第六周：多态

第一节 虚函数和多态的基本概念

1.虚函数

在类的定义中，前面有 virtual 关键字的成员函数就是虚函数。

```
class base{
    virtual int get();
}
int base::get(){}
```

virtual 关键字只在类定义里的函数声明中，写函数体的时候不用。**构造函数和静态函数不能使用 virtual，即不能成为虚函数。**

2.多态

（1）表现形式一

派生类的指针可以赋给基类指针。

通过基类指针调用基类和派生类中的同名虚函数时：

①若该指针指向一个基类的对象，那么背调用的是基类的虚函数；

②若该指针指向一个派生类的对象，那么被调用的是派生类的虚函数。

这种机制称为多态。例如：

```
class CBase{
    public:
    virtual void SomeVirtualFunction(){}
};
class CDerived:public CBase{
    public:
    virtual void SomVirtualFunction(){}
};
int main(){
    CDerived ODerived;
    CBase *p = &ODerived;
    p->SomeVirtualFunction();//p 指向派生类对象，所以调用派生类里面的
    return 0;
}
```

（2）表现形式二

派生类的对象可以赋给基类引用

通过基类引用调用基类和派生类中的同名虚函数时：

①若该引用引用的是一个基类的对象，那么被调用的是基类的虚函数；

②若该引用引用的是一个派生类的对象，那么调用的是派生类的虚函数。

```
class CBase{
    public:
    virtual void SomeVirtualFunction(){}
```

```
};
class CDerived:public CBase{
    public:
    virtual void SomVirtualFunction(){}
};
int main(){
    CDerived ODerived;
    CBase &r = ODerived;
    r->SomeVirtualFunction();//在这里调用的是派生类中的
    return 0;
}
```

3.多态的作用

在面向对象的程序设计中使用多态，能够增强程序的可扩充性，即程序需要修改或增加功能的时候，需要改动和增加的代码较少。

第二节 多态实例：魔法门之英雄无敌

在游戏中有很多怪物，没种怪物都有一个类与之对应，每个怪物就是一个对象。怪物能够互相攻击，攻击敌人和被攻击时都有相应的动作，动作是通过对象的成员函数实现的。游戏版本升级时，要增加新的怪物——雷鸟（CThunderBird）。如何编程使代码改动和增加量较小？

基本思路：

为每个怪物类编写 Attack、FightBack 和 Hurted 成员函数。

Attact 函数表现攻击动作，攻击某个怪物，并调用被攻击怪物的 Hurted 函数，以减少被攻击怪物的生命值，同时也调用被攻击怪物的 FightBack 成员函数，遭受被攻击怪物反击。

Hurted 函数减少自身生命值，并表现受伤动作。

FightBack 成员函数表现反击动作，并调用被反击对象的 Hurted 成员函数，使被反击对象受伤。

设置基类 CCreature，并且使 CDragon、CWolf 等其他类都从该基类派生而来，如图 2.1 所示。

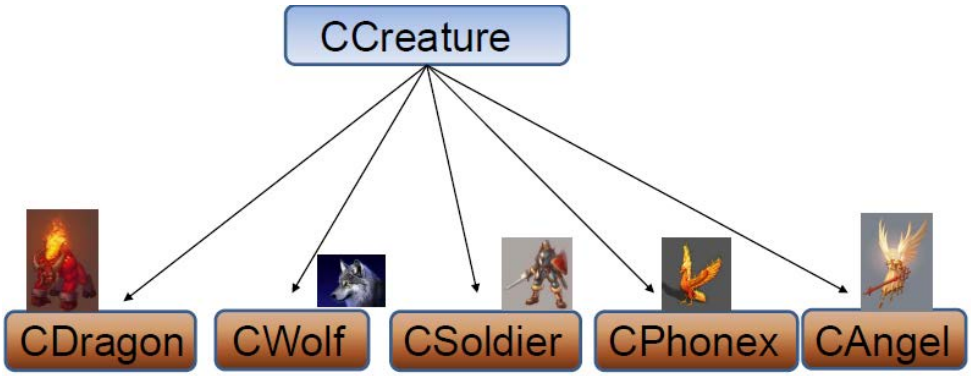

图 2.1 基本思路图

1.非多态实现方法：

```
class class CCreature {
    protected:
    int nPower ; //代表攻击力
```

```
    int nLifeValue ; //代表生命值
};
class CDragon:public CCreature   {
    public:
    void Attack(CWolf * pWolf) {
        //. . . 表现攻击动作的代码
        pWolf->Hurted( nPower);
        pWolf->FightBack( this);
    }
    void Attack(CGhost * pGhost){
        //表现攻击动作代码
        pGhost->Hurted( nPower);
        pGhost->FightBack( this);
    }
    void Hurted(int nPower){
        //表现受伤动作代码
        nLifeValue -=nPower;
    }
    void FightBack(CWolf *pWolf){
        //表现反击动作的代码
        pWolf->Hurted(nPower/2);
    }
    void FightBack(CGhost *pGhost){
        //表现反击动作代码
        pGhost->Hurted(nPower/2);
    }
}
//……还有很多
```

