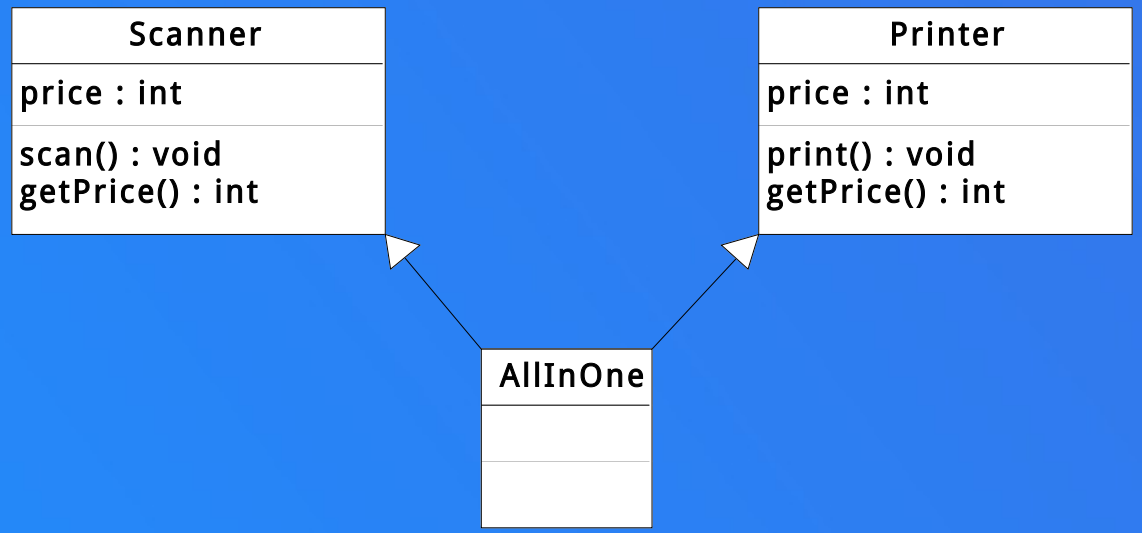
Module03-11
C++ 面向对象编程：类层次结构

面向对象编程 - 类层次结构

- 类层次结构 (Classes Hierarchies)：
  - 多继承 (Multiple Inheritance)
  - 访问控制 (Access Control)
  - 运行期类型识别 (Runtime Type Identification)
  - 指向成员的指针 (Pointers to Members)

类层次结构 - 多继承

- 关于多继承
  - C++ 语言中允许一个类继承与多个直接基类
  - 如果一个类继承于一个以上的直接基类，则称为“多继承”
  - 如果一个类仅仅继承于一个直接基类，则称为“单继承”
- 多继承图例：



类层次结构 - 多继承

- 多继承示例代码

```
class Scanner {
    int price;
    // ...
public:
    void scan();
    int getPrice() const;
    // ...
};

class Printer {
    int price;
    // ...
public:
    void print();
    int getPrice() const;
    // ...
};

// 类AllInOne 继承于2个基类 Scanner 和 Printer
class AllInOne: public Scanner, public Printer {
    // ...
};
```

类层次结构 - 多继承

- 多继承示例代码（续）

```
void f1(Printer* p, Scanner* s) {
    p->print();
    s->scan();
}

int main() {
    AllInOne a, b;
    f1(&a, &b); // OK, AllInOne既是Printer、也是Scanner
}
```

类层次结构 - 多继承

- 解决二义性
  - 解决多个基类中相同名字的成员的二义性：
    - 明确指定基类，如 Printer::getPrice()

```
AllInOne a;
a.getPrice();   // Error, 'getPrice' is ambiguous
a.Printer::getPrice(); // ok
```

