面向对象之继承

在学习类和对象时，我们知道对象是基本，我们从对象上抽象出类。但是，世界可并不是一层对象一层类那么简单，对象抽象出类，在类的基础上可以再进行抽象，抽象出更高层次的类。所以经过抽象的对象论世界，形成了一个树状结构。(动物分类图　金融类　家用电器分类图　家族图谱 --- 泛化)

对象论的世界观认为，世界的基本元素是对象，我们将抽象思维作用于对象，形成了类的概念，而抽象的层次性形成了**抽象层次树**的概念。而抽象层次树为我们衍生出继承的概念，提供了依据。

我们需要继承这个概念，本质上是因为对象论中世界的运作往往是在某一抽象层次上进行的，而不是在最低的基本对象层次上。举个例子，某人发烧了，对其他人说：“我生病了，要去医院看医生。”这句简短的话中有一个代词“我”和三个名词“**病**”、“**医院**”、“**医生**”。这四个具有名词性的词语中，除了“我”是运作在世界的最底层——基本对象层外，其他三个都运作在抽象层次，在这个语境中，“**病**”、“**医院**”、“**医生**”都是抽象的。但是，本质上他确实是生了一个具体的病，要去一个具体的医院看一个具体的医生，那么在哲学上要如何映射这种抽象和具体呢？就是靠继承， 拿医生来说吧，所有继承自“医生”类的类所指的所有具体对象都可以替换掉这里具体的医生，这都不影响这句话语义的正确性。

回到编程语言层面，在C语言中重用代码的方式就是拷贝代码、修改代码。C++中代码重用的方式之一就是采用继承。继承是面向对象程序设计中重要的特征，可以说，不掌握继承就等于没有掌握面向对象的精华。**通过继承，我们可以用原有类型来定义一个新类型，定义的新类型既包含了原有类型的成员，也能自己添加新的成员，而不用将原有类的内容重新书写一遍。**原有类型称为“基类”或“父类”，在它的基础上建立的类称为“派生类”或“子类”。

1、继承的定义

当一个派生类继承一个基类时，需要在派生类的**类派生列表**中明确的指出它是从哪个基类继承而来的。类派生列表的形式是在类名之后，大括号之前用冒号分隔，后面紧跟以逗号分隔的基类列表，其中每个基类前面可以有访问修饰限定符，其形式如下:

```
class 派生类
: public/protected/private 基类
{

};
```

派生类的生成过程包含3个步骤:

1. 吸收基类的成员
2. 改造基类的成员
3. 添加自己新的成员

```
class Point3D
: public Point
{
public:
    Point3D(int x, int y, int z)
    : Point(x, y)
    , _z(z)
    {
        cout << "Point3D(int,int,int)" << endl;
    }
```

```
11
12      void display() const
13      {
14          print();
15          cout << _z << endl;
16      }
17  private:
18      int _z;
19  };
```

2、继承的局限

不论何种继承方式，下面这些基类的特征是不能从基类继承下来的：

- 构造函数
- 析构函数
- 用户重载的operator new/delete运算符
- 用户重载的operator=运算符
- 友元关系

3、派生方式对基类成员的访问权限

派生类继承了基类的全部成员变量和成员方法（除了构造和析构之外的成员方法），但是这些成员的访问属性，在派生过程中是可以调整的。

派生（继承）方式有3种，分别是

- public（公有）继承
- protected（保护型）继承
- private（私有）继承

继承方式	基类成员访问权限	在派生类中访问权限	派生类对象访问
公有继承	public protected private	public protected 不可直接访问	可直接访问 不可直接访问 不可直接访问
保护继承	public protected private	protected protected 不可直接访问	不可直接访问
私有继承	public protected private	private private 不可直接访问	不可直接访问

通过继承，除了基类私有成员以外的其它所有数据成员和成员函数，派生类中可以直接访问。
private成员是私有成员，只能被本类的成员函数所访问，派生类和类外都不能访问。
public成员是公有成员，在本类、派生类和外部都可访问。
protected成员是保护成员，只能在本类和派生类中访问，是一种区分血缘关系内外有别的成员。

总结：派生类的访问权限规则如下：
1．不管以什么继承方式，派生类内部都不能访问基类的私有成员。
2．不管以什么继承方式，派生类内部除了基类的私有成员不可以访问外，其他的都可以访问。
3．不管以什么继承方式，**派生类对象**除了公有继承基类中的公有成员可以访问外，其他的一律不能访问。

4、派生类对象的构造

我们知道，构造函数和析构函数是不能继承的，为了对数据成员进行初始化，**派生类必须重新定义构造函数和析构函数。**由于派生类对象通过继承而包含了基类数据成员，因此，创建派生类对象时，系统首先通过派生类的构造函数来调用基类的构造函数，完成基类成员的初始化，而后对派生类中新增的成员进行初始化。