有 n 种怪物，CDrogon 就要有 n 个 Attack 成员函数和 n 个 FightBack 成员函数，对于其他类也是如此。

如果游戏版本升级，增加了新的怪物雷鸟 CthunderBird，则程序改动非常大，所有的类都需要增加两个成员函数（攻击和反击）。

2.多态实现方法：

代码如下：

```
class class CCreature {
    protected:
    int nPower ; //代表攻击力
    int nLifeValue ; //代表生命值
    public:
    virtual void Attack(CCreature *pCreature){}
    virtual void Hurted(int nPower){}
    virtual void FightBack(CCreature *pCreature){}
};
class CDragon:public CCreature   {
```

```
    public:
        virtual void Attack(CCreature *pCreature);
    virtual void Hurted(int nPower);
    virtual void FightBack(CCreature *pCreature);
};

void CDragon::Attack(CCreature * p) {
        //... 表现攻击动作的代码
        p->Hurted(m_nPower);
        pWolf->FightBack(this);
}
void CDragon::Hurted(int nPower){
        //表现受伤动作代码
        m_nLifeValue -=nPower;
    }
void CDragon::FightBack(CCreature *p){
        //表现反击动作的代码
        pWolf->Hurted(m_nPower/2);
}
```

基类只有一个 Attack、FightBack 成员函数，所有 CCreature 的派生类也一样。

如果游戏版本升级，增加了新的怪物雷鸟 CThunderBird，只需要编写心累 CThunderBird，不需要在已有的类例专门为新怪物增加：

```
void Attack( CThunderBird * pThunderBird) ;
void FightBack( CThunderBird * pThunderBird) ;
```

第三节 多态实例：几何形体程序

几何形体处理程序：输入若干个几何形体的参数，要求按面积排序输出，输出时要指明形状。

输入：

第一行是几何体数目 n（不超过 100 行），下面有 n 行，每行以一个字母开头：

（1）若字母为“R”，则代表一个矩形，本行后面跟着两个整数，分别是矩形的宽和高；

（2）若字母为“C”，则代表一个圆，本行后面跟着一个整数代表其半径；

（3）若字母为“T”，则代表一个三角形，本行后面跟着三个整数，代表三条边的长度。

输出：

按面积从小到大依次输出每个几何形体的种类及面积。每行一个几何形体，输出格式为：形体名称：面积

代码如下：

```
#include <iostream>
#include <stdlib.h>
#include <math.h>
using namespace std;
class CShape {
public:
```

```
    virtual double Area() = 0;//纯虚函数
    virtual void PrintInfo() = 0;
};

class CRectangle :public CShape {
public:
    int w, h;
    virtual double Area();
    virtual void PrintInfo();
};

class CCircle :public CShape {
public:
    int r;
    virtual double Area();
    virtual void PrintInfo();
};

class CTriangle :public CShape {
public:
    int a, b, c;
    virtual double Area();
    virtual void PrintInfo();
};
double CRectangle::Area() {
    return w*h;
}
void CRectangle::PrintInfo() {
    cout << "Rectangle:" << Area() << endl;
}
double CCircle::Area() {
    return 3.14*r*r;
}

void CCircle::PrintInfo() {
    cout << "Circle:" << Area() << endl;
}
double CTriangle::Area() {
    double p = (a + b + c) / 2.0;
    return sqrt(p*(p - a)*(p - b)*(p - c));
}

void CTriangle::PrintInfo() {
    cout << "Triangle:" << Area() << endl;
```

```
}
CShape * pShapes[100];//基类指针，可以指向不同派生类
int MyCompare(const void *s1, const void*s2);
int main() {
    int i, n;
    CRectangle *pr; CCircle *pc; CTriangle *pt;
    cin >> n;
    for (i = 0; i < n; i++) {
        char c;
        cin >> c;
        switch (c) {
        case 'R':
            pr = new CRectangle();
            cin >> pr->w >> pr->h;
            pShapes[i] = pr;
            break;
        case 'C':
            pc = new CCircle();
            cin >> pc->r;
            pShapes[i] = pc;
            break;
        case 'T':
            pt = new CTriangle();
            cin >> pt->a >> pt->b >> pt->c;
            pShapes[i] = pt;
            break;
        }
    }
    qsort(pShapes, n, sizeof(CShape*), MyCompare);
    for (int i = 0; i < n; i++)
        pShapes[i]->PrintInfo();//利用多态操作不同类的输出函数

    system("pause");
    return 0;

}
int MyCompare(const void * s1, const void *s2) {
    double a1, a2;
    CShape **p1;//s1,s2是void*,不可写*s1来取得s1指向的内容
    CShape **p2;
    p1 = (CShape **)s1;//s1,s2指向pShapes数组中的元素，数组元素的类型是CShape *
    p2 = (CShape **)s2;
    a1 = (*p1)->Area();
    a2 = (*p2)->Area();
```

```
    if (a1 < a2)
        return  -1;
    else if (a2 < a1)
        return 1;
    else
        return 0;
}
```

