面向对象程序设计

Object Oriented Programming

主 讲 教 师：陈洪刚
开 课 时 间：2021年
honggang_chen@yeah. net

CHAPTER 6

多态性与虚函数

01 什么是多态性

02 一个典型的例子

03 利用虚函数实现动态多态性

04 纯虚函数与抽象类

6.1 什么是多态性

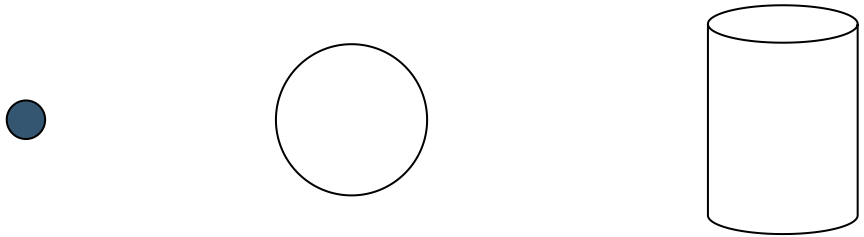
- 多态性是面向对象程序设计的重要特征。
- 多态性： 对不同的对象发送同一个消息，不同的对象在接收时会产生不同的行为。也就是，每个对象可以用自己的方式去响应共同的消息。
- 其中，消息是指对类的成员函数的调用；不同的行为是指不同的实现，也就是调用了不同的函数。
- 函数的重载、运算符重载都是多态现象。
- C++中多态性的表现形式之一：具有不同功能的函数可以用同一个函数名，这样就可以用同一个函数名调用不同内容的函数。
- 多态性是“一个接口，多种方法”。不管对象怎么变，用户都是同样的形式去调用，使他们按照事先的安排分别作出反应。

6.1 什么是多态性

- 从系统实现的观点看，多态性分为两类：静态多态和动态多态性。
- 静态多态性是通过函数重载实现的。函数重载和运算符重载属于静态多态性，在编译程序时系统就可以确定调用哪个函数，因此静态多态性又称编译时的多态性。
- 动态多态性是在程序运行中才能确定操作所针对的对象。它又称运行时的多态性。动态多态性是通过虚函数实现的。
- 这章要研究的问题是：从一个基类派生出不同的派生类时，各派生类可以使用与基类成员相同的成员名，如果在运行时用相同的成员名调用类的成员，会调用哪个类的成员？

6.2 一个典型的例子

- 例 6.1 点-圆-圆柱
- 先建立一个点类(point)，有数据成员x，y(坐标)；
- 以它为基类派生一个圆类，增加数据成员r(半径)；
- 再以圆为直接基类派生出一个圆柱体类，再增加数据成员h(高)；
- 要求重载运算符<<和>>使之能输出以上的类对象。



6.2 一个典型的例子

```
//  (1) 声明基类point类
#include <iostream>
using namespace std;

class Point
{
protected:                          // 保护数据成员
  float x, y;
public:
  Point(float = 0, float = 0); // 带默认参数的构造函数
  void setPoint(float, float); // set函数，赋值用
  float getX() const { return x; } // 常成员函数，返回x
  float getY() const { return y; } // 常成员函数，返回y
  // 友元方式重载<<运算符，显示x, y
  friend ostream & operator<<(ostream &, const Point &);
};
```

```
// Point的构造函数
Point::Point(float a, float b) {
  x = a; y = b;}

// 设置x和y的坐标值
void Point::setPoint(float a, float b) {
  x = a; y = b;}

//    输出点的坐标
ostream & operator<<(ostream
&output, const Point &p)
{output << "[" << p.x << "," << p.y
<< "]" << endl;
  return output;
}
```

