程序设计实习

C++ 面向对象程序设计

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2025 年 3 月 7 日

大纲

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。

```
1  class Base {
2    virtual int get();
3  };
4  int Base::get() {
5    //...
6  }
```

virtual 关键字只用在类定义里的函数声明中，写函数体时不用。

多态的表现形式-1

派生类的指针可以赋给基类指针。

多态的表现形式-1

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同参虚函数时:

多态的表现形式-1

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同参虚函数时:

- 若该指针指向一个基类的对象，那么被调用是基类的虚函数；
- 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。

多态的表现形式-1

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同参虚函数时:

- 若该指针指向一个基类的对象，那么被调用是基类的虚函数;
- 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。

这种机制就叫做“多态”。

多态的表现形式-1

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同参虚函数时:

- 若该指针指向一个基类的对象，那么被调用是基类的虚函数;
- 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。

这种机制就叫做“多态”。

```
4   class CBase {
5   public:
6     virtual void someVirtualFunction() { cout << "base function" << endl; }
7   };
8   class CDerived : public CBase {
9   public:
10    virtual void someVirtualFunction() { cout << "derived function" << endl; }
11  };
12  int main() {
13    CDerived ODerived;
14    CBase *p = &ODerived;
15    p->someVirtualFunction(); //调用哪个虚函数取决于 p 指向哪种类型的对象
16    return 0;
17  }
```

多态的表现形式-2

派生类的对象可以赋给基类引用。

多态的表现形式-2

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同参虚函数时:

多态的表现形式-2

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同参虚函数时:

- 若该引用引用的是一个基类的对象，那么被调用是基类的虚函数;
- 若该引用引用的是一个派生类的对象，那么被调用的是派生类的虚函数。

多态的表现形式-2

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同参虚函数时:

- 若该引用引用的是一个基类的对象，那么被调用是基类的虚函数;
- 若该引用引用的是一个派生类的对象，那么被调用的是派生类的虚函数。

这种机制也叫做“多态”。

多态的表现形式-2

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同参虚函数时:

- 若该引用引用的是一个基类的对象，那么被调用是基类的虚函数;
- 若该引用引用的是一个派生类的对象，那么被调用的是派生类的虚函数。

这种机制也叫做“多态”。

```
4   class CBase {
5   public:
6     virtual void someVirtualFunction() { cout << "base function" << endl; }
7   };
8   class CDerived : public CBase {
9   public:
10    virtual void someVirtualFunction() { cout << "derived function" << endl; }
11  };
12  int main() {
13    CDerived ODerived;
14    CBase &r = ODerived;
15    r.someVirtualFunction(); //调用哪个虚函数取决于 r 引用哪种类型的对象
16    return 0;
17  }
```

多态的简单示例

```cpp
#include <iostream>
using namespace std;

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

```cpp
int main() {
  A a;
  B b;
  E e;
  A *pa = &a;
  B *pb = &b;
  E *pe = &e;

  pa->print();
  pa = pb;
  pa->print();
  pa = pe;
  pa->print();
  return 0;
}
```

多态的简单示例

```cpp
#include <iostream>
using namespace std;

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

```cpp
int main() {
  A a;
  B b;
  E e;
  A *pa = &a;
  B *pb = &b;
  E *pe = &e;

  pa->print();
  pa = pb;
  pa->print();
  pa = pe;
  pa->print();
  return 0;
}
```

输出结果：

```
A::Print
B::Print
E::Print
```

单选题

下面关于多态的说法哪个正确?

- A 通过基类指针调用基类和派生类里的同名同参函数，是多态
- B 通过基类对象调用基类和派生类里的同名同参函数，不是多态
- C 通过基类引用调用基类和派生类里的同名同参函数，是多态
- D 以上都不对

单选题

下面关于多态的说法哪个正确?

