Abstract Base Classes

Pure Virtual Functions

In practice, we'd like to prevent users from creating objects for some classes at all. Such classes represent the general concept, not concrete classes.

We can enforce this design intent by defining function as a **pure virtual** function. Unlike ordinary virtuals, a pure virtual function does not have to be defined. We specify a function is a pure virtual by

```
virtual void fun()=0;
```

The declaration given inside class has no function body. And the `=0` may appear only on the declaration of a virtual function in the class body.

```
class Quote{
    public:
        std::string isbn() const;
        virtual double net_price(std::size_t n) const;
};
class Disc_quote:public Quote{
    public:
        Disc_quote()=default;
        Disc_quote(const std::string &book,double price,std::size_t qty,double
disc):Quote(book,price),quantity(qty),discount(disc){}

        double net_price(std::size_t) const =0;// pure virtual function,
inheriting from Quote

    protected:
        std::size_t quantity=0;
        double discount=0.0;
};
```

It is worth noting that we can provide a definition for a pure virtual. However, the function body must be defined outside the class. That is, we cannot provide a function body inside the class for a function that is `=0`.

Classes with Pure Virtuals Are Abstract Base Classes

A class containing (or inheriting without overriding,继承抽象类但是没有重写纯虚函数也是抽象类) a pure virtual is an **abstract base class**). An abstract base class defines an interface for subsequent classes to override. We cannot (directly) create objects of a type that is an abstract base class. We can define objects of classes that inherit those abstract base classes, so long as such classes override pure virtual function.

```
// Ab is an abstract class.
AB obj;// error, can't define a AB object
Derived_AB obj;// ok, Derived_AB has no pure virtual functions
```

A Derived Class Constructor Initializes Its Direct Base Class Only

We can rewrite the definition of class `Bulk_quote`.

```
class Bulk_quote:public Disc_quote{
    public:
        Bulk_quote()=default;
        Bulk_quote(const std::string &book,double price,std::size_t qty,double
disc):Disc_quote(book,price,qty,disc){}

        // overrides the base version
        double net_price(std::size_t )const override{
            //...
        }
};
```

This version of `Bulk_quote` has a direct base class, `Disc_quote`, and an indirect base class, `Quote`. Each `Bulk_quote` object has three subobjects: an (empty) `Bulk_quote` part, a `Disc_quote` subobject, and a `Quote` subobject.

Each class controls the initialization of objects of its type. Therefore, even though `Bulk_quote` has no data of its own, it provides the same four-argument constructor as in our original class. Our new constructor passes its arguments to the `Disc_quote` constructor. That constructor in turn runs the `Quote` constructor. The `Quote` constructor initializes the `bookNo` and `price` members of `bulk`. When the `Quote` constructor ends, the `Disc_quote` constructor runs and initializes the `quantity` and `discount` members. At this point, the `Bulk_quote` constructor resumes. That constructor has no further initializations or any other work to do.

It is worth noting that we need to initialize members of base class by call the base-class constructor and we cannot directly make it initialized by initializer list.

```
class Base{
    protected:
        int val;
    public:
        Base()=default;
        Base(const int &v):val(v){}// initializer list
};
class Derived:public Base{
    string str;
    public:
        Derived(const int &v,string &s):val(v),str(s){}// error
        Derived(const int &v,string &s):Base(v),str(s){}// ok
};
```

也就是说，派生类初始化列表不可以直接初始化基类成员，必须通过基类构造函数间接初始化构造，不可以“越级初始化”。