6.2 一个典型的例子

```
//  (1) 声明基类point类
#include <iostream>
using namespace std;

class Point
{
protected:                          // 保护数据成员
  float x, y;
public:
  Point(float = 0, float = 0); // 带默认参数的构造函数
  void setPoint(float, float); // set函数，赋值用
  float getX() const { return x; } // 常成员函数，返回x
  float getY() const { return y; } // 常成员函数，返回y
  // 友元方式重载<<运算符，显示x, y
  friend ostream & operator<<(ostream &, const Point &);
};
```

```
int main()
{
  Point p(3.5, 6.4);
  cout <<"x=" << p.getX() << ",y="
<< p.getY() << endl;
  p.setPoint(8.5, 6.8);
  cout << "p(new):" << p << endl;
  return 0;
}

// 执行结果：
x=3.5 y=6.4
p(new):[8.5,6.8]
```

6.2 一个典型的例子

```
// (2) 声明派生类circle
// 在(1)的基础上，再写出声明派生类circle的部分

class Circle : public Point
{
protected:
  float radius;          // 增加保护成员圆的半径
public:
  Circle(float x = 0, float y = 0, float r = 0);
  void setRadius(float);
  float getRadius() const;
  float area () const;          // 计算圆面积
  // 友元方式重载<<运算符，显示圆的信息
  friend ostream &operator<<(ostream &, const
Circle &);
};
```

```
Circle::Circle(float a, float b, float r) : Point(a,
b), radius(r) { }  // 构造
// 设置半径
void Circle::setRadius(float r) {radius=r; }
// 读取半径
float Circle::getRadius() const { return radius; }
// 计算面积
float Circle::area() const
{ return 3.14159 * radius * radius; }
// 重载<<
ostream &operator<<(ostream &output,
const Circle &c)
{output << "Center=[" << c.x << "," << c.y
<< "], Radius=" << c.radius << ", area=" <<
c.area() << endl;
  return output;}
```

6.2 一个典型的例子

```
// (2) 声明派生类circle
// 在(1)基础上，再写出声明派生类circle的部分

class Circle : public Point
{
protected:
  float radius;          // 增加保护成员圆的半径
public:
  Circle(float x = 0, float y = 0, float r = 0);
  void setRadius(float);
  float getRadius() const;
  float area () const;          // 计算圆面积
  // 友元方式重载<<运算符，显示圆的信息
  friend ostream &operator<<(ostream &,
const Circle &);
};
int main()
{Circle c(3.5,6.4,5.2);
  cout << "original circle:\nx=" << c.getX() << ", y="
<< c.getY() << ", r=" << c.getRadius()  << ", area="
<< c.area() << endl;
  c.setRadius(7.5);  c.setPoint(5, 5);
  cout << "new circle:\n" << c;
  Point &pRef = c;  // 派生类对象初始化基类对象引用
                    // pRef不是c的别名，只是c中基类部分的别名
cout << "pRef:" << pRef; //作为基类对象point输出
  return 0;}
// 执行结果:
original circle:
x=3.5, y=6.4, r=5.2, area=84.9486
new circle:
Center=[5,5], r=7.5, area=176.714
pRef:[5,5]
```

6.2 一个典型的例子

```
// (3) 声明circle 的派生类cylinder
以circle为基础，从circle 类派生出cylinder类
class Cylinder : public Circle
{
public:
  Cylinder (float x = 0, float y = 0, float r = 0,
float h = 0);
  void setHeight(float);
  float getHeight() const;
  float area() const;
  float volume() const;
  friend ostream& operator<<(ostream&, const
Cylinder&);
protected:
  float height;
};
```

```
Cylinder::Cylinder(float a, float b, float r, float h)
                : Circle(a, b, r), height(h) { }
void Cylinder::setHeight(float h) { height = h; }
float Cylinder::getHeight() const { return height; }
float Cylinder::area() const
{ return 2 * Circle::area() + 2 * 3.14159 * radius *
height; }
float Cylinder::volume() const
{ return Circle::area() * height; }

ostream &operator<<(ostream &output, const
Cylinder& cy)
{output << "Center=[" << cy.x<<"," << cy.y << "],
r=" << cy.radius << ", h=" << cy.height << "
\narea=" << cy.area() <<
    ", volume=" << cy.volume() << endl;
  return output;}
```

