基类和派生类

派生类继承基类的成员（成员函数和数据成员）：

- 成员函数而言，派生类只是在类中存储了基类成员函数的地址而已（前提是有访问基类的成员函数的权限）。
- 数据成员，暂不清楚，先略过。

Defining a Base Class

```
class Quote{
    public:
        Quote()=default;// default constructor systhesized by compiler
        Quote(const std::string &book, double
sales_price):bookNo(book),price(salse_price){}

        std::string isbn() const{return bookNo;}

        virtual double net_price(std::size_t n) const {return n*price;}

        virtual ~Quote()=default;// dynamic binding for the destructor

    private:
        std::string bookNo;
    protected:
        double price=0.0;
};
```

成员函数和继承

> C++中，基类中有两类成员函数：1）基类希望其派生类进行override的函数，2）一种是基类直接继承不需改变的函数（派生类需要访问权限才可访问）。前者定义为**virtual function**。

1. 任何**构造函数之外**的**非static函数**都可以是虚函数；基类（继承层的root）类内声明虚函数时，需要`virtual`关键字，类外定义不需要。同时派生类也不需要，因为派生类的函数隐式定义为虚函数。
2. 基类中的虚函数，则该函数在派生类**隐式地**是虚函数，即派生类中的虚函数不需要加上`virtual`关键词，当然写上会更好。同时，派生类的虚函数也可以被其派生类继续继承并继续override。
3. 如果派生类没有override基类的虚函数，则该虚函数和普通的成员函数一样（前提是有访问权限），直接被继承，使用基类中的版本
4. 没被声明为虚函数的成员函数，其解析过程在编译时就确定了。因此其不需要“延迟绑定”，编译器即可将非虚函数的函数地址链接到相应的位置。
5. 最顶层的基类总是需要定义一个**virtual destructor**，即使该虚析构函数什么实际工作也没做。
6. 派生类中的`override`关键词是为了提醒用户，该函数是`override`基类的函数，同时是为了指明被修饰的函数**必须**是override基类的函数的函数，否则会报错，

> 即该override关键字仅仅是为了帮助debug的，怕用户写错函数名，从而定义了一个不是override基类函数的函数。

7. 只有当通过基类的指针或引用（实质上也是指针），才可以实现动态绑定。

访问控制和继承

派生类不一定有权限访问基类的成员。可以访问：

- 公有成员
- 受保护成员（**pretected**），只可被派生类访问，禁止其他用户访问。
  不可以访问：
- 私有成员

1. 继承基类时候，可以指定访问说明符，来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。
   Defining a Derived Class

```
class Bulk_quote:public Quote{
    Bulk_quote()=default;

    Bulk_quote(const std::string &,double, std::size_t,double);

    // overrides the base version
    double net_price(std::size_t)const override;
    private:
    std::size_t min_qty=0;
    double discount=0.0;
};
```

Virtual Functions in the Derived Class

If a derived class does not override a virtual from its base, then, like any other member, the derived class inherits the version defined in its base class.

Derived-Class Objects and the Derived-to-Base Conversion

A derived object contains multiple parts: a subobject containing the (`nonstatic`) members defined in the derived class itself, plus subobjects corresponding to each base class from which the derived class inherits. Thus, a `Bulk_quote` object will contain four data elements: the `bookNo` and `price` data members that it inherits from `Quote`, and two members, which are defined by `Bulk_quote`.

> Although the standard does not sepcify how derived objects are laid out in memory, we can think of a `Bulk_quote` object as consisting of two parts.

[bookNo price min_qty discount]
The first part is data members inheriting from the base class `Quote`, and the second part is data members defined in the derived class `Bulk_quote`.

> The base and derived parts of an object are not guaranteed to be stored contiguously. This is a conceptual, not physical, representation of how classes work.

Because a derived object contains subparts corresponding to its base class(es), we can use an object of a derived type as if it were an object of its base type(s). In particular, we can bind a base-class reference or pointer to the base-class part of a derived object. That is,

```
Quote item; // object of base type
Bulk_quote bulk; // object of derived type

Quote *p=&item; // p points to a Quote object
p=&bulk; // p points to the Quote part of bulk

Quote &r=bulk; // r bound to the Quote part of bulk
```

This conversion is often referred to as the **derived-to-base** conversion. As with any other conversion, the compiler will apply the derived-to-base conversion implicitly.

