继承

面向对象的三大特性：

1. 封装 ---- 基于类
2. 继承 ---- 基于封装
3. 多态 ---- 基于继承

继承是类与类之间的关系，是一个很简单很直观的概念，与现实世界中的继承（例如儿子继承父亲财产）类似。

继承的作用是什么

> **代码重用**

继承方式

从基类派生出派生类，派生类继承基类的继承方式有三种：public、protected和private。在未指定的情况下编译器会默认继承方式为protected或private方式。

1. public继承方式

- 基类中所有public成员在派生类中为public属性；
- 基类中所有protected成员在派生类中为protected属性；
- 基类中所有private成员在派生类中不可访问。

2. protected继承方式

- 基类中的所有public成员在派生类中为protected属性；
- 基类中的所有protected成员在派生类中为protected属性；
- 基类中的所有private成员在派生类中仍然不可访问。

3. private继承方式

- 基类中的所有public成员在派生类中为private属性；
- 基类中的所有protected成员在派生类中为private属性；
- 基类中的所有private成员在派生类中仍然不可访问。

什么是继承

> 保持已有类的特性而构造新类的过程称为继承。

什么是派生

在已有类的基础上新增自己的特性而产生新类的过程称为派生。

被继承的已有类称为基类（或父类）。

派生出的新类称为派生类（或子类）。

继承可以理解为一个类从另一个类获取方法（函数）和属性（成员变量）的过程。如果类B继承于类A，那么B就拥有A的方法和属性。被继承的类称为父类或基类，继承的类称为子类或派生类。

例如：

```
// 基类/父类
class Base{
public:
    int getValue(){
        return x;
    }
private:
    int x;
};

// 子类/派生类(没有特殊要求,这里是公有继承)
class Derived : public Base{
};

int main(){
    Base b;
    cout << b.getValue() << endl;

    Derived d;
    cout << d.getValue() << endl;

    return 0;
}
```

例子

```
enum language{cpp, java, python, javascript, php, ruby};

class book{
public:
    void setprice(double a);
    double getprice() const;
    void settitle(char* a);
    char *gettitle() const;
    void display();
private:
    double price;
    char *title;
};
```

```
class codingbook : public book{
public :
    void setlang(language lang);
    language getlang(){return lang;}

private:
    language lang;
};
```

为了方便起见，我们先在类定义前声明了一个全局的枚举类型language，用于表示编程语言。book类我们已经很熟悉了，关键是codingbook类的定义。在定义codingbook类时多出了“: public book”，除此之外codingbook类的定义和上一章介绍的类定义方法没有什么差别。其中关键字public指明继承方式属于公有继承，book为被继承的类名。采用公用继承方式，则基类的公有成员变量和成员函数的属性继承到派生类后不发生变化。例如book类的公有的setprice和settitle成员函数继承到codingbook类后，这两个成员变量的属性仍将是public属性。如果在继承过程中不指名继承方式时，编译器系统会默认继承方式为private或protected属性。

继承方式	基类成员特性	派生类成员特性	派生类对象访问
公有继承	public	public	可直接访问
	protected	protected	不可直接访问
	private	不可直接访问	不可直接访问
私有继承	public	private	不可直接访问
	protected	private	不可直接访问
	private	不可直接访问	不可直接访问
保护继承	public	protected	不可直接访问
	protected	protected	不可直接访问
	private	不可直接访问	不可直接访问

在本例中因为已经定义过一个book类，具有book类的基本属性：书名和书的价格。现在需要一个新的类codingbook来描述编程类书籍，为此我们继承book类中的所有成员及成员函数，并新增language属性及相应的操作函数。虽然我们可以继承到book类的私有成员，但是book类的私有成员变量在派生类中我们是无法直接访问的，只能通过间接的方式访问。间接访问则是通过getprice、gettitle、setprice和settitle函数来实现的，因为这些函数在派生类中是public属性的。如下表所示为codingbook类中的所有成员的一览。

扩展

final关键字

1.禁用继承

C++11中允许将类标记为final，使用时直接在类名称后面使用关键字final，如此，意味着继承该类会导致编译错误。

实例如下：

```
class Super final
{
  //......
};
```

2.禁用重写

C++中还允许将方法标记为final，这意味着无法再子类中重写该方法。这时final关键字至于方法参数列表后面，如下

```
class Super{
  public:
    Supe();
    virtual void SomeMethod() final;
};
```