6.2 一个典型的例子

```
// (3) 声明circle 的派生类cylinder
以circle为基础，从circle 类派生出cylinder类
class Cylinder : public Circle
{
public:
  Cylinder (float x = 0, float y = 0, float r =
0, float h = 0);
  void setHeight(float);
  float getHeight() const;
  float area() const;
  float volume() const;
  friend ostream& operator<<(ostream&,
const Cylinder&);
protected:
  float height;
};
```

```
int main()
{
  Cylinder cy1(3.5,6.4,5.2,10);
  cout << "\n original cylinder:\n x=" << cy1.getX() <<
y=" << cy1.getY() << ", r=" << cy1.getRadius() << ",
h=" << cy1.getHeight() << "\narea=" << cy1.area() <<
volume=" << cy1.volume() << endl;
  cy1.setHeight(15);
  cy1.setRadius(7.5);
  cy1.setPoint(5, 5);
  cout << "\nnew cylinder:\n" << cy1;
  Point &pRef = cy1; // 派生类对象初始化基类对象引用
  cout << "\npRef as a point:" << pRef;
  Circle &cRef = cy1; // 派生类对象初始化基类对象引用
  cout << "\ncRef as a Circle:" << cRef;
  return 0;}
```

6.3 利用虚函数实现动态多态性

6.3.1 虚函数的作用

6.3.2 静态关联与动态关联

6.3.4 虚析构函数

6.3.3 在什么情况下应当声明虚函数

6.3.1 虚函数的作用

- 同一个类中，不能出现完全相同的函数：

```
class A {
public:
  int func(int);
  // int func(int); // 错误};
```

- 不同层次的类中，可以出现完全相同的函数：

```
class A { public: int func(int); };
class B : public A { public: int func(int); };
```

- 主函数中：

```
B b;
// 同名覆盖原则调用B类的func
b.func(10);
b.A::func(10); // 调用A类的func
```

- 例6.1中，circle::area用于计算圆的面积，circle的派生类cylinder::area用于计算圆柱体的表面积，这两个函数名字相同，参数也相同，但是功能不同。
- 编译系统按照同名覆盖的原则决定调用哪个函数。

```
cy1.area()
// 调用的是派生类cylinder::area
cy1.circle::area()
// 调用基类circle::area
```

- 能否用同一个调用形式，即能调用派生类的函数也能调用基类的同名函数？

6.3.1 虚函数的作用

- C++ 中的虚函数就是解决这个问题的。虚函数的原理是在派生类中定义与基类函数同名的函数，通过基类指针或引用访问基类或派生类中的同名函数。

```
class Student {
public:
  Student(int,string,float); // 构造函数
  void display();        // 输出函数
protected:               // 保护成员
  int num;
  string name;
  float score;};

Student::Student(int n, string nam, float s)
{ num = n; name = nam; score=s; }
```

```
void Student::display() {
  cout << "num:" << num << "\nname:" <<name
<< "\nscore:" << score << "\n\n";}


class Graduate : public Student {  // 派生类
public:
  Graduate(int, string, float, float);
  void display();        // 基类同名函数
private:
  float wage;
};
```

6.3.1 虚函数的作用

```
Graduate::Graduate(int n, string nam,
float s, float w) : Student(n, nam, s),
wage(w) { }

void Graduate::display()
{
  cout << "num:" << num << "\nname:"
<< name << "\nscore:" << score <<
"\nwage=" << wage << endl;
}
```

```
int main()
{
  Student stud1(1001, "Li", 87.5);
  Graduate grad1(2001, "Wang", 98.5, 563.5);
  Student *pt = &stud1; // 基类指针指向基类对象
  pt->display();        // 输出什么?
  pt = &grad1;          // 基类指针指向派生类对象
  pt->display();        // 输出什么?
  return 0;
}
```