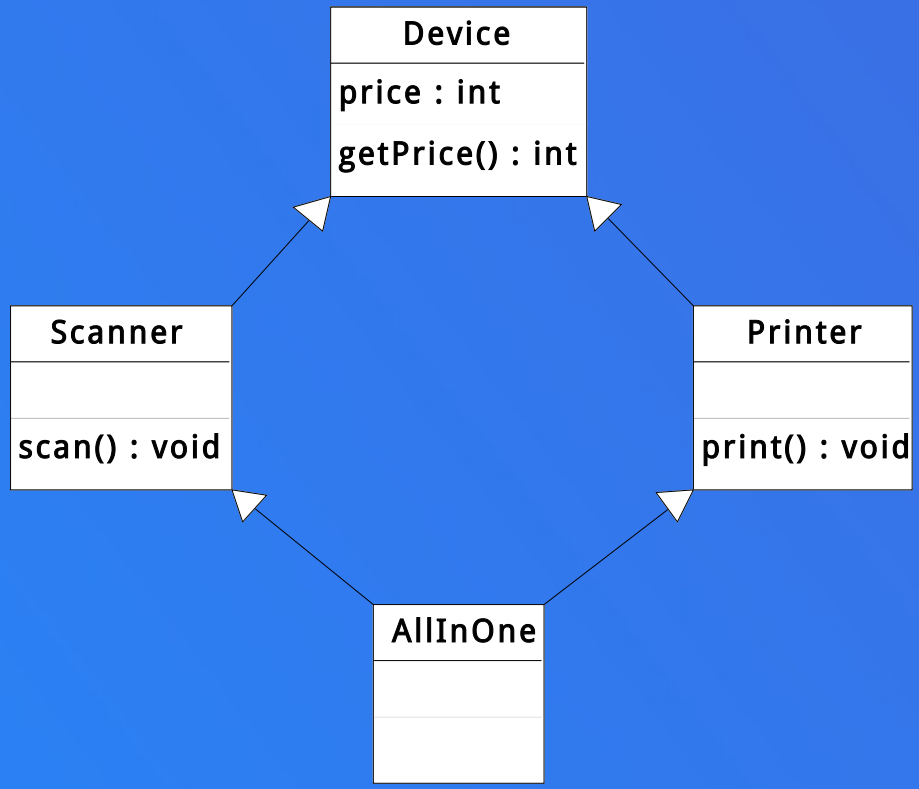
    - 在子类的声明中使用 using 指令

```
class AllInOne: public Scanner, public Printer {
public:
    using Printer::getPrice;
    // ...
};

void f() {
    AllInOne a;
    a.getPrice(); // OK, call Printer::getPrice()
}
```

类层次结构 - 多继承

- 虚基类 (Virtual Base Classes)
  - AllInOne 类体系的一种可能的布局（如右图）

```
class Device { /* ... */};
class Scanner: public Device
{ /* ... */ };

class Printer: public Device
{ /* ... */ };

class AllInOne: public Scanner,
public Printer { /* ... */ };

void f3() {
    AllInOne a;
    a.getPrice();  // Error
    a.Device::getPrice(); //
Error
}
```



类层次结构 - 多继承

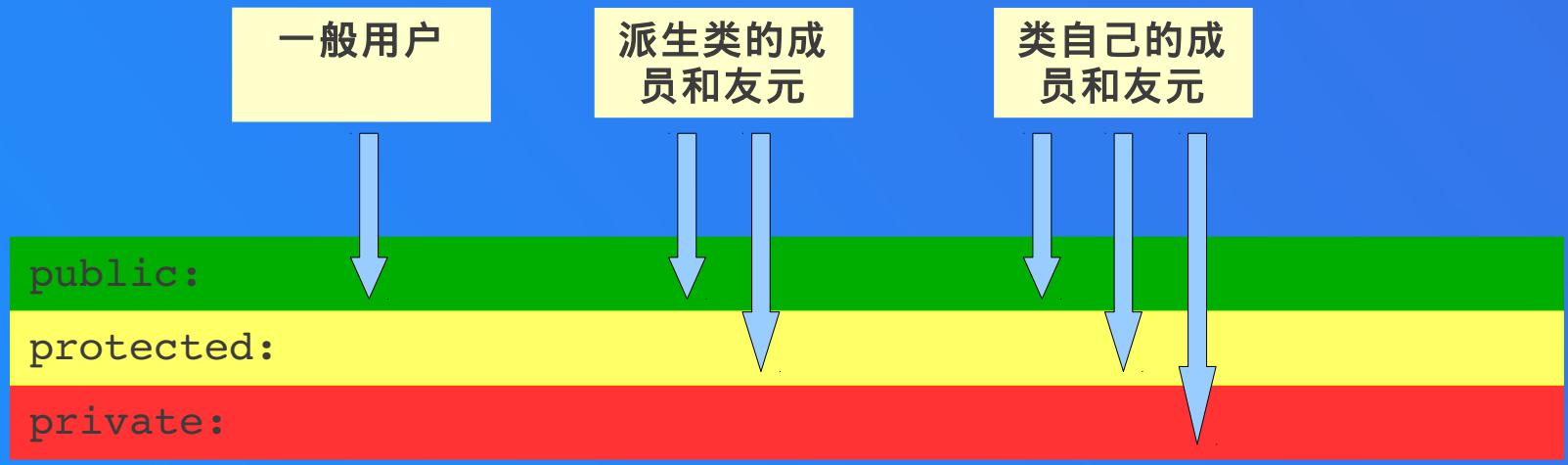
- 虚基类 (Virtual Base Classes)（续）
  - 将基类 Device 作为虚基类，即让类 Printer 和类 Scanner 以虚继承的方式继承于 Device
  - 虚基类是这么一种机制：虚 基类在的任何派生类中总是用同一个（共享的）对象表示

```
class Device { /* ... */};
class Scanner: public virtual Device { /* ... */ };
class Printer: public virtual Device { /* ... */ };
class AllInOne: public Scanner, public Printer { /* ... */ };

void f3() {
    AllInOne a;
    a.getPrice();  // OK
}
```

