果派生类（既内层作用域）的成员与基类（既外层作用域）的某个成员**同名**，则派生类将在其作用域内**隐藏**该基类成员。**即使派生类成员和基类成员的形参列表不一致，基类成员也仍然会被隐藏掉。**
10. 成员函数无论是否是虚函数都可以被重载，如果派生类希望所有的重载版本对于它来说都是可见的，那么它需要覆盖所有的版本，或者一个也不覆盖。当为重载的成员提供一条 `using` 声明语句就无须覆盖基类中的每一个重载版本了，此时派生类只需要定义其特有的函数就可以了，而无须为继承而来的其他函数重新定义。

15.7 构造函数与拷贝控制

15.7.1 虚析构函数

1. 继承关系对基类拷贝控制最直接的影响是基类通常应该定义一个**虚析构函数**，这样就能动态分配继承体系中的对象了。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。
2. 虚析构函数将阻止合成移动操作。

15.7.2 合成拷贝控制与继承

1. 基类或派生类的合成拷贝控制成员的行为与其它合成的构造函数、赋值运算符或析构函数类似：它们对类本身的成员依次进行初始化、赋值或销毁的操作。此外，这些合成的成员还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、赋值或销毁的操作。

15.7.3 派生类的拷贝控制成员

1. 当派生类定义了拷贝或移动操作时，该操作负责拷贝或移动包括基类部分成员在内的整个对象。

2. 默认情况下，基类默认构造函数初始化派生类对象的基类部分。如果我们想拷贝（或移动）基类部分，则必须在派生类的构造函数初始值列表中显式地使用基类的拷贝（或移动）构造函数。
3. 派生类的赋值运算符也必须显式地为其基类部分赋值。
4. 在析构函数执行完成后，对象的成员会被隐式销毁。类似的，对象的基类部分也是隐式销毁的。因此派生类析构函数只负责销毁由派生类自己分配的资源。
5. 对象销毁的顺序正好与创建的顺序相反：派生类析构函数首先执行，然后是基类的析构函数，以此类推，沿着继承体系的反方向直至最后。
6. 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相对应的虚函数版本。

15.7.4 继承的构造函数

1. 在c++11新标准中，派生类能够重用其直接基类定义的构造函数。
2. 通常情况下 `using` 声明语句只是令某个名字在当前作用域内可见。而当用作构造函数时， `using` 声明语句将令编译器产生代码。

```
class Bulk_quote : public Disc_quote {
public:
    using Disc_quote::Disc_quote;    // 继承Disc_quote的构造函数
}

// 编译器生成的构造函数等价于:
Bulk_quote(const std::string& book, double price,
           std::size_t qty, double disc):
    Disc_quote(book, price, qty, disc) {}
```

3. 一个构造函数的 `using` 声明不会改变该构造函数的访问级别。
4. 一个构造函数的 `using` 声明不能指定 `explicit` 或 `constexpr` 。
5. 当一个基类构造函数含有默认实参时，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掉一个含有默认实参的形参。
6. 如果基类含有几个构造函数，则除了两个例外情况，大多数时候派生类会继承所有这些构造函数。
   (1) 第一个例外是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。
   (2) 第二个例外是默认、拷贝和移动构造函数不会被继承。这些构造函数按照正常规则被合成。

15.8 容器与继承

1. 当我们使用容器存放继承体系中的对象时，通常必须采取间接存储的方式。因为不允许在容器中保存不同类型的元素，所以我们不能把具有继承关系的多种类型的对象直接存放在容器当中。

2. 当派生类对像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。
3. 当我们希望在容器中存放具有继承关系的对象时，我们实际上存放的通常是基类的指针（更好的选择是智能指针），这些指针所指的对象的动态类型可能是基类类型，也可能是派生类类型。

15.9 文本查询程序再探

1. 用面向对象的思想来写这个文本查询程序。