- 程序运行结果如下：

num:1001	num:2001
name:Li	name:Wang
score:87.5	score:98.5

- 原因是指针是指向基类的指针。用虚函数就能顺利地解决这个问题。

6.3.1 虚函数的作用

- 虚函数的使用方法
- (1)在基类用virtual声明成员函数为虚函数。在派生类中重新定义同名函数，让它具有新的功能。
- (2)在派生类中重新定义此函数时，要求函数名、函数类型、参数个数和类型与基类的虚函数相同，根据C++规定，当一个成员函数被声明为虚函数后，其派生类中的同名函数自动成为虚函数。
- (3)定义一个指向基类对象的指针变量，并让它获得同一类族中某个对象的地址。
- (4)用该指针变量调用虚函数，调用的就是该对象所属类的虚函数。

6.3.1 虚函数的作用

- 根据虚函数的要求，程序作一点修改，在student类中声明display函数时，在最左边加一个关键字virtual，即
  virtual void display();

- 把student类的display函数声明为虚函数。程序其他部分不变，再编译后运行程序，请看程序运行结果：

```
num:1001
name:Li
score:87.5

num:2001
name:Wang
score:98.5
pay=563.5
```

- 现在用同一个指针变量不仅输出了基类对象的数据，而且输出了派生类的所有数据说明调用了grad1的display函数。

- 用同一种调用形式pt->display( )，而且pt是一个基类的指针，可以调用同一类族中不同类的虚函数。这就是多态性，对同一消息，不同对象有不同的响应方式。

- 原来，基类指针是用来指向基类对象的，如用它指向派生类对象，系统要进行指针类型转换，所以基类指针指向的是派生类对象中的基类部分。虚函数突破了这个限制，在基类指针指向派生类对象后，就能调用派生类的虚函数。

6.3.1 虚函数的作用

- 函数重载(横向重载)：处理同一个类的同名函数问题，要求函数名相同，但函数的参数(个数、类型、顺序)不能相同；

- 虚函数(纵向重载)：处理类的不同派生层次上的同名函数问题，要求不仅函数名相同，而且函数参数(个数、类型、顺序)也要相同。

- (1) 在基类用virtual声明成员函数为虚函数。在派生类中重新定义同名函数，让它具有新的功能。
- (2) 在派生类中重新定义此函数时，要求函数名、函数类型、参数个数和类型与基类的虚函数相同，根据C++规定，当一个成员函数被声明为虚函数后，其派生类中的同名函数自动成为虚函数。
- (3) 定义一个指向基类对象的指针变量，并让它获得同一类族中某个对象的地址。
- (4) 用该指针变量调用虚函数，调用的就是该对象所属类的虚函数。

6.3.2 静态关联与动态关联

- display加了virtual后在不同类中有不同的作用，呈现了多态。
- 函数重载和用对象名调用虚成员函数，在编译时即可确定其调用的虚函数属于哪个类，其过程称为静态关联。
- 从例6.2里，在调用虚函数时没有指定对象名，系统怎样确定关联呢？编译系统把它放在运行阶段处理，在运行阶段确定关联关系。
- 在运行阶段，基类指针变量先指向了某个类对象，然后通过此指针变量调用该对象中的函数。此时调用哪个对象的函数无疑是确定的。
- 由于在运行中把虚函数和类对象绑定在一起，此过程称为动态关联，这多态性是动态的多态性。
- 在运行阶段，指针可以先后指向不同的类对象，从而调用同一类族中不同类的虚函数。

6.3.3 在什么情况下应当声明虚函数

- 注意事项：
- 只能用virtual声明类的成员函数，把它作为虚函数，而不能将类外的普通函数声明虚函数。因为虚函数的作用是允许派生类中对基类的虚函数重新定义。显然，只能用于类的继承层次结构中。
- 一个成员函数被声明为虚函数后，在同一类族不能再定义一个非virtual的但与该虚函数具有相同参数（个数和类型）和函数返回值类型的的同名函数。

