面向对象程序设计

基本概念

1. 面向对象程序设计基于三个基本概念：数据抽象、继承和动态绑定。
2. 某些函数，基类希望它的派生类能各自定义适合自身的版本，因此基类会把这些函数声明为虚函数。派生类必须在其内部对所有重新定义的虚函数进行声明，可以在这样的函数之前加上virtual关键字，但并不是必须的。C++11新标准允许派生类显式地注明它将使用哪个成员函数改写基类的虚函数————在该函数的形参列表之后增加一个“override”关键字。
3. 函数的运行版本由实参决定，这就是动态绑定。**C++中，当我们使用基类的引用或指针调用一个虚函数时将发送动态绑定**。

   ```
   void func(基类类型的引用 obj){
       //调用obj的虚函数
   }
   ```

   函数会根据obj的类型决定要执行虚函数对应的版本。

基类和派生类

1. 基类通常应该定义个虚析构函数，即使该函数不执行任何实际操作
2. virtual关键字只能出现在类内部的声明语句，不能用于类外部的函数定义。
3. 成员函数如果没有被声明为虚函数，则其解析过程发送在编译时而非运行时。
4. 派生类可以继承定义在基类中的成员，但是**派生类的成员函数不一定有权访问从基类继承而来的成员**。派生类能访问基类的共有成员，不能访问私有成员。若**基类希望它的派生类有权访问某成员，同时禁止其他用户访问，则用protected说明这样的成员**。
5. 类派生列表中的访问说明符用于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。
6. 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。
7. **因为在派生类对象中含有与其基类对应的组成部分，所以我们能把派生类的对象当成基类对象来使用，而且我们也能将基类的指针和引用绑定到派生类对象中的基类部分上**。因此，当使用基类的引用或指针时，我们并不清楚该引用或指针所绑定对象的真实类型。
   而由于基类的对象可能是派生类对象的一部分，也可能不是，所以不存在从基类向派生类的自动类型转换。智能指针也支持派生类向基类的类型转换。

```
//假设B派生自A
A item;
B b;
A *p=&item; //p指向A对象
p=&b;       //p指向b的A部分
A &r=b;     //r绑定到b的A部分
```

8. 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。无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。静态成员也遵循同通用的访问控制规则，如果是private则派生类无权访问。
9. **派生类在声明时不能包含派生列表**。因为一条声明语句的目的是令程序知道某个名字的存在以及该名字表示的实体，
10. 作为基类的类必须已经被定义，这隐含地说明一个类不能派生它本身。
11. C++11提供了一个关键字final用于防止类被用做基类。

```
class NoDerived final {}
```

12. 表达式的静态类型在编译时是已知的，是变量声明时的类型或表达式生成的类型（这也说明了编译器在编译时只能通过检查指针或引用的静态类型来推断是否合法）；动态类型则是变量或者表达式表示的**内存中**的对象的类型，直到运行时才可知。
13. 虽然我们可以将派生类对象赋值给基类对象，但这要求基类需要定义了形参为引用的拷贝构造函数，且赋值后派生类自定义的成员会被忽略。

虚函数

1. **所有的虚函数都必须有定义**，因为编译器直到运行时才知道调用了哪个版本的函数。
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. **一旦某个函数被声明成虚函数，则在所有派生类中它都是虚函数**，所以在派生类中不是必须要再次用关键字virtual指出。
4. 一般情况下派生类中虚函数的返回类型必须与基类函数的返回类型匹配。但有一个例外，就是当基类的虚函数返回类型是类本身的指针或引用时，则可以不同（是定义该虚函数的类的类型）。即若D由B派生得到，则B的虚函数可以返回B*而D的对应函数可以返回D*，但是要求从D到B的类型转换是可访问的。
5. final关键字除了指定类，使得类无法被继承外，还可以指定函数。将函数定义为final后，任何尝试**覆盖（覆盖不等于重载）**该函数的操作都将引发错误。**final只能限定虚函数**。
6. 可以用作用域运算符强制执行虚函数的某个特定版本。