3.final函数和类

C++11的关键字final有两个用途。第一，它阻止了从类继承；

第二，阻止一个虚函数的重载。

我们先来看看final类:

程序员常常在没有意识到风险的情况下坚持从std::vector派生。在C++11中，无子类类型将被声明为如下所示：

```
class TaskManager {/*..*/} final;
class PrioritizedTaskManager: public TaskManager {
};  //compilation error: base class TaskManager is final

class A final{
};

class B : public A{
};
// complier error class A is final
```

同样，你可以通过声明它为final来禁止一个虚函数被进一步重载。如果一个派生类试图重载一个final函数，编译器就会报错：

```
class A{
pulic:
  virtual void func() const;
};

class  B : public A{
pulic:
    void func() const override final;     // OK
};

class C: public B{
pulic:
    void func() const;     // error, B::func is final
};
```

- C::func()是否声明为override没关系，一旦一个虚函数被声明为final，派生类不能再重载它。

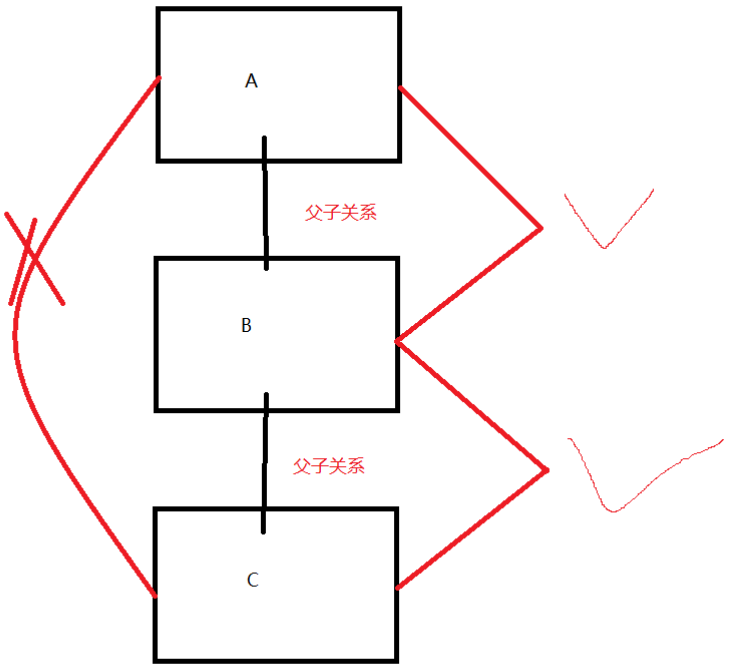
改变基类成员在派生类中的访问属性

使用using声明可以改变基类成员在派生类中的访问属性。我们知道基类的公有成员经过公有继承，在派生类中其属性为public的，但是通过using声明，我们可以将其改为private或protected属性。

```
enum language{
    cpp,java,python,javascript,php,ruby
};

class book{
public:
    void setprice(double a);
private:
    double price;
};

class codingbook : public book{
private:
    language lang;
    using book::setprice;       // 在这里修改了基类成员在派生类中的访问属性
};

int main(){
    codingbook code;
    code.setprice(11.2);        // 在这里访问就会报错,因为setprice函数已经被修改了访问权限
    return 0;
}
```

间接继承

假设类C继承自类B，类B继承自类A。那么类C中的除了能够继承B类的成员函数和成员变量外，同样也能继承B类继承自A类的所有成员。换言之，类C可以继承来自类A的所有成员。因此继承既可以是直接继承，也可以是间接继承。



继承关系的构造函数和析构函数

```
#include <iostream>

using namespace std;

class Base{
public:
    Base(){
        cout << "This is Base's constructor" << endl;
    }
    ~Base(){
        cout << "This is Base's destructor" << endl;
    }
};

class Derived : public Base{
public:
    Derived(){
        cout << "This is Derived's constructor" << endl;
    }
    ~Derived(){
        cout << "This is Derived's constructor" << endl;
    }
};

int main(){
```

```
    Base b;
    // 1. "This is Base's constructor"
    // 2. "This is Base's destructor"

    Derived d;
    // 1. "This is Base's constructor"
    // 2. "This is Derived's constructor"
    // 3. "This is Derived's destructor"
    // 4. "This is Base's destructor"

    return 0;
}
```