试试看，采用虚继承机制前后的 AllInOne 类型的 size ！

类层次结构 - 访问控制

- 访问类成员
  - 类中的一个成员可以是 private 、 protected 或 public 的：
    - 如果成员是 private ， 其名字将只能由其声明所在的类的成员函数和友元使用
    - 如果成员是 protected ， 其名字只能由其声明所在的类的成员函数和友元，以及该类的各级派生类的成员函数和友元使用
    - 如果一个成员是 public ， 其名字可以由任何函数使用
  - 类成员访问控制图例：



类层次结构 - 访问控制

- 访问基类
  - 如同类的成员，基类也可以是 private 、 protected 或 public：

```
class B { /* ... */ };
class X : public B { /* ... */ }; // public继承
class Y : protected B { /* ... */ }; // protected继承
class Z : private B { /* ... */ }; // private继承
```

  - 如果没有指明对基类的访问描述：
    - 对于 class 而言，其基类默认为 private
    - 对于 struct 而言，其基类默认为 public

```
class X: B { /* ... */ }; // B 是private 基类
struct Y: B { /* ... */ }; // B 是public 基类
```

类层次结构 - 访问控制

- 访问基类（续）
  - 对于基类的访问描述符控制着对基类成员的访问，以及从派生类型到基类类型的指针、引用转换，如冲基类 B 派生类 D：
    - 如果 B 是 private 基类，则其 public 和 protected 成员只能由 D 的成员函数和友元访问；只有 D 的成员和友元能将 D* 转换到 B*
    - 如果 B 是 protected 基类，则其 public 和 protected 成员只能由 D 的成员函数和友元、以及 D 的派生类的成员函数和友元访问；只有 D 的成员和友元以及 D 的派生类的成员函数和友元能将 D* 转换成 B*
    - 如果 B 是 public 基类，则其 public 成员可以由任何函数使用，除此之外，其 protected 成员还能由 D 的成员函数和友元，以及 D 的派生类的成员函数和友元访问；任何函数都可将 D* 转换到 B*

类层次结构 - 访问控制

- 访问基类（续）
  - 示例 1：通过任意函数访问基类

```
class X {
public:
    void f1();
    // ...
};
class Y1: public X { };
class Y2: protected X { };
class Y3: private X { };

void f(Y1* py1, Y2* py2, Y3* py3) {
    X* px = py1; // OK, X是Y1的public基类
    py1->f1(); // OK

     px = py2; // Error, X是Y2的protected基类
    py2->f1(); // Error

    X* px = py3; // Error, X是Y3的private基类
    py3->f1(); // Error
}
```

类层次结构 - 访问控制

- 访问基类（续）

```
class X {
public:
    void f1();
    // ...
};
class Y1: public X { };
class Y2: protected X { };
class Y3: private X { };
```

  - 示例 1：通过任意函数访问 public 基类 X

```
void f(Y1* py1, Y2* py2, Y3* py3) {
    X* px = py1; // OK, X是Y1的public基类
    py1->f1(); // OK

     px = py2; // Error, X是Y2的protected基类
    py2->f1(); // Error

    X* px = py3; // Error, X是Y3的private基类
    py3->f1(); // Error
}
```

类层次结构 - 访问控制

- 访问基类（续）
  - 示例 2：通过派生类的成员函数访问 protected 基类 X

```
class Z2 : public Y2 { void f(Y1* py1, Y2* py2, Y3* py3); };

void Z2::f(Y1* py1, Y2* py2, Y3* py3) {
    X* px = py1; // OK, X是Y1的public基类
    py1->f1(); // OK

    px = py2; // OK, X是Y2的protected基类且Z2是Y2的派生类
    py2->f1(); // OK

    X* px = py3; // Error, X是Y3的private基类
    py3->f1(); // Error
}
```