因为 s1 和 s2 都是 void*，所以没办法直接获取到指向的内容。所以设置了两个 CShape** 的变量 p1 和 p2，把 s1 和 s2 强制转换成 CShape**类型。s1 和 s2 是指向 pShapes 数组中的元素，数组元素类型为 CShape*。p1，p2 是指向指针的指针，所以要用两个*。

用基类指针数组存放指向各种派生类对象的指针，然后遍历该数组，就能对各个派生类对象做各种操作。

又一个例子，还是比较好理解的：

```
class Base {
public:
   void fun1() { this->fun2(); } //this是基类指针，fun2是虚函数，所以是多态
   virtual void fun2() {  cout << "Base::fun2()" << endl;  }
};
class Derived:public Base {
public:
     virtual void fun2() { cout << "Derived:fun2()" << endl; }
};
int main() {
        Derived d;
        Base * pBase = & d;
        pBase->fun1();
        return 0;
}
```

输出： Derived:fun2()

在非构造函数，非析构函数的成员函数中调用虚函数，是多态!!!

50

图 3.1 多态又一例子

在非构造函数、非析构函数的成员函数中调用虚函数，是多态。

构造函数和析构函数中调用虚函数，不是多态。编译时即可确定，调用的函数是自己的类或基类中定义的函数，不会等到运行时才决定调用自己的还是派生类的函数。

派生类中和基类中虚函数同名同参数表的函数，不加 virtual 也自动成为虚函数。

```
class Base {
private:
   virtual void fun2()  { cout << "Base::fun2()" << endl; }
};
class Derived:public Base {
public:
     virtual void fun2() { cout << "Derived:fun2()" << endl; }
};
Derived d;
Base * pBase = & d;
pBase -> fun2();  // 编译出错
```

图 3.2 虚函数访问权限介绍

在这里会出现编译出错。因为语法检查是不考虑运行结果，所以在 fun2()是 Base 的私有成员的时候，语法检查直接给判错，即便运行到此时实际上按照规定是调用的 Derived 的公有成员 fun2()。

但是如果基类中是 public，派生类中即使是 private，编译依然能通过，因为语法检查还是只看能不能访问基类的，运行的时候再看实际情况。这里面因为基类指针指向了派生类，所以派生类中成员函数是 private 也是可以运行出来的。

第四节 多态的实现原理

“多态”的关键在于通过基类指针或引用调用一个虚函数时，编译时不确定到底调用的是基类还是派生类的函数，运行时才确定——这叫“动态联编”。**当然多态就是在运行的时候才确定调用谁的函数，而不是在编译的时候确定**。

例程如下：

```
class Base{
    public:
    int i;
    virtual void Print(){cout << "Base Print"<<endl;}
};
class Derived:public Base{
    public:
    int n;
    virtual void Print){cout<<"Derived Print"<<endl;}
};
int main(){
    Derived d;
    cout<<sizeof(Base)<<sizeof(Derived);
    return 0;
}
```

输出结果本来我们认为是 4 和 8，但实际上却是 8 和 12，这是为什么呢？

1.多态实现的关键——虚函数表

每一个有虚函数的类（或有虚函数的类的派生类）都有一个虚函数表，该类的任何对象中都放着虚函数表的指针。虚函数表中列出了该类的虚函数地址。多出来的 4 个字节就是用来放虚函数表的地址的。