- A 通过基类指针调用基类和派生类里的同名同参函数，是多态
- B 通过基类对象调用基类和派生类里的同名同参函数，不是多态
- C 通过基类引用调用基类和派生类里的同名同参函数，是多态
- D 以上都不对

答案：B

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。

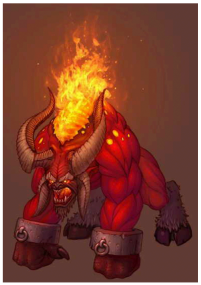
使用多态的游戏程序实例

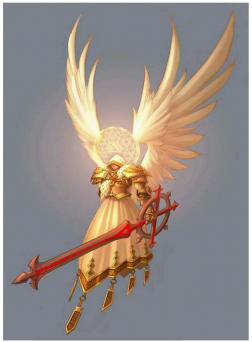
游戏 << 魔法门之英雄无敌 >>
游戏中有很多种怪物，每种怪物都有一个类与之对应，每个怪物就是一个对象。

类：CSoldier

类CPhoenix

类：CDragon





类：CAngel

使用多态的游戏程序实例

游戏 << 魔法门之英雄无敌 >>
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。

使用多态的游戏程序实例

游戏 << 魔法门之英雄无敌 >>
游戏版本升级时，要增加新的怪物 – – 雷鸟。
如何编程才能使升级时的代码改动和增加量较小?

使用多态的游戏程序实例

游戏 << 魔法门之英雄无敌 >>
基本思路：为每个怪物类编写 `attack`，`fightBack`，`hurt`成员函数。

使用多态的游戏程序实例

游戏 << 魔法门之英雄无敌 >>
基本思路：为每个怪物类编写 attack，fightBack，hurt成员函数。

- attack函数表现攻击动作，攻击某个怪物，并调用被攻击怪物的 hurt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。

使用多态的游戏程序实例

游戏 << 魔法门之英雄无敌 >>
基本思路：为每个怪物类编写 attack，fightBack，hurt成员函数。

- attack函数表现攻击动作，攻击某个怪物，并调用被攻击怪物的 hurt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。
- hurt函数减少自身生命值，并表现受伤动作。

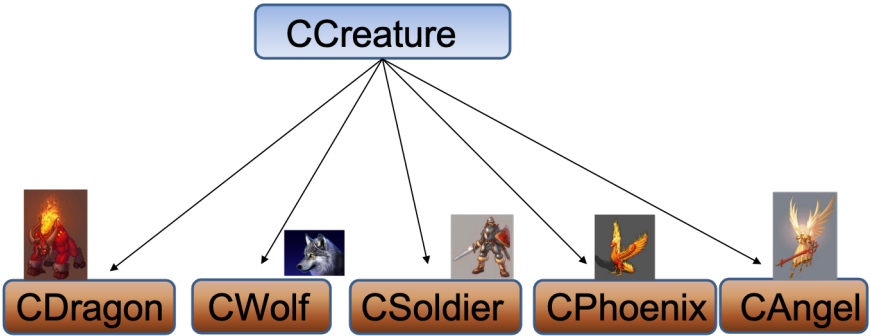
使用多态的游戏程序实例

游戏 << 魔法门之英雄无敌 >>
基本思路：为每个怪物类编写 attack，fightBack，hurt成员函数。

- attack函数表现攻击动作，攻击某个怪物，并调用被攻击怪物的 hurt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。
- hurt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hurt成员函数，使被反击对象受伤。

使用多态的游戏程序实例

游戏 << 魔法门之英雄无敌 >>
基本思路：设置基类 CCreature，并且使CDragon，CWolf等其他类都从CCreature派生而来。



非多态的实现方法

```
1  class class CCreature {
2  protected:
3    int nPower;     //代表攻击力
4    int nLifeValue; //代表生命值
5  };
6  class CDragon : public CCreature {
7  public:
8    void attack(CWolf *pWolf) {
9      //表现攻击动作的代码
10     pWolf->hurt(nPower);
11     pWolf->fightBack(this);
12   }
13   void attack(CGhost *pGhost) {
14     //表现攻击动作的代码
15     pGhost->hurt(nPower);
16     pGohst->fightBack(this);
17   }
18   void hurt(int nPower) {
19     //表现受伤动作的代码
20     nLifeValue -= nPower;
21   }
22   void fightBack(CWolf *pWolf) {
23     //表现反击动作的代码
24     pWolf->hurt(nPower / 2);
25   }
26   void fightBack(CGhost *pGhost) {
27     //表现反击动作的代码
28     pGhost->hurt(nPower / 2);
29   }
30 }
```