The fact that the derived-to-base conversion is implicit means that we can use an object of derived type or a reference to a derived type when a reference to the base type is required. Similarly, we can use a pointer to a derived type where a pointer to the base type is required.

> The fact that a derived object contains subobjects for its base classes is key to how inheritance works.

Derived-Class Constructor

Altough a derived object contains members that it inherits from its base, it cannot directly initialize those members. A derived class must use a base-class constructor to initialize its base-class part.

> Each class controls how its members are initialized.

Example as to how a derived-class define its constructor:

```
Bulk_quote(const string &book,double p, size_t qty, double
disc):Quote(book,p),min_qty(qty),discount(disc){}
```

The base constructor initializes the derived-class's base-class part (i.e., the `bookNo` and `price` members). ***When the base constructor body completes,*** the base-class part of the object being constructed will have been initialized. Next the direct members, `min_qty` and `discount`, are initialized. Finally, the function body of the `Bulk_quote` constructor is run.

Using Members of the Base Class from the Derived Class

A derived class may access the `public` and `protected` members of its base class:

```
double Bulk_quote::net_price(size_t cnt)const {
    if(cnt>=min_qty) return cnt*(1-discount)*price;
    else
        return cnt*price;
}
```

It's worth noting that the scope of a derived class is nested inside the scope of its base class. As a result, there is no distinction between how a member of the derived class uses members defined in its own class (e.g., `min_qty` and `discount`) and how it uses members defined in its base (e.g., `price`).

Inheritance and `static` Members

> If a base class defines a `static` member, there is only one such member defined for the entire hierarchy.

```
class Base{
    public:
        static void foo();// static member obey normal access control
        //                    thus, foo() is defined as public member
};
class Derived:public Base{
    public:
        void f(const Derived &);
};

void Derived::f(const Derived &derived_obj){
    Base::foo();// ok, Base defines foo
    Derived::foo();// ok, Derived inherits foo

    // ok, derived objects can be used to access static from base
    derived_obj.foo();// accessed through a Derived object
    foo();// ok,accessed through this object,i.e., this->foo();
}
```

Declarations of Derived Classes

A derived class is declared like any other class.

```
class Derived:public Base{/*...*/};

Derived obj;// right way to declare a derived class
```

Classes Used as a Base Class

A class must be defined, not just declared, before we can use it as a base class:

```
class Base;// declared but not defined
// error, Base must be defined
class Derived:public Base{/*...*/};
```

The reason should be easy to see: Each deried class contains, and may use, the members it inherits from its base class. To use those members, the derived class must know what they are.

A direct base is named in the ***derivation list***. An indirect base is one that a derived class inherits through its direct base class.

Each class inherits all the members of its direct base class. (对于一个继承链来说，最底层的派生类会继承所有base class和indirect class的所有所有数据成员)

> Remark: 派生类可以继承基类所有的成员，但是只可以访问基类的public和protected成员。对于private成员，访问方法和基类对象一样，必须要使用基类成员函数来访问。而且，派生类的作用范围是在基类的作用范围内，所有派生类可以直接使用基类的public成员。

Preventing Inheritance

We can prevent a class from being used as a base by following the class name with `final`:

```
class NoDerived final{/*...*/};

class Base{/*...*/};

class Last final:Base{/*...*/};// Last is final, we cannot inherit from Last

class Bad:NoDerived{/*...*/};// error, NoDerived is final

class Bad2:Last{/*...*/};// error, Last is final
```

Conversions and Inheritance

We can bind a pointer or reference to a base-class type to an object of a type derived from that base class. The reason is objects of the derived classes have base-class part. (The reasons have been explained in the previous section)

Static Type and Dynamic Type

The ***static type*** of an expression is always known at compile time. The ***dynamic type*** may not be known until run time.

```
double print_total(ostream &os,const Quote &item,size_t n){
    double ret=item.net_price(n);
    ///....
    return ret;
}
```

In this case, the static type of `item` is `Quote &`. The dynamic type depends on the type of the argument to which `item` is bound. That type cannot be known until a call is executed at run time. If we pass a `Bulk_quote` object to `print_total`, then the static type of `item` will differ from its dynamic type. That is, the static type of `item` is `Quote &`, the dynamic type is `Bulk_quote`.