抽象基类

1. 纯虚函数：一个纯虚函数无需定义，将函数体用“=0”代替即可，表明该函数没有实际意义，只是用于被其他函数覆盖(函数前可以不用virtual声明)。我们也可以为纯虚函数提供定义，但只能写在类外。
2. **含有纯虚函数的类是抽象基类**。抽象基类负责定义接口，而后续的其他类可以覆盖该接口。**我们不能定义抽象基类的对象，且抽象基类的派生对象必须定义了覆盖抽象基类的纯虚函数的函数，才可以被定义.**

访问控制与继承

1. 一个类使用“protected”关键字来声明那么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。
   - 受保护的成员对于类的用户来说是不可访问的。
   - 受保护的成员对于派生类的成员和友元来说是可访问的。
   - **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。对于一个基类对象（这个对象不是派生类对象的子对象）中的受保护成员，派生类没有任何访问特权。**

```
class Base{
    protected:
        int prot_mem;
};
class Derived:public Base{
    friend void func(Derived&);
    friend void func(Base&);
};
//正确 友元可以通过派生类Derived对象来访问基类的受保护成员
//因为友元可以访问友元类的所有成员，这些成员就包括了基类子对象
void func(Derived &d){ d.prot_mem=0; }
//错误 func 和Base并不是友元关系 Base的受保护成员只能在自己类内访问
void func(Base &b) { b.prot_mem=0; }
```

2. 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影响，其目的是控制**派生类用户**对于基类成员的访问权限。（即以什么样的方式去继承基类的成员）

```
public_derived d1;      //以public的方式继承自base
private_derived d2;     //以private的方式继承自base
//pub_mem()是base中的公有成员 则
d1.pub_mem();   //正确
d2.pub_mem();   //错误 pub_mem在派生类d2中是私有的
```

3. 派生类向基类转换的可访问性：假设D继承自B
   - 只有当D公有地继承自B时，**用户代码**才能使用派生类向基类的转换
   - 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
   - 若D以公有或者受保护的方式继承自B，则**D的派生类的成员和友元**可以使用D向B的类型转换。

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。当存在继承关系时，**派生类的作用域嵌套在其基类的作用域之内，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**
2. 函数调用的解析过程：假定调用p->mem()或者obj.mem()
   - 首先确定p或者obj的静态类型。因为我们调用的是一个成员，所以该类型必须是类类型；
   - 在p或者obj的静态类型对应的类中查找mem。如果找不到则依次在直接基类中不断查找直到到达继承链的顶端。若找遍了该类及基类仍然找不到，则编译器报错；
   - 一旦找到了mem，就进行常规的类型检查以确认对于当前找到的mem，本次调用是否合法；（**就算不合法也不忘基类找了**）
   - 若调用合法，则编译器将根据调用的是否是虚函数而产生不同代码：
     - 若mem是虚函数且通过指针或引用调用，则编译器产生的代码将在运行时确定到底运行该虚函数的哪个版本，依据是对象的动态类型
     - 若mem不是虚函数或者我们是通过对象调用的，则编译器将产生一个常规函数调用
3. 若派生类与基类的某个成员同名，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，基类成员也会被隐藏**。（往作用域方向考虑即可）
4. 在类内部使用using声明语句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，也就是说若一条using声明语句出现在类的private部分则该语句声明的名字只能被类的成员和友元访问；若using出现在类的public部分则可以被类的所有用户访问。

构造函数和拷贝控制

1. 当delete一个动态分配的对象的指针时将执行析构函数，若该指针指向继承体系中的某个类型，则有可能出现**指针的静态类型与被删除对象的动态类型不符**的情况。因此通常基类要定义虚析构函数。（若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. 派生类的赋值运算符必须显示地为其基类部分赋值：

```
Derived& Derived::operator=(const Derived& rhs){
    Base::operator=(rhs);
    //为派生类成员赋值
    return *this;
}
```

3. 派生类析构函数只负责销毁由派生类自己分配的资源。
4. **若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**。因为构建/析构一个对象时，需要把对象的类和构造/析构函数的类看作是同一个，对虚函数的调用绑定也符合这种要求，析构函数也是。

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，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。例如，若基类有一个接受两个形参的构造函数，其中第二个形参含有默认实参，则派生类将获得两个构造函数：一个构造函数接受两个形参（没有默认实参），另一个构造函数只接受一个形参。

容器与继承

1. 当我们希望在容器中存储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。