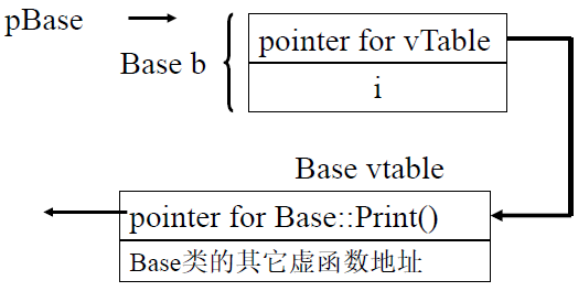

图 4.1 Base 类的虚函数表

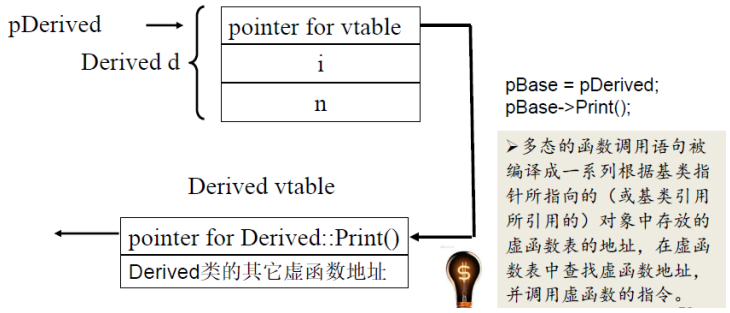

图 4.2 Deriverd 类的虚函数表

多态程序会有额外的时间（查询函数表）和空间开销（虚函数表）。

又一个例程：

```
#include <iostream>
using namespace std;
class A{
    public:virtual void Func(){cout <<"A::Func""<<endl;}
};
class B: public A{
    public:virtual void Func(){cout<<"B::Func"<<endl;}
};
int main(){
    A a;
    A* pa = new B();
    pa->Func();//显然，输出结果为 B::Func
    //在 64 位电脑执行，64 位程序指针为 8 字节
    long long *p1 = (long long *)&a;//把 a 的地址转换成一个 longlong 型的地址指针赋值给 p1
    long long *p2 = (long long *)pa;//把 new 出来的 B 的地址转换成 longlong 型的地址指针赋给 p2
    *p2 = *p1;//=是赋值号。把 p1 指向地方的内容放在 p2 指向的地方去。
//由于 p1 是 longlong 型，所以*p1 是 8 字节的内容。8 个地址正好也是一个地址，这个地址放在 a 对象的开头，是 classA 的虚函数表的地址。
//p2 指向 New 出来的 B 对象，所以 p2 的开头也应该放着 classB 的虚函数表的地址，也是 8 个字节。
//把 p1 指向的 8 个字节赋值给了 p2 指向的 8 个字节，所以下面语句输出值为 A::Func
    pa->Func();
    return 0;
}
```

第五节 虚析构函数、纯虚函数和抽象类

1.虚析构函数（不是多态）

通过基类的指针删除派生类对象时，通常情况下只调用基类的析构函数。但是，删除一个派生类的对象时，应该先调用派生类的析构函数，然后调用基类的析构函数。

解决方法：把基类的析构函数声明为 virtual。

派生类的析构函数可以不进行声明也是虚函数。通过基类的指针删除派生类对象时，首先调用派生类的析构函数，然后调用基类的析构函数。

一般来说，**一个类如果定义了虚函数，则应该将析构函数也定义成虚函数**。或者，一个类打算作为基类使用，也应该将析构函数定义成虚函数。

注意：**不允许以虚函数作为构造函数**。

例子：

```
class son{
    public:
    virtual ~son() {cout<<"bye from son"<<endl;}; };
class grandson:public son{
    public:
    ~grandson(){cout<<"bye from grandson"<<endl;}; };
int main() {
    son *pson;
    pson= new grandson();
    delete pson;
    return 0;
}
```

输出结果：bye from grandson

bye from son

2.纯虚函数和抽象类

纯虚函数：没有函数体的虚函数，**不等同于函数体无语句!!!**。

格式如下：

```
virtual void Print( )=0;//纯虚函数
```

包含纯虚函数的类叫抽象类。抽象类只能作为基类来派生新类使用，**不能创建抽象类的对象**；抽象类的指针和引用可以指向由抽象类派生出来的类的对象。

如下例：

```
A a ; //  错，  A  是抽象类，不能创建对象
A * pa ; // ok,可以定义抽象类的指针和引用
pa = new A ; //错误, A  是抽象类，不能创建对象
```

在抽象类的成员函数内可以调用纯虚函数，但是在构造函数或析构函数内部不能调用纯虚函数。

如果一个类从抽象类派生而来，那么当且仅当它实现了基类中的所有纯虚函数，它才能成为非抽象类。

例程：

```
class A{
    public:
    virtual void f() =0;//纯虚函数
    void g(){this->f();//ok
    }
    A(){//f();//错误
    }
};
class B:public A{
```

```
    public:
    void f(){cout<<"B:f()"<<endl;}
};
int main(){
    B b;
    b.g();
    return 0;
}
```

输出：B.f()