> When we use types related to inheritance, we must distinguish the static type of a variable or other expression to the dynamic type of the object that expression represents.

The dynamic type of an expression that is neither a reference nor a pointer is always the same as that expression's static type. (如果表达式既不是引用也不是指针，则它的动态类型和静态类型一致) For example, a variable of the type `Quote` is always a `Quote` object; there is nothing we can do that will change the type of the object to which that variable corresponds.

There Is No Implicit Conversion from Base to Derived ...

`It is crucial to understand that there exists conversion from derived to base because every derived object contains a base-class part which a pointer or reference of the base-class type can be bound to. There is no similar guarantee for base-class objects. A base-class object can exist either as an independent object or as part of a derived object.`

1. `A base object that is not part of a derived object has only the members defined by the base class;`
2. `A base object that is not part of a derived object doesn't have the members defined by the derived class;`

Because a base object might or might not be part of a derived object, there is no automatic conversion from the base class to its derived class(s):

```
Quote base;
Bulk_quote *bulkp=&base;// error, can't convert base to derived
Bulk_quote &bulkref=base;// error, can't covert base to derived
```

If these assignments were legal, we might attempt to use `bulkp` or `bulkref` to use members that do not exist in `base`.

What is sometimes a bit surprising is that we cannot convert from base to derived even when a base pointer or reference is bound to a derived object:

```
Bulk_quote bulk;
Quote *itemp=&bulk;// ok, dynamic type is Bulk_quote
Bulk_quote *bulkp=itemp;// error, can't convert base to derived
```

The compiler has no way to know (at compile time) that a sepcific conversion will be safe at run time. The compiler looks only at the static type of the pointer or reference to determine whether a conversion is legal. If the base class has one or more virtual functions, we can use a `dynamic_cast` to request a conversion that is checked at run time. Alternatively, in those cases when we know that the conversion from base to derived is safe, we can use a `static_cast` to override the compiler. (如果我们确保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。)

... and No Conversion between Objects

The automatic derived-to-base conversion applies only for conversions to a reference or pointer type. There is no such conversion from a derived-class type to base-class type. Nevertheless, it is often possible to convert an object of a derived class to its base-class type. However, such conversions may not behave as we might want.

Remember that when we initialize or assign an object of a class type, we are actually calling a function. When we initialize, we're calling a constructor; when we assign, we're calling an assignment operator. These members normally have a parameter that is a reference to the `const` version of the class type.

```
class Foo;
Foo &operator=(const Foo &f);// assignment operator
```

The reasons that these members normally have a parameter that is reference to the `const` version of the class type, have been given in the setion **copy, assign, and destroy**.

Because these members take references, the derived-to-base conversion lets us pass a derived object to a base-class copy/move operation. These operations are not virtual. When we pass a derived object to a base-class constructor, the constructor that is run is defined in the base class. That constructor knows only about the members of the base class itself. Similarly, if we assign a derived object to a base object, the assignment operator that is run is the one defined in the base class. That operator also knows only about the members of the base class itself.

```
Bulk_quote bulk;// object of derived type
Quote item(bulk);// use the Quote::Quote(const Quote &) constructor
item=bulk;// calls Quote::operator=(const Quote &)
```

When `item` is constructed, the `Quote` copy constructor is run. That constructor knows only about the `bookNo` and `price` members defined in the `Quote` class. It copies those members from the `Quote` part of `bulk` and ignores the members that are part of the `Bulk_quote` portion of `bulk`. Similarly for the assignment of `bulk` to `item`; only the `Quote` part `bulk` is assigned to `item`.

Because the `Bulk_quote` part is ignored, we say that the `Bulk_quote` portion of `bulk` is **sliced down**.

> When we initialize or assign an object of a base type from an object of a derived type, only the base-class part of the derived object is copied, moved, or assigned. The derived part of the object is ignored.