Constructors and Copy Control

Each class controls what happens when objects of its type are created, copied, assigned, moved, or destroyed. As with any other class, if a class (base or derived) dose not itself define one of the copy-control operations, the compiler will synthesize that operation. Also, the systhesized version of any of these members might be a deleted function.

Virtual Destructor

The primary direct impact that inheritance has on copy control for a base class is that a base class generally should define a virtual destructor. The reason is the destructor needs to be virtual to allow objects in the inheritance hierarhy to be dynamic allocated. (虚函数搭配动态绑定机制实现类的多态，因为必须要将析构函数设置成虚函数，以便于动态绑定)

Thus, we arrange to run the proper destructor by defining the destructor as virtual in the base class:

```
class Base{
    public:
    virtual ~Base()=default;// dynamic binding for the destructor
};
```

As with any other virtual, the virtual nature of the destrucotr is inherited. Therefore, classes derived from base class have virtual destructor, whether they use the synthesized destructor or define their own version. So long as the base class destructor is virtual, when we `delete` a pointer to base, the correct destrcutor will be run:

```
Base * bp=new Base;
delete bp;// destructor ~Base()
bp=new Derived;
delete bp;// destructor ~Derived
```

> 如果，基类的析构函数不是虚函数，那么`delete`一个指向基类的指针（该指针指向派生类对象的base-class部分），该行为是undefined。

The virtual destructor of a base class is one important exception to the rule of thumb that copy and assignment is needed if the destructor is needed.

Virtual Destructors Turn Off Synthesized Move

The fact that a base class needs a virtual destructor has an important indirect impact on the definition of base and derived classes: If a class defines a destructor-even if it uses `=default` to use the synthesized version-the compiler will not synthesize a move operation for that class.

Quesion: what operation must a virtual destructor perform?
A: Release resource that allocated dynamiclly.

Synthesized Copy Control and Inheritance

The code as to `Quote`.

```
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;
};
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{
            //...
        }
};
```

The synthesized copy-control members in a base or a derived class executes like any other synthesized construcotr, assignment operator, or destructor: They memberwise initialize, assign, or destroy the members of the class itself. In addition, these synthesized members initialize, assign, or destroy the direct base part of an object by using the corresponding operation from the base class. For example,

- The synthesized `Bulk_quote` default constructor runs the `Disc_quote`default constructor, which in turn runs the `Quote` default constructor. (These classes have been defined in previous section)
- the `Quote` default constructor default initializes the `bookNo` member to the empth string and uses the in-class initializer to initialize `price` to zero.
- when the `Quote` constructor finishes, the `Disc_Quote` constructor continues, which uses the in-class initializers to initialize `qty` and `discount`.
- when the `Disc_quote` constructor finishes, the `Bulk_quote` constructor continues but has no other work to do.

Similarly, the synthesized `Bulk_quote` copy constructor uses the (synthesized) `Disc_quote` copy constructor, which uses the (synthesized) `Quote` copy constructor. The `Quote` copy constructor copies the `bookNo` and

`price` members; and the `Disc_quote` copy constructor copies the `qty` and `discount` members.（也就是，每个类控制每个类的成员）

! It is worth noting that it doesn't matter whether the base-class member is itself synthesized or has a an user-provided definition. All that matters is that the corresponding member is accessible and that it is not a deleted function.

Each of our `Quote` classes use the synthesized destructor. The derived classes do so implicitly, whereas the `Quote` class does so explicitly by defininig its (virtual) destructor as `=defautl`. The synthesized destructor is (as usual) and its implicit destruction part destroys the members of the class. In addition to destroying its own members, the destruction phase of a destructor in a derived class also destroys its direct base. That destructor in turn invokes the destructor for its own direct base, if any. And, so on up to the root of the hierarchy.（每个类的析构函数负责释放自己对象的资源，继承的部分，由继承的基类的析构函数释放，依次类推。）

`Quote` does not have synthesized move operations because it defines a destructor. The (synthesized) copy operations will be used whenever we move a `Quote` object. The fact that `Quote` does not have move operation means that its derived classes don't either.