6.3.3 在什么情况下应当声明虚函数

- 根据什么考虑把一个成员函数声明为虚函数?
- (1) 首先看成员函数的类是否会作为基类。然后看在派生类里看它是否会被改变功能，如要改变，一般应该将它声明为虚函数。
- (2) 如果成员函数在类被继承后功能不需要更改，或派生类用不到该函数，则不要声明为虚函数。
- (3) 应考虑对成员函数的访问是通过对象名还是通过基类指针或引用去访问，如果是基类指针或引用，则可以声明为虚函数。
- (4) 有时在定义虚函数时，函数体是空的。具体功能留给派生类去添加。

6.3.4 虚析构函数

- 当派生类的对象撤销时一般先调用派生类的析构函数，然后调用基类的析构函数。
- 如用new运算符建立一个动态对象，如基类中有析构函数，并且定义了一个指向基类的指针变量。
- 在程序中用带指针参数的delete运算符撤销对象时，系统只会执行基类的析构函数，而不执行派生类的析构函数。

6.3.4 虚析构函数

```
class Point {
public:
  Point() {cout << "executing Point
constructor" << endl; }
  ~Point()
  { cout << "executing Point destructor" <<
endl; }};

class Circle : public Point
{public:
  Circle() { cout << "executing Circle
constructor" << endl; }
  ~Circle()
  { cout << "executing Circle destructor" <<
endl; }
private:
  int radus;};
```

```
int main()
{
  Point *p = new Circle;  // Circle &p
  delete p;
  return 0;
}
```

运行结果为

executing Point constructor
executing Circle constructor
executing Point destructor

表示只执行了基类point的析构函数，未执行派生类的析构函数。

6.3.4 虚析构函数

- 这时可将基类的析构函数声明为虚函数，如

```
virtual ~Point()
{
  cout << "executing Point estructor" << endl;
}
```

- 程序其他部分不变，再运行程序，结果为

executing Circle destructor
executing Point destructor

- 先调用派生类的析构函数，再调用基类的析构函数。

```
int main()
{
  Point *p = new Circle;
  delete p;
  return 0;
}
```

运行结果为
executing Point
destructor

表示只执行了基类point的析构函数，未执行派生类的析构函数。

6.3.4 虚析构函数

- 当基类的析构函数为虚函数时无论指针指的是同一类族中的哪一个类对象，撤销对象时，系统会采用动态关联，调用相应的析构函数。

- 如果将基类的析构函数声明为虚函数，由该基类所派生的类的析构函数也都自动成为虚函数。在程序中，最好把基类的析构函数都声明为虚函数。即使不需要析构函数，也可以定义一个函数体为空的析构函数。

- 构造函数不能声明为虚函数。因为构造函数执行时还没有建立类对象，不存在函数与类对象的绑定。

6.4 纯虚函数与抽象类

- 纯虚函数
- 抽象类
- 应用实例

6.4.1 纯虚函数

- 虚函数，有时并不是基类本身的需求，而是考虑到派生类的需要，预留函数名，具体功能由派生类去定义。

- Point / Circle / Cylinder 类： area() 函数
  virtual float area () const {return 0;}

- 返回值为0，表示点没有面积。Point 本身不使用这个函数，且返回值0没有意义。

- 为了简化，可以用纯虚函数。

6.4.1 纯虚函数

- 定义：声明虚函数时被“初始化”为0 的函数。
- 一般格式：virtual 函数类型 函数名(参数表) = 0;
- 无函数体；“=0”不是说返回值为0，只是形式上的作用，告诉编译系统这是个纯虚函数；最后有分号；
- 纯虚函数的作用：在许多情况下，基类中不能为虚函数给出一个有意义的定义，而将它说明为纯虚函数，其作用是：为派生类提供一个一致的接口(界面)。它的定义留给派生类来做，派生类根据需要来定义各自的实现。