非多态的实现方法的缺点

如果游戏版本升级，增加了新的怪物雷鸟 CThunderBird，则程序改动较大。

非多态的实现方法的缺点

如果游戏版本升级，增加了新的怪物雷鸟 CThunderBird，则程序改动较大。所有的类都需要增加两个成员函数:

```
1  void attack(CThunderBird *pThunderBird);
2  void fightBack(CThunderBird *pThunderBird);
```

在怪物种类多的时候，工作量可想而知

多态的实现方法

```
1  //基类 CCreature:
2  class CCreature {
3  protected:
4    int nLifeValue;
5    int nPower;
6  public:
7    virtual void attack(CCreature *pCreature) {}
8    virtual void hurt(int nPower) {}
9    virtual void fightBack(CCreature *pCreature) {}
10 };
```

基类只有一个attack 成员函数；也只有一个fightBack成员函数；所有CCreature 的派生类也是这样。

多态的实现方法

```
1  //派生类   CDragon:
2  class CDragon : public CCreature {
3  public:
4    virtual void attack(CCreature *pCreature);
5    virtual void hurt(int nPower);
6    virtual void fightBack(CCreature *pCreature);
7  };
8  void CDragon::attack(CCreature *p) {
9    //表现攻击动作的代码
10   p->hurt(nPower);  //多态
11   p->fightBack(this); //多态
12 }
13 void CDragon::hurt(int _nPower) {
14   //表现受伤动作的代码
15   nLifeValue -= _nPower;
16 }
17 void CDragon::fightBack(CCreature *p) {
18   //表现反击动作的代码
19   p->hurt(nPower / 2); //多态
20 }
```

多态实现方法的优势

如果游戏版本升级，增加了新的怪物雷鸟`CThunderBird`。

多态实现方法的优势

如果游戏版本升级，增加了新的怪物雷鸟CThunderBird。
只需要编写新类CThunderBird, 不需要在已有的类里专门为新怪物增加：

```
1  void attack(CThunderBird *pThunderBird);
2  void fightBack(CThunderBird *pThunderBird);
```

已有的类可以原封不动

多态实现方法的优势

```
1  CDragon Dragon;
2  CWolf Wolf;
3  CGhost Ghost;
4  CThunderBird Bird;
5  Dragon.attack(&Wolf);  //(1)
6  Dragon.attack(&Ghost); //(2)
7  Dragon.attack(&Bird);  //(3)
```

多态实现方法的优势

```
CDragon Dragon;
CWolf Wolf;
CGhost Ghost;
CThunderBird Bird;
Dragon.attack(&Wolf);  //(1)
Dragon.attack(&Ghost); //(2)
Dragon.attack(&Bird);  //(3)
```

```
void CDragon::attack(CCreature *p) {
  p->hurt(nPower);  //多态
  p->fightBack(this);  //多态
}
```

多态实现方法的优势

```
1  CDragon Dragon;
2  CWolf Wolf;
3  CGhost Ghost;
4  CThunderBird Bird;
5  Dragon.attack(&Wolf);  //(1)
6  Dragon.attack(&Ghost); //(2)
7  Dragon.attack(&Bird);  //(3)
```

```
1  void CDragon::attack(CCreature *p) {
2    p->hurt(nPower);  //多态
3    p->fightBack(this);  //多态
4  }
```

根据多态的规则，上面的 (1)，(2)，(3) 进入到CDragon::attack函数后，能分别调用：

```
CWolf::hurt
CGhost::hurt
CBird::hurt
```

几何形体处理程序

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:
第一行是几何形体数目 n（不超过 100）. 下面有 n 行，每行以一个字母 c 开头.
若 c 是'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;
若 c 是'C'，则代表一个圆，本行后面跟着一个整数代表其半径
若 c 是'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

几何形体处理程序

Sample Input:

```
3
R 3 5
C 9
T 3 4 5
```

Sample Output:

```
Triangle:6
Rectangle:15
Circle:254.34
```

几何形体处理程序

```
5  class CShape {
6  public:
7    virtual double area() = 0; //纯虚函数
8    virtual void printInfo() = 0;
9  };
10
11 class CRectangle : public CShape {
12 public:
13   int w;
14   int h;
15   virtual double area();
16   virtual void printInfo();
17 };
18 double CRectangle::area() {
19   return w * h;
20 }
21 void CRectangle::printInfo() {
22   cout << "Rectangle:" << area() << endl;
23 }
```