类层次结构 - 访问控制

- 访问基类（续）
  - 示例 3：通过派生类的成员函数访问 private 基类 X

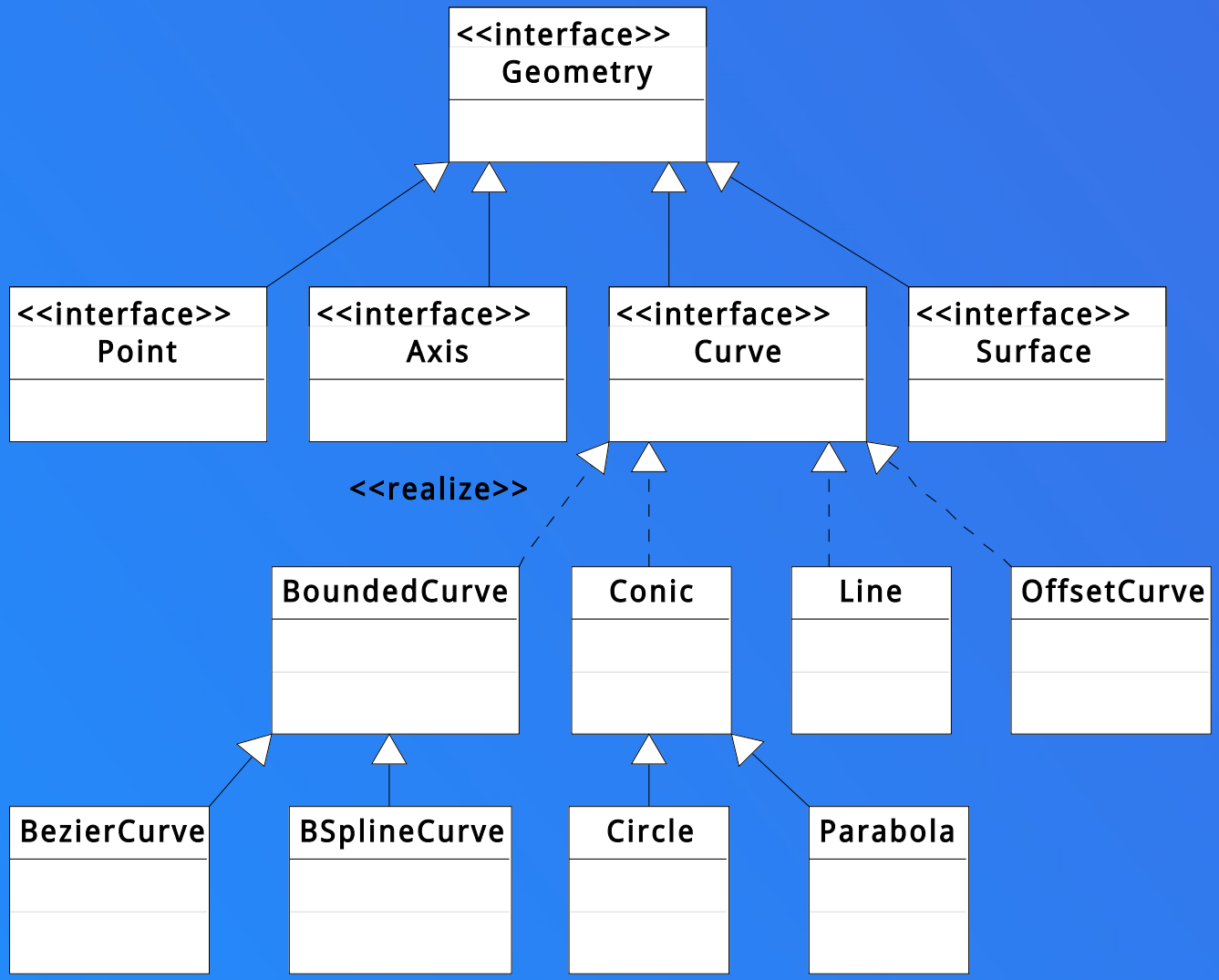
```
class Y3 : public X { void f(Y1* py1, Y2* py2, Y3* py3); };

void Y3::f(Y1* py1, Y2* py2, Y3* py3) {
    X* px = py1; // OK, X是Y1的public基类
    py1->f1(); // OK

    px = py2; // Error, X是Y2的protected基类
    py2->f1(); // Error

    X* px = py3; // OK, X是Y3的private基类但Y3::f()是Y3的成员
    py3->f1(); // OK
}
```