6.4.1 纯虚函数

- 注意
- 一个类可以说明一个或多个纯虚函数
- 纯虚函数与函数体为空的虚函数的区别：
  - 纯虚函数—根本没有函数体；所在的抽象类，不能直接进行实例化
  - 纯虚函数只有函数名字而不具备函数功能，不能被调用
  - 空的虚函数—函数体为空，所在的类可以实例化
  - 共同的特点，可以派生出新的类

6.4.2 抽象类

- 定义：带有纯虚函数的类叫抽象类
- 抽象类的主要作用：通过它为一个类族建立一个公共的接口，使它们能够更有效地发挥多态特性
- 抽象类刻画了一组子类的公共操作接口的通用语义，这些接口的语义也传给子类。一般而言，抽象类只描述这组子类共同操作接口，而完整的实现留给子类。
- 一个类层次结构中可以不包含抽象基类，每一个层次都是可以用的，建立对象。但是好的系统，层次结构的顶层或多个层为抽象类。

6.4.2 抽象类的说明

- 抽象类的说明
- 抽象类是一个特殊的类，是为了抽象和设计的目的而建立的，它处于继承层次的结构的较上层，即只能用作其他类的基类，抽象类是不能定义对象的。
- 如果抽象类的派生类对所有纯虚函数进行了定义，则函数可以被调用，派生类也可以定义对象。如果派生类中没有对所有纯虚定义，则仍未抽象类，不能用来定义对象。
- 抽象类不能用作参数类型、函数返回类型或显式转换的类型，但可以说明指向抽象类的指针和引用，此指针可以指向它的派生类，实现多态性。

6.4.3 应用实例

```
class B0 {     //抽象基类B0声明
public:        //外部接口
  virtual void display( ) = 0;    //纯虚函数成员
};

class B1 : public B0 {   //公有派生
public:
  void display()
  { cout << "B1::display()" << endl; }  //虚成员函数
};

class D1: public B1 {  //公有派生
public:
  void display()
  { cout << "D1::display()" << endl; }  //虚成员函数
};

void fun(B0 *ptr)      //普通函数
{ ptr->display(); }

int main() {               //主函数
  B0 *p;    //声明抽象基类指针
  B1 b1;    //声明派生类对象
  D1 d1;    //声明派生类对象
  p = &b1;
  fun(p);   //调用派生类B1函数成员
  p = &d1;
  fun(p);   //调用派生类D1函数成员
  return 0;
}

// 运行结果:
B1::display()
D1::display()
```

6.4.3 应用实例

例6.4 抽象基类的应用。

```
class Shape
{
public:
  virtual float area() const { return 0.0; }
            // 虚函数
            // 非纯虚函数，point类可以不再对area()重新定义
  virtual float volume() const { return 0.0; }
            // 虚函数
            //非纯虚函数，point类可以不再对area()重新定义
  virtual void shapeName() const =0;
            // 纯虚函数
            // 与派生类密切相关，不应该在基类定义，而在派生类定义
};
```

6.4.3 应用实例

```
class Point : public Shape
// Point是Shape的公用派生类
{
protected:
  float x,y;
public:
  Point(float=0,float=0);
  void setPoint(float,float);
  float getX() const {return x;}
  float getY() const {return y;}
  // 对纯虚函数进行定义
  virtual void shapeName() const { cout << "Point:"; }

  friend ostream & operator << (ostream &, const Point &);
};
```

```
Point::Point(float a, float b)
{ x = a; y = b; }

void Point::setPoint(float a, float b)
{ x = a; y = b; }

ostream & operator<<(ostream &output, const Point &p)
{
  output << "[" << p.x << "," << p.y << "]";
  return output;}
```

6.4.3 应用实例

```
class Circle : public Point
 //    声明Circle类
{
protected:
  float radius;
public:
  Circle(float x = 0, float y = 0, float r = 0);
  void setRadius(float);
  float getRadius() const;
  virtual float area() const;
  // 对纯虚函数进行再定义
  virtual void shapeName() const { cout
<< "Circle:"; }
  friend ostream &operator<<(ostream &,
const Circle &);
};
```

```
Circle::Circle(float a, float b, float r) : Point(a, b),
radius(r) {}

void Circle::setRadius(float r)
{ radius = r; }

float Circle::getRadius() const { return radius; }

float Circle::area() const
{ return 3.14159 * radius * radius; }

ostream &operator<<(ostream &output,const Circle
&c)
{
  output << "[" << c.x << "," << c.y << "], r=" <<
c.radius;
  return output;}
```