几何形体处理程序

```
25 class CCircle : public CShape {
26 public:
27   int r;
28   virtual double area();
29   virtual void printInfo();
30 };
31 double CCircle::area() {
32   return 3.14 * r * r;
33 }
34 void CCircle::printInfo() {
35   cout << "Circle:" << area() << endl;
36 }
```

几何形体处理程序

```
38 class CTriangle : public CShape {
39 public:
40   int a;
41   int b;
42   int c;
43   virtual double area();
44   virtual void printInfo();
45 };
46 double CTriangle::area() {
47   double p = (a + b + c) / 2.0;
48   return sqrt(p * (p - a) * (p - b) * (p - c));
49 }
50 void CTriangle::printInfo() {
51   cout << "Triangle:" << area() << endl;
52 }
```

几何形体处理程序

```
54  int MyCompare(const void *s1, const void *s2) {
55    double a1, a2;
56    CShape **p1; // s1,s2 是 void * , 不可写 “* s1”来取得 s1 指向的内容
57    CShape **p2;
58    p1 = (CShape **)s1; // s1,s2 指向 pShapes 数组中的元素,
59                        //数组元素的类型是 CShape *
60    p2 = (CShape **)s2; // 故 p1,p2 都是指向指针的指针, 类型为 CShape **
61    a1 = (*p1)->area(); // * p1 的类型是 CShape * , 是基类指针, 故此句为多态
62    a2 = (*p2)->area();
63    if (a1 < a2)
64      return -1;
65    else if (a2 < a1)
66      return 1;
67    else
68      return 0;
69  }
```

几何形体处理程序

```
71  CShape *pShapes[100];
72
73  int main() {
74    int n;
75    CRectangle *pr;
76    CCircle *pc;
77    CTriangle *pt;
78    cin >> n;
79    for (int i = 0; i < n; i++) {
80      char c;
81      cin >> c;
82      switch (c) {
83      case 'R':
84        pr = new CRectangle();
85        cin >> pr->w >> pr->h;
86        pShapes[i] = pr;
87        break;
```

```
88      case 'C':
89        pc = new CCircle();
90        cin >> pc->r;
91        pShapes[i] = pc;
92        break;
93      case 'T':
94        pt = new CTriangle();
95        cin >> pt->a >> pt->b >> pt->c;
96        pShapes[i] = pt;
97        break;
98      }
99    }
100   qsort(pShapes, n, sizeof(CShape *), MyCompare);
101   for (int i = 0; i < n; i++)
102     pShapes[i]->printInfo();
103   return 0;
104 }
```

几何形体处理程序

如果添加新的几何形体，比如五边形，则只需要从CShape派生出CPentagon, 以及在main中的switch语句中增加一个case，其余部分不变!

几何形体处理程序

如果添加新的几何形体，比如五边形，则只需要从CShape派生出CPentagon, 以及在main中的switch语句中增加一个case，其余部分不变!

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，是很常用的做法

多态的又一例子

```
4  class Base {
5  public:
6    void fun1() {
7      fun2();
8    }
9    virtual void fun2() {
10     cout << "Base::fun2()" << endl;
11   }
12 };
13 class Derived : public Base {
14 public:
15   virtual void fun2() {
16     cout << "Derived:fun2()" << endl;
17   }
18 };
19 int main() {
20   Derived d;
21   Base *pBase = &d;
22   pBase->fun1();
23   return 0;
24 }
```

程序的输出结果是

- A Base::fun2()
- B Derived::fun2()
- C Hello,world
- D

多态的又一例子

```
4  class Base {
5  public:
6    void fun1() {
7      fun2();
8    }
9    virtual void fun2() {
10     cout << "Base::fun2()" << endl;
11   }
12 };
13 class Derived : public Base {
14 public:
15   virtual void fun2() {
16     cout << "Derived:fun2()" << endl;
17   }
18 };
19 int main() {
20   Derived d;
21   Base *pBase = &d;
22   pBase->fun1();
23   return 0;
24 }
```