Base Classes and Deleted Copy Control in the Derived

The synthesized default copy-control members of a base or a derived class, may be defined as deleted for the same reasons as in any other class (The reasons have been given). In addtion, the way in which a base class is defined can cause a derived-class member to be defined as deleted（基类的定义方式会影响派生类成员是否被定义为`deleted`函数）:

- If the default constructor, copy constructor, copy-assignment operator, or destructor in the base class is **deleted or inaccessible**, then the corresponding member in the derived class is defined as deleted, because the compiler can't use the base-class member to construct, assign, or destroy the base-class part of the object.
- If the base class has an **inaccessible or deleted** destructor, then the synthesized default constructor and copy constructor（合成的默认构造函数和合成的拷贝构造函数）in the derived classes are defined as deleted, because there is no way to destroy the base part of the derived object. (of course, the destructor synthesized in the derived class also will be deleted)
- As usual, the compiler will not synthesize a deleted move operation. If we use `=default` to request a move operation, it will be a deleted function in hte derived if the corresponding operation in the base is deleted or inaccessible, becuase the base class destructor is deleted or inaccessible.

```
class B{
    public:
        B();
        B(const B &)=delete;
        // other members, not including a move constructor
};

class D:public B{
    // no constructors
};
D d;// ok, D's synthesized default constructor uses B's default constructor
D d2(d);// error, D's synthesized copy constructor is deleted, as the B's copy
```

```
    constructor is deleted
    D d3(std::move(d));// error, implicitly uses D's deleted copy constructor
```

基类B含有一个可访问的默认构造函数，和一个显示的删除的拷贝构造函数。因为，我们显示定义了拷贝构造函数，所以编译器不会给B合成一个移动构造函数。因此，我们既不能移动也不能拷贝B的对象。如果B的派生类希望它自己的对象可以被移动和拷贝，则派生类需要自定义相应版本的构造函数。当然，这一过程中派生类还必须考虑**如果移动或拷贝其基类部分的成员**。实际编程中，如果基类中没有默认、拷贝或移动构造函数，则一般情况下派生类也不会定义相应的操作。

Move Operation and Inheritance

Most base classes define a virtual destructor. As a result, by default, base classes generally do not get synthesized move operations. Moreover, by default, classes derived from a base class that doesn't have move operations don't get synthesized move operations either.

We can define the move operations for some base classes if it is sensible to do so. Our `Quote` class can use the synthesized versions. However, `Quote` must define these members explicitly. Once it defines its move operations, it must also explicitly define the copy version as well:

```
class Quote{
    public:
        Quote()=default;                // memberwise default initialize
        Quote(const Quote &)=default;   // memberwise copy
        Quote(Quote &&)=default;        // memberwise move
        Quote &operator=(const Quote &)=default;
        Quote &operator=(Quote &&)=default;
        virtual ~Quote()=default;
        // other members as before
};
```

Moreover, classes derived from `Quote` will automatically obtain synthesized move operations as well, unless they have members that otherwise preclude move.

Derived-Class Copy-Control Members

The initialization phase of a derived-class constructor initializes the base-class part of a derived object as well as initializing its own members. As a result, the copy and move constructors for a derived class must copy/move the members of its base part as well as the members in the derived. Similarly, a derived-class assignment operator must assign the members in the base part of the derived object.

Unlike the constructors and assignment operators, the destructor is reponsible only for destroying the resources allocated by the derived class. The base-class part of a derived is destroyed automatically (by the base destructor implicitly).

> The copy or move operation is responsible for copying or moving the entire object, including base-class members.

Defining a Derived Copy or Move Constructor

When we define a copy or move constructor for a derived class, we ordinarily use the corresponding base-class constructor to initialize the base part of the object:

```
class Base{/*...*/};
class D:public Base{
    public:
        // by default, the base class default constructor initializes the base
part of an object
        // to use the copy or move constructor, we must explicitly call that
        // constructor in the constructor initializer list
        D(const D &d):Base(d)/* initializers for members of D */{/*...*/}
        D(D&&d):Base(std::move(d))/*initializers for members of D */{/*...*/}
};
```

The initializer `Base(d)` pass a `D` object to a base-class constructor. 常理来说，`Base`可以包含一个参数类型为`D`的构造函数，但是实际编程中不会这么做。相反，`Base(d)`一般会匹配`Base`的拷贝构造函数。The object `d`, will be bound to the `Base &` parameter in that constructor. The `Base` copy constructor will copy the base part of `d` into the object that is being created. Had the initializer for the base class been omitted: (假设没有提供基类的initializer)

```
// probably incorrect definition of The D copy constructor
// base-class part is default initialized, not copied
D(const D &d)/* member initializers, but no base-class initializer */{/*...*/}
```

the `Base` default constructor would be used to initialize the base part of a `D` object. Assuming `D`'s constructor copies the derived members from `d`, this newly constructed object would be oddly configured: Its `Base` members would hold default values, while its `D` members would be copies of the data from another object.