6.4.3 应用实例

```
//    声明Cylinder类
class Cylinder : public Circle
{
public:
  Cylinder (float x = 0, float y = 0,float r = 0,float h = 0);
  void setHeight(float);
  float getHeight() const;
  virtual float area() const;
  virtual float volume() const;
  //    对纯虚函数进行再定义
  virtual void shapeName() const { cout << "Cylinder:"; }
  friend ostream& operator<<(ostream&, const Cylinder&);
protected:
  float height;
};
```

6.4.3 应用实例

```
Cylinder::Cylinder(float a, float b, float r, float h): Circle(a, b, r), height(h) { }

void Cylinder::setHeight(float h) { height = h; }

float Cylinder::getHeight() const { return height; }

float Cylinder::area() const
{ return 2 * Circle::area() + 2 * 3.14159 * radius * height; }

float Cylinder::volume() const
{ return Circle::area() * height; }

ostream &operator<<(ostream &output, const Cylinder& cy)
{
  output << "[" << cy.x << "," << cy.y << "], r=" << cy.radius << ", h=" << cy.height;
  return output;
}
```

6.4.3 应用实例

```
int main()
{
  Point point(3.2, 4.5);                  // 建立Point类对象point
  Circle circle(2.4, 12, 5.6);          // 建立Circle类对象circle
  Cylinder cylinder(3.5, 6.4, 5.2, 10.5);
  // 建立Cylinder类对象cylinder
  point.shapeName();                              // 静态关联
  cout<<point<<endl;

  circle.shapeName();                            // 静态关联
  cout<<circle<<endl;

  cylinder.shapeName();                          // 静态关联
  cout<<cylinder<<endl<<endl;

  Shape *pt;                                      // 定义基类指针
```

6.4.3 应用实例

```
  pt = &point;                          // 指针指向Point类对象
  pt->shapeName();                      // 动态关联
  cout << "x=" << point.getX() << ",y=" << point.getY() << "\narea=" << pt->area() <<
"\nvolume=" << pt->volume()<<"\n\n";

  pt = &circle;                         // 指针指向Circle类对象
  pt->shapeName();                      // 动态关联
  cout << "x=" << circle.getX() << ",y=" << circle.getY() << "\narea=" << pt->area() <<
"\nvolume=" << pt->volume() << "\n\n";

  pt = &cylinder;                       // 指针指向Cylinder类对象
  pt->shapeName();                      // 动态关联
  cout << "x=" << cylinder.getX() << ",y=" << cylinder.getY() << "\narea=" << pt->area()
<< "\nvolume=" << pt->volume() << "\n\n";
  return 0;
}
```

6.4.3 应用实例 — 结论

- 1、一个基类如果包含一个或者多个纯虚函数，就是抽象基类，不能也不必要定义对象；
- 2、抽象基类与普通基类不同，一般不是现在存在对象的抽象，可以没有任何物理的或者其他实际意义。
- 3、类的层次结构中，顶层或者最上面的几层可以为抽象基类。抽象基类体现本类族中各类的共性，把各类中共有的成员函数集中在抽象基类中声明。
- 4、抽象基类是本类族的公共接口。同一基类派生出的多个类有同一接口。
- 5、区别静态关联和动态关联。通过对象名调用，编译阶段确定调用哪个类的虚函数，属于静态关联；通过基类指针调用，运行时才能确定，为动态关联。
- 6、如果在基类中声明了虚函数，则在派生类中凡是与该函数有相同函数名、函数类型、参数个数和类型的函数，均为虚函数。

面向对象程序设计

Object Oriented Programming

2021

谢谢大家