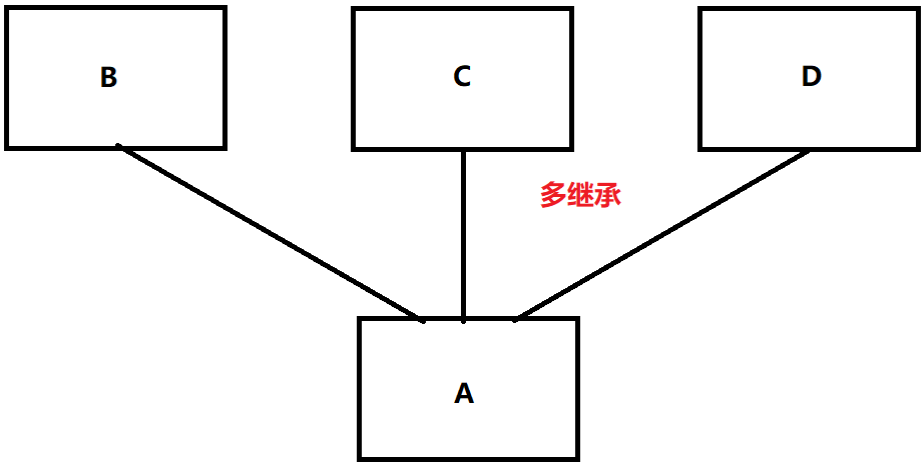
如上例所示：(构造和析构函数调用规则可以联想到堆栈的进出规则)

当子类创建对象时，先调用父类的构造函数，再调用子类的构造函数

当子类析构对象时，先调用子类的析构函数，再调用父类的析构函数

多继承

在前面所有的例子中，派生类都只有一个基类，我们称这种情况为单继承。而在C++中一个派生类中允许有两个及以上的基类，我们称这种情况为多继承。单继承中派生类是对基类的特例化，例如前面中编程类书籍是书籍中的特例。而多继承中，派生类是所有基类的一种组合。

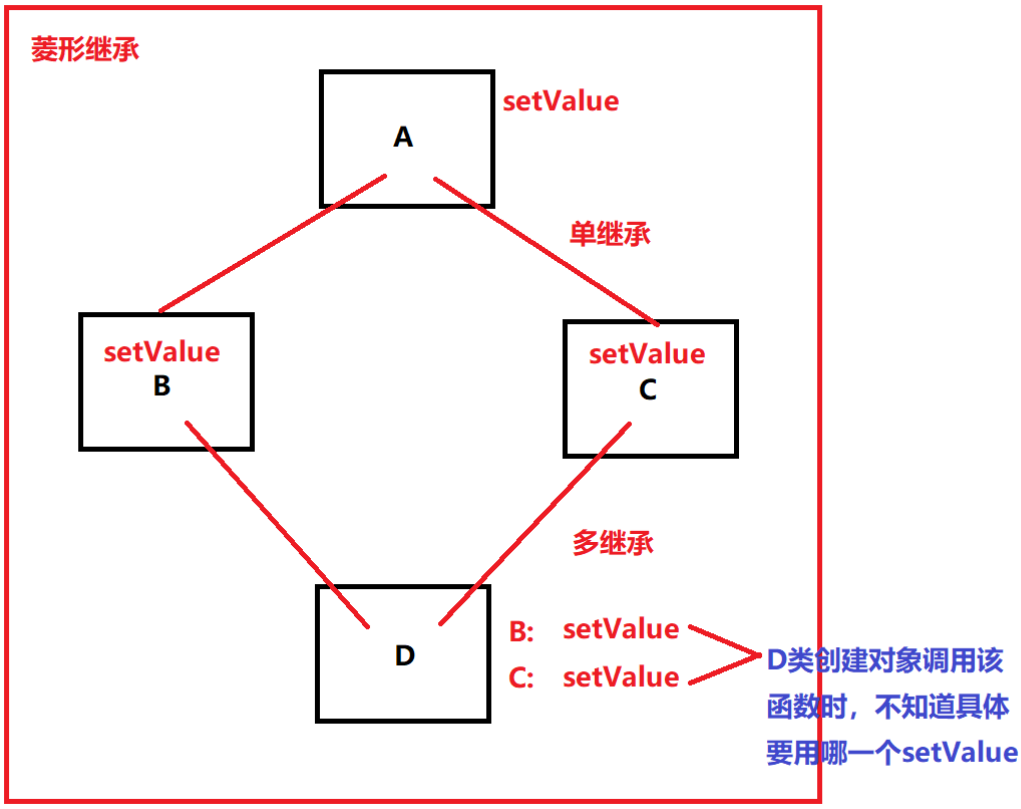



```
class teacher{
public:
    void settitle(char *a){ title = a; }
    char *gettitle() { return title; }
private:
    char *title;   // 职称
};

class cadre{
public:
    void  setpost(char *a){ post = a; }
    char  *getpost() { return  post; }
private:
```

```
    char  *post;   // 职务
};

class teacher_cadre : public cadre,  public teacher{
public:
    void setwages(int a){ wages = a; }
    int  getwages(){ return wages; }
private:
    int  wages;   // 工资
};
```