Code to illustrate:

```
#include <iostream>
class Base {
protected:
    int base_mem=999;
public:
    Base() { std::cout << "Call the Base default constructor" << std::endl; };
    Base(const int& v) :base_mem(v) {
        std::cout << "Call the Base constructor with parameter"<<std::endl; }
    Base(const Base& b) :base_mem(b.base_mem) {
        std::cout << "Call the Base copy constructor" << std::endl;
    };
};
class D :public Base {
private:
    int d_mem = 888;
public:
    D() = default;
    D(const int &a,const int &b):Base(a),d_mem(b){}
```

```
    //D(const D& d) :Base(d),d_mem(d.d_mem) {}
    D(const D&d):d_mem(d.d_mem){}

    void foo()const {
        std::cout << "base_mem:" << base_mem << " ";
        std::cout << "d_mem:" << d_mem << std::endl;
    }
};
int main() {
    D d, d1(1,2);
    D d3(d1);
    d3.foo();
}
```

The output will be as:

```
Call the Base default constructor
Call the Base copy constructor with parameter
Call the Base default constructor
base_mem:999 d_mem:2
```

> By default, the base-class default constructor initializes the base-class part of a derived object. If we want copy (or move) the base-class part, we must explicitly use the copy (or move) constructor for the base class in the derived's constructor initializer list.

Derived-Class Assignment Operator

A derived-class assignment operator must assign its base part explicitly:

```
// Base::operator=(const Base &) is not invoked automatically
D &D::operator=(const D &rhs){
    Base::operator=(rhs); // assigns the base part

    // assign the members in the derived class, as usual,
    // handling self-assignment and freeing existing resources as appropriate
    return *this;
}
```

In a word, the derived-class assignment operator first invokes explicitly the base-class assignment operator to assign the members of the base-class part of the derived object. Once it finishes, we continue doing whatever is needed to assign the members in the derived class.

Derived-Class Destructor

The data members of an object are destroyed after the destructor body completes. Similarly, the base-class part of an object are also implicitly (automatically) destroyed. Therefore, a derived destructor is responsible only for destroying the resources allocated by the derived class（这一点和构造函数，拷贝构造函数，赋值操作等不同，它们需要显示的处理基类部分的成员）:

```
class D:public Base{
    public:
        // Base::~Base invoked automatically
        ~D(){/* do what it takes to clean up derived-only members */}
};
```

Objects are destroyed in a opposite order from which they are constructed: The derived destructor is run first, and the base-class destructor are invoked (implicitly and automatically), back up through the inheritance hierarchy.

! Calls to Virtuals in Constructors and Destructors

对于派生类对象而言，构造过程：

1. 先（显示）调用基类构造函数初始化该对象的base-class部分
2. 然后利用initializer list初始化该对象的derived-class部分
   析构过程：
3. 先销毁该对象的derived-class部分
4. 销毁该对象的base-class部分

While the base-class constructor is executing, the derived part of the object is unitialized. Similarly, derived objects are destroyed in reverse order, so that when a base class destructor runs, the derived part has already been destroyed.As a result, while these base-class members are excuting, the object is incomplete.

To accommodata this incompleteness, the compiler treats the object ***as if*** its type changes during construction or derstruction. That is, while an object is being constructed it is treated as if it has the same class as the constructor; calls to virtual functions will be bound as if the object has the same type as the constructor itself. Similarly, for destructors. This binding applies to virtuals called directly or that are called indirectly from a function that the constructor (or destructor) calls.

To understand this behaviour, consider that what would happen if the derived-class version of a virtual was called from a base-class constructor. This virtual probably accesses members of the derived object. After all, if the derived class probably could use the version in its base class. However, those members are uninitialized while a base constructor is running. If such access were allowed, the program would probable crash.

> 如果构造函数或析构函数调用虚函数，那么该虚函数将调用和构造函数或析构函数相对应类型的版本。

Inherited Constructors

C++11新标准，允许派生类重用其direct基类定义的构造函数。这些构造函数并非以常规的方式继承而来，但是为了方便，我们不妨姑且称之为“继承”的。类不可以“继承”默认、拷贝和移动构造函数。如果类没有直接定义，那么编译器将会合成它们。

派生类“继承”基类构造函数的方式，是通过`using`声明。

```
class Bulk_quote:public Disc_quote{
    public:
        using Disc_quote::Disc_quote;// “继承”基类的构造函数
```

```
        // other members
};
```

Ordinarily, a `using` declaration only makes a name visible in the current scope. When applied to a constructor, a `using` declaration causes the compiler to generate code. The compiler generates a derived constructor corresponding to each constructor in the base. That is, for each constructor in the base class, the compiler generates a constructor in the derived class that has the same parameter list.