程序的输出结果是

A. Base::fun2()
B. Derived::fun2()
C. Hello,world
D.

答案：B

成员函数调用虚函数

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

在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。

virtual 的传递

```
4  class myclass {
5  public:
6    virtual void hello() {
7      cout << "hello from myclass" << endl;
8    }
9    virtual void bye() {
10     cout << "bye from myclass" << endl;
11   }
12 };
13 class son : public myclass {
14 public:
15   void hello() {
16     cout << "hello from son" << endl;
17   }
18   son() {
19     hello();
20   }
21   ~son() {
22     bye();
23   }
24 };
```

```
25 class grandson : public son {
26 public:
27   void hello() {
28     cout << "hello from grandson" << endl;
29   };
30   void bye() {
31     cout << "bye from grandson" << endl;
32   }
33   grandson() {
34     cout << "constructing grandson" << endl;
35   }
36   ~grandson() {
37     cout << "destructing grandson" << endl;
38   }
39 };
40 int main() {
41   grandson gson;
42   son *pson;
43   pson = &gson;
44   pson->hello(); //多态
45   return 0;
46 }
```

virtual 的传递

输出结果：

```
hello from son
constructing grandson
hello from grandson
destructing grandson
bye from myclass
```

virtual 的传递

输出结果：

```
hello from son
constructing grandson
hello from grandson
destructing grandson
bye from myclass
```

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

虚函数的访问权限

```cpp
class Base {
private:
  virtual void fun2() {
    cout << "Base::fun2()" << endl;
  }
};
class Derived : public Base {
public:
  virtual void fun2() {
    cout << "Derived:fun2()" << endl;
  }
};
```

```cpp
Derived d;
Base *pBase = &d;
pBase->fun2();  // 编译出错
```

编译出错是因为 fun2() 是Base的私有成员。即使运行到此时实际上调用的应该是Derived的公有成员 fun2()也不行，因为语法检查是不考虑运行结果的。

如果将Base中的 private换成public, 即使Derived中的fun2() 是private的，编译依然能通过，也能正确调用Derived::fun2()。

动态联编

“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”。“动态联编”是怎么实现的呢?

多态的实现原理

```
4  class Base {
5  public:
6    int i;
7    virtual void Print() { cout << "Base:Print" << endl; }
8  };
9  class Derived : public Base {
10 public:
11   int n;
12   virtual void Print() { cout << "Drived:Print" << endl; }
13 };
14 int main() {
15   Derived d;
16   cout << sizeof(Base) << "," << sizeof(Derived);
17   return 0;
18 }
```

多态的实现原理

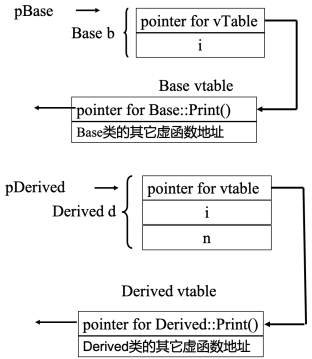
```
4  class Base {
5  public:
6    int i;
7    virtual void Print() { cout << "Base:Print" << endl; }
8  };
9  class Derived : public Base {
10 public:
11   int n;
12   virtual void Print() { cout << "Drived:Print" << endl; }
13 };
14 int main() {
15   Derived d;
16   cout << sizeof(Base) << "," << sizeof(Derived);
17   return 0;
18 }
```

32 位系统下，程序运行输出结果：8,12

64 位系统下，程序运行输出结果：16,16

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。



多态的函数调用语句被编译成一系列根据基类指针所指向的（或基类引用所引用的）对象中存放的虚函数表的地址，在虚函数表中查找虚函数地址，并调用虚函数的指令。

单选题

```cpp
#include <iostream>
using namespace std;

class A {
public:
  virtual void func() { cout << "A::func "; }
};
class B : public A {
public:
  virtual void func() { cout << "B::func "; }
};
int main() {
  A a;
  A *pa = new B();
  pa->func();
  //64 位程序，指针为 8 字节，用 long long
  long long *p1 = (long long *)&a;
  long long *p2 = (long long *)pa;
  *p2 = *p1;
  pa->func();
  return 0;
}
```

左边程序的输出结果是：

- A B::func A::func
- B B::func B::func
- C A::func A::func
- D A::func B::func

单选题

```cpp
#include <iostream>
using namespace std;

class A {
public:
  virtual void func() { cout << "A::func "; }
};
class B : public A {
public:
  virtual void func() { cout << "B::func "; }
};
int main() {
  A a;
  A *pa = new B();
  pa->func();
  //64 位程序，指针为 8 字节，用 long long
  long long *p1 = (long long *)&a;
  long long *p2 = (long long *)pa;
  *p2 = *p1;
  pa->func();
  return 0;
}
```