派生类构造函数的一般格式为：

```
派生类名(总参数表)：基类构造函数(参数表)
{
    //函数体
};
```

对于派生类对象的构造，我们分下面4种情况进行讨论：

1. 如果派生类有显式定义构造函数，而基类没有显示定义构造函数，则创建派生类的对象时，派生类相应的构造函数会被自动调用，此时都自动调用了基类缺省的无参构造函数。

```
class Base
{
pulbic:
    Base()
    {
        cout << "Base()" << endl;
    }
};

class Derived
: public Base
{
public:
    Derived(long derived)
    : _derived(derived)
    {
        cout << "Derived(long)" << endl;
    }

    long _derived;
};

void test()
{
    Derived d(1);
}
```

2. 如果派生类没有显式定义构造函数而基类有显示定义构造函数，则基类必须拥有默认构造函数。

```
class Base
{
pulbic:
    Base(long base)
    {
        cout << "Base(long)" << endl;
    }
private:
    long _base;
};

class Derived
: public Base
{
public:
    //没有显示定义那就是编译器合成
};

void test()
{
    Derived d;//error
}
```

3. 如果派生类有构造函数，基类有默认构造函数，则创建派生类的对象时，基类的默认构造函数会自动调用，如果你想调用基类的有参构造函数，必须要在派生类构造函数的初始化列表中显示调用基类的有参构造函数。
4. 如果派生类和基类都有构造函数，但基类没有默认的无参构造函数，即基类的构造函数均带有参数，则派生类的每一个构造函数必须在其初始化列表中显示的去调用基类的某个带参的构造函数。如果派生类的初始化列表中没有显示调用则会出错，因为基类中没有默认的构造函数。

```
class Base
{
pulbic:
    Base(long base)
    {
        cout << "Base(long)" << endl;
    }
private:
    long _base;
};

class Derived
: public Base
{
public:
    Derived(long base, long derived)
    : Base(base)
    , _derived(derived)
    {
        cout << "Derived(long, long)" << endl;
    }

    long _derived;
};

void test()
```

```
{
    Derived d(1, 2);
}
```

虽然上面细分了四种情况进行讨论，但不管如何，谨记一条: 必须将基类构造函数放在派生类构造函数的初试化列表中，以调用基类构造函数完成基类数据成员的初始化。派生类构造函数实现的功能，或者说调用顺序为：

1. 完成对象所占整块内存的开辟，由系统在调用构造函数时自动完成。
2. 调用基类的构造函数完成基类成员的初始化。
3. 若派生类中含对象成员、const成员或引用成员，则必须在初始化表中完成其初始化。
4. 派生类构造函数体执行。

5、派生类对象的销毁

当派生类对象被删除时，派生类的析构函数被执行。析构函数同样不能继承，因此，在执行派生类析构函数时，**基类析构函数会被自动调用**。执行顺序是先执行派生类的析构函数，再执行基类的析构函数，这和执行构造函数时的顺序正好相反。当考虑对象成员时，继承机制下析构函数的调用顺序：

1. 先调用派生类的析构函数
2. 再调用派生类中成员对象的析构函数
3. 最后调用普通基类的析构函数

6、多基继承（多基派生）

C++除了支持单根继承外，还支持多重继承。那为什么要引入多重继承呢？其实是因为在客观现实世界中，我们经常碰到一个人身兼数职的情况，如在学校里，一个同学可能既是一个班的班长，又是学生会中某个部门的部长；在创业公司中，某人既是软件研发部的CTO，又是财务部的CFO；一个人既是程序员，又是段子手。诸如此类的情况出现时，单一继承解决不了问题，就可以采用多基继承了。

多重继承的定义形式如下:

```
class 派生类
: public/protected/private 基类1
, ...
, public/protected/private 基类N
{

};
```

下面我们举一个例子：

```
class Fairy
{
public:
    void fly()
    {
        cout << " can fly.\n";
    }
};

class Monstor
```

```
{
public:
    void attack()
    {
        cout << " take an attack.\n";
    }
};

class Monkey
: public Fairy
, public Monstor
{
public:
    Monkey(const string & name)
    : _name(name)
    {
    }

    void print() const
    {
        cout << _name << " ";
    }
private:
    string _name;
};

void test()
{
    Monkey sunWukong("Sun Wukong");
    sunWukong.print();
    sunWukong.fly();
    sunWukong.attack();
}
```