多继承中的特例(菱形继承)

在多继承时很容易产生命名冲突问题，如果我们很小心地将所有类中的成员变量及成员函数都命名为不同的名字时，命名冲突依然有可能发生，比如非常经典的菱形继承层次。类A派生出类B和类C，类D继承自类B和类C，这个时候类A中的成员变量和成员函数继承到类D中变成了两份，一份来自A派生B然后派生D这一路，另一份来自A派生C然后派生D这一条路。




```
#include <iostream>

using namespace std;

class A{
public:
    void setValue(){
        cout << "This is A's setValue" << endl;
    }
};
```

```
class B : public A{
public:
};

class C : public A{
public:
};

class D : public B, public C{
public:
};

int main(){
    A a;
    a.setValue();       // √
    B b;
    b.setValue();       // √
    C c;
    c.setValue();       // √
    D d;
    d.setValue();       // ×

    return 0;
}
```

```
D d;
d.setValue();    ◦ non-static member 'setValue' found in multiple base-class subobjects of type 'A':
```

菱形继承中遇到的错误该如何解决?

本例即为典型的菱形继承结构，类A中的成员变量及成员函数继承到类D中均会产生两份，这样的命名冲突非常的棘手，通过域解析操作符已经无法分清具体的变量了。为此，C++提供了虚继承这一方式解决命名冲突问题。虚继承只需要在继承属性前加上virtual关键字。

上述代码中，只需要将class B和class C虚继承即可，实现方法：

```
class B : virtual public A{
public:
};

class C : virtual public A{
public:
};
```

如果要调用B中的setValue或者C中的setValue该如何做?

```
int main(){
    D d;
    d.setValue();
    d.B::setValue();    // 调用B类中的setValue
    d.C::setValue();    // 调用C类中的setValue

    return 0;
}
```

在本例中，类B和类C都是继承类A都是虚继承，如此操作之后，类D只会得到一份来自类A的数据。在本例的主函数中，定义了类D的对象test，然后通过该对象调用从类A间接继承来的setx和getx成员函数，因为B和C继承自类A采用的是虚继承，故通过D调用setx和getx不会有命名冲突问题，因为D类只得到了一份A的数据。

问题：后续讲virtual关键字的时候解释

使用虚继承后，B和C发生了什么变化，使D避免了命名冲突问题？方便解释下内在原因吗？(王志强)

多态

多态是基于继承和封装的

早绑定/编译时绑定(静态多态)

晚绑定/运行时绑定(动态多态)

在前面的所有列举的程序中，函数的入口地址与函数名是在编译时进行绑定的，我们称之为编译期绑定，而多态的功能则是将函数名动态绑定到函数入口地址，这样的动态绑定过程称为运行期绑定，换句话来说就是函数名与函数入口地址在程序编译时无法绑定到一起，只有等运行的时候才确定函数名与哪一个函数入口绑定到一起。

那么多态到底有什么用处呢？我们不妨来看个例子。在windows操作系统中，我们经常会进行一些关闭操作，比如关闭文件夹、关闭文本文件、关闭播放器窗口等，这些关闭动作对应的函数close假设都继承自同一个基类，但是每一个类都需要有自己的一些特殊功能，比如清理背景、清除缓存等工作，如此一来当我们执行close函数时，我们当然希望根据当前所操作的窗口类型来决定该执行哪一个close函数，因此运行期绑定就可以派上用场了。