左边程序的输出结果是：

- A B::func A::func
- B B::func B::func
- C A::func A::func
- D A::func B::func

答案：A

虚析构函数

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。

虚析构函数

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。解决办法：把基类的析构函数声明为`virtual`

- 派生类的析构函数可以`virtual`不进行声明
- 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

虚析构函数

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。解决办法：把基类的析构函数声明为`virtual`

- 派生类的析构函数可以`virtual`不进行声明
- 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。

虚析构函数

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。解决办法：把基类的析构函数声明为virtual

- 派生类的析构函数可以virtual不进行声明
- 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

虚析构函数

```
4  class son {
5  public:
6    ~son() {
7      cout << "bye from son" << endl;
8    }
9  };
10 class grandson : public son {
11 public:
12   ~grandson() {
13     cout << "bye from grandson" << endl;
14   }
15 };
16 int main() {
17   son *pson;
18   pson = new grandson();
19   delete pson;
20   return 0;
21 }
```

虚析构函数

```
4  class son {
5  public:
6    ~son() {
7      cout << "bye from son" << endl;
8    }
9  };
10 class grandson : public son {
11 public:
12   ~grandson() {
13     cout << "bye from grandson" << endl;
14   }
15 };
16 int main() {
17   son *pson;
18   pson = new grandson();
19   delete pson;
20   return 0;
21 }
```

输出：bye from son

没有执行 grandson:: grandson()!!!

虚析构函数

```
4  class son {
5  public:
6    virtual ~son() {
7      cout << "bye from son" << endl;
8    }
9  };
10 class grandson : public son {
11 public:
12   ~grandson() {
13     cout << "bye from grandson" << endl;
14   }
15 };
16 int main() {
17   son *pson;
18   pson = new grandson();
19   delete pson;
20   return 0;
21 }
```

虚析构函数

```
4  class son {
5  public:
6    virtual ~son() {
7      cout << "bye from son" << endl;
8    }
9  };
10 class grandson : public son {
11 public:
12   ~grandson() {
13     cout << "bye from grandson" << endl;
14   }
15 };
16 int main() {
17   son *pson;
18   pson = new grandson();
19   delete pson;
20   return 0;
21 }
```

输出结果

```
bye from grandson
bye from son
```

执行grandson::~grandson()，引起执行son::~son()

纯虚函数和抽象类

纯虚函数：带有纯说明符 = 0的虚函数

```
4  class A {
5  private:
6    int a;
7  public:
8    virtual void print() = 0; //纯虚函数
9    void fun() {
10     cout << "fun";
11   }
12 };
```

- 可以为纯虚函数提供定义（而且如果纯虚函数是析构函数就必须提供）
- 此定义必须在类体之外提供（函数声明的语法不允许纯说明符 $= 0$ 和函数体一起出现）

纯虚函数和抽象类

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

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

纯虚函数和抽象类

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

```
1  A a; // 错，A 是抽象类，不能创建对象
2  A *pa; // ok，可以定义抽象类的指针和引用
3  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。

纯虚函数和抽象类

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

```
1  A a; // 错，A 是抽象类，不能创建对象
2  A *pa; // ok，可以定义抽象类的指针和引用
3  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。

纯虚函数和抽象类

```
4  class A {
5  public:
6    virtual void f() = 0; //纯虚函数
7    void g() {
8      this->f(); // ok
9    }
10   A() {
11     // f(); // 错误
12   }
13 };
14 class B : public A {
15 public:
16   void f() {
17     cout << "B:f()" << endl;
18   }
19 };
20 int main() {
21   B b;
22   b.g();
23   return 0;
24 }
```

纯虚函数和抽象类

```
4  class A {
5  public:
6    virtual void f() = 0; //纯虚函数
7    void g() {
8      this->f(); // ok
9    }
10   A() {
11     // f(); // 错误
12   }
13 };
14 class B : public A {
15 public:
16   void f() {
17     cout << "B:f()" << endl;
18   }
19 };
20 int main() {
21   B b;
22   b.g();
23   return 0;
24 }
```

输出结果:

```
B:f()
```