6.1、多基继承的派生类对象的构造和销毁

多基派生时，派生类的构造函数格式如（假设有N个基类）：

```
派生类名(总参数表)
: 基类名1(参数表1)
, 基类名2(参数表2)
, ...
, 基类名N(参数表N)
{
    //函数体
}
```

和前面所讲的单基派生类似，总参数表中包含了后面各个基类构造函数需要的参数。

多基继承和单基继承的派生类构造函数完成的任务和执行顺序并没有本质不同，唯一一点区别在于：首先要执行所有基类的构造函数，再执行派生类构造函数中初始化表达式的其他内容和构造函数体。各基类构造函数的执行顺序与其在初始化表中的顺序无关，而是由定义派生类时指定的基类顺序决定的。

析构函数的执行顺序同样是与构造函数的执行顺序相反。但在使用多基继承过程中，会产生两种二义性。

6.2、成员名冲突的二义性

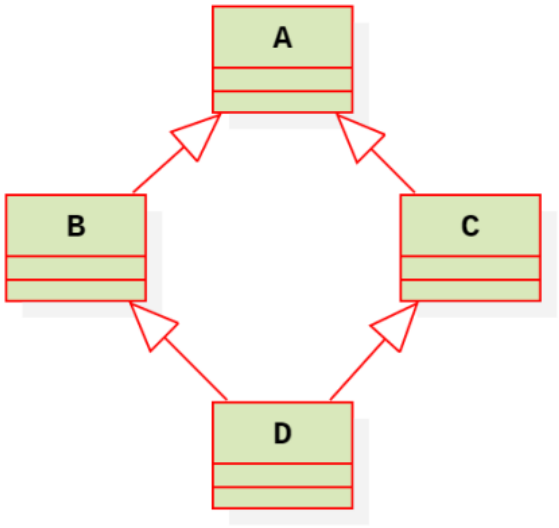
一般来说，在派生类中对基类成员的访问应当具有唯一性，但在多基继承时，如果多个基类中存在同名成员的情况，造成编译器无从判断具体要访问的哪个基类中的成员，则称为对基类成员访问的二义性问题。如下面的例子，我们先定义3个不同的类A、B、C，这3个类中都有一个同名成员函数print，然后让类D继承自A、B、C，则当创建D的对象d，用d调用成员函数print时，发生编译错误。

```
class A
{
public:
    void print()
    {
        cout << "A::print()" << endl;
    }
};

class B
{
public:
    void print()
    {
        cout << "B::print()" << endl;
    }
};

class C
{
public:
    void print()
    {
        cout << "C::print()" << endl;
    }
};

class D
: public A
, public B
, public C
{

};

void test()
{
    D d;
    d.print();//error
    d.A::print();//ok
    d.B::print();//ok
    d.C::print();//ok
}
```

解决该问题的方式比较简单，只需要在调用时，指明要调用的是某个基类的成员函数即可，即使用作用域限定符就可以解决该问题。

6.3、菱形继承的二义性问题

而另外一个就是菱形继承的问题了。多基派生中，如果在多条继承路径上有一个共同的基类，如下图所示，不难看出，在D类对象中，会有来自两条不同路径的共同基类（类A）的双重拷贝。



出现这种问题时，我们的解决方案是采用虚拟继承。中间的类B、C虚拟继承自A，就可以解决了。至于背后到底发生了什么，待我们学了多态的知识后一起做讲解。

```
class A
{
public:
    void setNumber(long number)
    {
        _number = number;
    }

private:
    long _number;
};

class B
: virtual public A
{

};

class C
: virtual public A
{

};

class D
: public B
, public C
{

};

```

```
int main(void)
{
    D d;
    d.setNumber(10);

    return 0;
}
```

7、基类与派生类间的相互转换

“类型适应”是指两种类型之间的关系，说A类适应B类是指A类的对象能直接用于B类对象所能应用的场合，从这种意义上讲，**派生类适应于基类**，派生类的对象适应于基类对象，派生类对象的指针和引用也适应于基类对象的指针和引用。

- 可以把派生类的对象赋值给基类的对象
- 可以把基类的引用绑定到派生类的对象
- 可以声明基类的指针指向派生类的对象 (向上转型)

也就是说如果函数的形参是基类对象或者基类对象的引用或者基类对象的指针类型，在进行函数调用时，相应的实参可以是派生类对象。

```
class Base
{
public:
    Base(long base)
    {
        cout << "Base(long)" << endl;
    }
private:
    long _base;
};

class Derived
: public Base
{
public:
    Derived(long base, long derived)
    : Base(base)
    , _derived(derived)
    {
        cout << "Derived(long,long)" << endl;
    }
private:
    long _derived;
};

void test()
{
    Base base(1);
    Derived derived(10, 11);

    base = derived;//ok
    Base &refBase = derived;//ok
    Base *pBase = &derived;//ok