These compiler-generated constructors have the form
`derived(parms):base(args){}`
where `derived` is the name of the derived class, `base` is the name of the base class, `parms` is the parameter list of the constructor, and `args` pass the parameters from the derived constructor to the base constructor. The inherited constructor would be equivalent to

```
Bulk_quote(const string &book,double price,std::size_t qty,double
disc):Disc_quote(book,price,qty,disc){}
```

If the derived class has any data members of its own, those members are default initialized.

Characteristics of an Inherited Constructor

❗ Unlike `using` declaration for ordinary members, a constructor `using` declaration does not change the access level of the inherited constructor(s). For example, regardless of where the `using` declaration appears, a `private` constructor in the base is a `private` constructor in the derived; similarly for `protected` and `public` constructors.

❗ If a base-class constructor has default arguments, those arguments are not inherited. Instead, the derived class gets multiple inherited constructors in which each parameter with a default arguments is successively omitted. For example, if the base has a constructor with two parameters, the second of which has a default, the derived class will obtain two constructors: one with both parameters (and no default argument) and a second constructor with a single parameter corresponding to the left-most, non-defaulted parameter in the base class.

If a base class has several constructors, then with two exceptions, the derived class inherits each of the constructors from its base class. The first exception is that a derived class can inherit some constructor and define its own version of other construtors. If the derived class defines a constructor with the same parameters as a constructor in the base, then that constructor is not inherited. The one defined in the derived class is used in place of the inherited constructor.（这里和使用基类的重载函数一样，既可以自定义自己的个别版本，也可以使用基类的其他版本，参数相同的会覆盖掉基类的，也是通过使用`using`声明）

The second exception is that the default, copy, and move constructor are not inherited. These constructors are synthesized using the normal rules. An inherited constructor is not treated as a user-defined constructor. Therefore, a class that contains only inherited constructors will have a synthesized default constructor. 以下是默认构造函数、拷贝构造函数。。。等等可以由编译器合成的都不可以被继承。继承的构造函数不会被视为user-defined的构造函数。因此，如果一个类含有继承的构造函数，那么它也将拥有一个合成的默认构造函数。

```
class Foo{
    public:
        Foo()=default;// default constructor synthesized by compiler
        Foo(const Foo &)=default;// copy constructor synthesized by compiler
        Foo &operator=(const Foo &)=default;// assignment operator ...
        Foo(Foo &&)=default;// move copy constructor ...
        Foo &operator=(Foo &&)=default;// move assignment operator...

        Foo(int){/*..*/}// user-defined constructor1
        Foo(int,int){/*...*/}// user-defined constructor2
        // other constructors with distinct parameters
}
```

针对以上内容，给出以下代码说明一些问题：

```
#include <iostream>
class Base {
private:
    int base_mem=0; // initialized 0 by default
public:
    Base() = default;
    Base(int v):base_mem(v){}// user-defined constructor
    Base(int a, const int &b=111):base_mem(b) {
        // the second paremeter is 111 by default
        std::cout << "base(int,int)" << std::endl;
    }
    int get()const { return base_mem; }
};
class D :public Base {
private:
    int d_mem=999;
public:
    using Base::Base;

    void foo()const {
        std::cout <<"base_mem:"<<get()<<std::endl;
        std::cout<<"d_mem:"<<d_mem<<std::endl;
    }
};
int main() {
    D d(1);
    d.foo();
}
```

此时，对于`D d(1)`会报错，因为对于“继承”的构造函数：

- 没有默认参数，如`Base(int v):base_mem(v){}`，会直接被“继承”过来，由编译器在派生类中生成相应的代码，等价于

```
D(int v):Base(v){}
// 其中派生类的成员被默认初始化
```

- 含有默认参数，如`Base(int a, const int &b=111):base_mem(b)`，编译器会生成两个构造函数的代码，等价于

```
D(int a,const int &b):Base(a,b){}// 其中派生类成员被默认初始化
D(int a):Base(a){}//...
```

可以发现，出现了调用混淆冲突。