编译期绑定是指在程序编译时就将函数名与函数入口地址绑定到一起，运行期绑定是指在程序运行时才将函数名与函数入口地址绑定到一起，而在运行期绑定的函数我们称其是多态的。

为了说明虚函数的必要性，我们先来看一个示例程序。

```
class Base{
public:
    void func(){
        cout << "This is Base's func" << endl;
    }
};

class Derived : public Base{
public:
    void func(){
        cout << "This is Derived's func" << endl;
    }
```

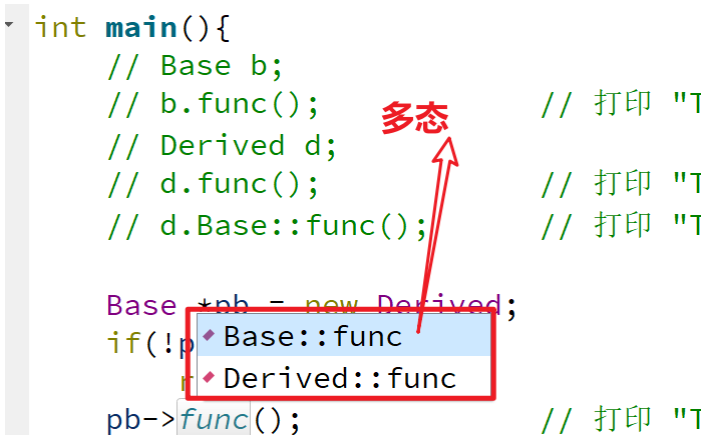
```
};

int main(){
    // Base b;
    // b.func();            // 打印 "This is Base's func"
    // Derived d;
    // d.func();            // 打印 "This is Derived's func"
    // d.Base::func();      // 打印 "This is Base's func"

    Base *pb = new Derived;
    if(!pb)
        return 0;
    pb->func();             //
    delete pb;
    pb = nullptr;

    return 0;
}
```

多态要满足的条件

1. 具有继承关系的两个类
2. 父类和子类中要有同名函数(不是重载，函数名，函数签名都要相同，甚至连返回值也必须相同)
3. 父类中的同名函数要加virtual关键字修饰，表示是一个虚函数，子类中的同名函数加不加virtual关键字都可以
4. 父类指针指向子类对象(形成多态)

virtual：

virtual修饰函数的时候，只能修饰一个非静态成员函数



举例：

计算形状的面积

```
#include <iostream>
#define PI 3.14
using namespace std;

class Shape{
```

```
public:
    // 在这里calcArea函数只是提供一个接口之用,具体函数实现的是什么无关紧要，后面还有更好的替代方案(纯虚函数)
    virtual double calcArea(){
        return 1.2;     // 返回值随便写
    }
};

// 计算圆的面积
class Radius : public Shape{
public:
    Radius(float r){
        m_fR = r;
    }
    // 在calcArea函数后最好加一个override,这样很明确知道要重写父类中的同名函数，virtual加不加无所谓
    virtual double calcArea() override{
        return PI * m_fR * m_fR;
    }
private:
    float m_fR;     // 圆的半径
};

// 计算矩形面积
class Rectangle : public Shape{
public:
    Rectangle(float width, float height){
        m_fWidth = width;
        m_fHeight = height;
    }
    // 在calcArea函数后最好加一个override,这样很明确知道要重写父类中的同名函数，virtual加不加无所谓
    virtual double calcArea() override{
        return m_fWidth * m_fHeight;
    }
private:
    float m_fWidth;     // 矩形宽度
    float m_fHeight;    // 矩形高度
};

int main(){
    // 计算并打印圆的面积
    Shape *pShape1 = new Radius(2.5);
    if(!pShape1)
        return 0;
    cout << "Radius's Area = " << pShape1->calcArea() << endl;
    delete pShape1;
    pShape1 = nullptr;

    // 计算并打印矩形面积
    Shape *pShape2 = new Rectangle(2.5, 4.8);
    if(!pShape2)
        return 0;
    cout << "Rectangle's Area = " << pShape2->calcArea() << endl;
    delete pShape2;
    pShape2 = nullptr;

    return 0;
```

```
}
```

重载/重写/覆盖/遮蔽/隐藏　区别和联系