```

```
    derived = base;//error
    Derived &refDerived = base;//error
    Derived *pDerived = &base;//error

    cout << endl << endl;
    Base base2(10);
    Derived derived2(20, 30);

    Derived *pderived2 = static_cast<Derived *>(&base2);//不安全的向下转型
    pderived2->print();

    cout << endl;
    Base *pbase3 = &derived2;
    Derived *pderived3 = static_cast<Derived *>(pbase3);//安全的向下转型
    pderived3->print();
}
```

8、派生类对象间的复制控制

从前面的知识，我们知道，基类的**拷贝构造函数**和**operator=运算符函数**不能被派生类继承，那么在执行派生类对象间的复制操作时，就需要注意以下几种情况:

1. 如果用户定义了基类的拷贝构造函数，而没有定义派生类的拷贝构造函数，那么在用一个派生类对象初始化新的派生类对象时，两对象间的派生类部分执行缺省的行为，而两对象间的基类部分执行用户定义的基类拷贝构造函数。
2. 如果用户重载了基类的赋值运算符函数，而没有重载派生类的赋值运算符函数，那么在用一个派生类对象给另一个已经存在的派生类对象赋值时，两对象间的派生类部分执行缺省的赋值行为，而两对象间的基类部分执行用户定义的重载赋值函数。
3. 如果用户定义了派生类的拷贝构造函数或者重载了派生类的对象赋值运算符=，则在用已有派生类对象初始化新的派生类对象时，或者在派生类对象间赋值时，将会执行用户定义的派生类的拷贝构造函数或者重载赋值函数，而不会再自动调用基类的拷贝构造函数和基类的重载对象赋值运算符，这时，通常需要用户在派生类的拷贝构造函数或者派生类的赋值函数中显式调用基类的拷贝构造或赋值运算符函数。

```
class Base
{
public:
    Base(const char *data)
    : _data(new char[strlen(data) + 1]())
    {
        cout << "Base(const char *)" << endl;
        strcpy(_data, data);
    }

    Base(const Base &rhs)
    : _data(new char[strlen(rhs._data) + 1]())
    {
        cout << "Base(const Base &)" << endl;
        strcpy(_data, rhs._data);
    }

    Base &operator=(const Base &rhs)
    {
        cout << "Base &operator=(const Base &)" << endl;
```

```
        if(this != &rhs)
        {
            delete [] _data;

            _data = new char[strlen(rhs._data) + 1]();
            strcpy(_data, rhs._data);
        }
        return *this;
    }

    ~Base()
    {
        cout << "~Base()" << endl;
        delete [] _data;
    }

    const char *data() const
    {
        return _data;
    }
private:
    char *_data;
};

class Derived
: public Base
{
public:
    Derived(const char *data, const char *data2)
    : Base(data)
    , _data2(new char[strlen(data2) + 1]())
    {
        cout << "Derived(const char *, const char *)" << endl;
        strcpy(_data2, data2);
    }

    Derived(const Derived &rhs)
    : Base(rhs)
    , _data2(new char[strlen(rhs._data2) + 1]())
    {
        cout << "Derived(const Derived &)" << endl;
        strcpy(_data2, rhs._data2);
    }

    Derived &operator=(const Derived &rhs)
    {
        cout << "Derived & operator=(const Derived &)" << endl;

        if(this != &rhs)
        {
            Base::operator=(rhs);//显式调用赋值运算符函数
            delete [] _data2;

            _data2 = new char[strlen(rhs._data2) + 1]();
            strcpy(_data2, rhs._data2);
        }
        return *this;
    }
```

```
79
80      ~Derived()
81      {
82          cout << "~Derived()" << endl;
83      }
84
85      friend std::ostream &operator<<(std::ostream &os, const Derived &rhs);
86  private:
87      char *_data2;
88  };
89
90  std::ostream &operator<<(std::ostream &os, const Derived &rhs)
91  {
92      os << rhs.data() << "," << rhs._data2;
93      return os;
94  }
95
96  int main(void)
97  {
98      Derived d1("hello", "world");
99      cout << "d1 = " << d1 << endl;
100
101     Derived d2 = d1;
102     cout << "d1 = " << d1 << endl;
103     cout << "d2 = " << d2 << endl;
104
105     Derived d3("guangdong", "shenzhen");
106     cout << "d3 = " << d3 << endl;
107
108     d3 = d1;
109     cout << "d1 = " << d1 << endl;
110     cout << "d3 = " << d3 << endl;
111
112     return 0;
113 }
```