类层次结构 - 运行期类型识别

- RTTI (Run Time Type Information)
- cast
  - Down Cast： 从基类向派生类 cast
  - Up Cast： 从派生类向基类 cast
  - Cross Cast： 从一个基类向其兄弟类的 cast

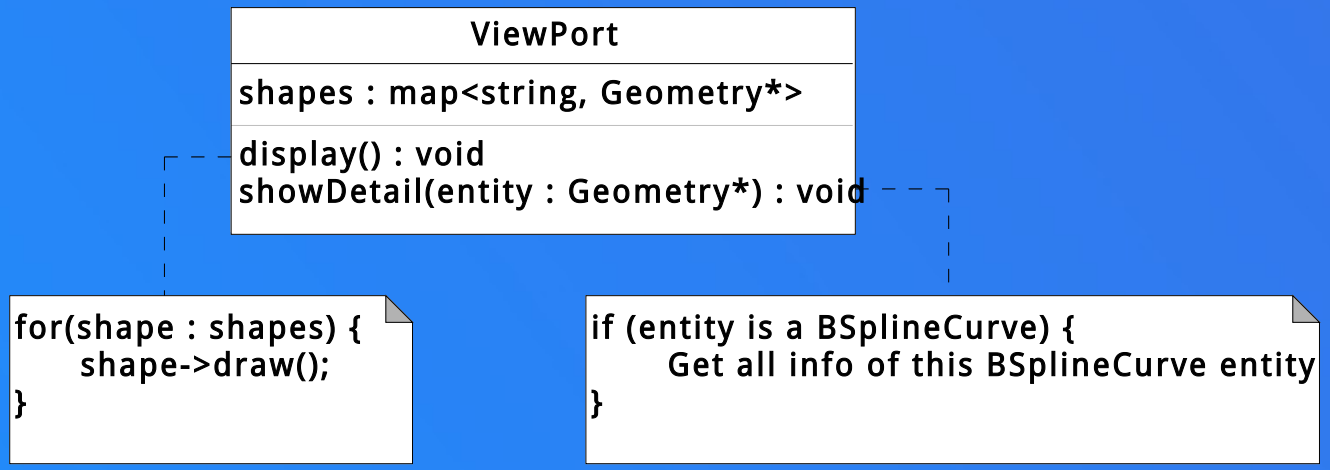
类层次结构 - 运行期类型识别

- 一个简单的类继承体系



类层次结构 - 运行期类型识别

- dynamic_cast
  - dynamic_cast 操作符通常用来查找对象的实际类型
  - 考虑下面的情形：
    - 尽管很多情况下要针对接口（基类）编程，如 ViewPort 的 display()
    - 但某些时候还是要知道某个对象的确切类型，如 ViewPort 的 showDetail()



类层次结构 - 运行期类型识别

- dynamic_cast（续）
  - dynamic_cast 操作符使用：

    ```
    dynamic_cast<T*>(ptr_of_base_type)
    ```

  - dynamic_cast 运算的结果：
    - 非 0 值表示转换成功，0（空指针）表示转换不成功
  - 示例：

```
void ViewPort::getDetail(Geometry* entity) {
    BSplineCurve* bspl = dynamic_cast<BSplineCurve*>(entity);
    if (bspl != 0) { // entity's type is BSplineCurve*
        // get detail info of bspl or do something ...
    } else {
        // Do nothing ...
    }
}
```

类层次结构 - 运行期类型识别

- dynamic_cast（续）
  - 使用 dynamic_cast 需注意：
    1. 不能违背对 private 和 protected 基类的保护：

```
class D: public B1, protected B2 { /*...*/ };
void f(D* p) {
    B1* e1 = p;
    B1* n1 = dynamic_cast<B1*> (p); // OK

    B2* e3 = p;  // Error
    B2* n3 = dynamic_cast<B2*> (p); // Error
}
```

类层次结构 - 运行期类型识别

- dynamic_cast（续）
  - 使用 dynamic_cast 需注意：
    2. dynamic_cast 要求一个到多态类型的指针或引用，以便完成 Down Cast 或 Cross Cast，但并不要求目标类型一定是多态的
    3. dynamic_cast 可以将一个具有多态类型的对象引用，转换成目标类型的对象引用，但没有办法检查类型转换的正确性，如果对引用的 dynamic_cast 的操作对象不是所需的类型（目标类型），则抛出 bad_cast 异常

类层次结构 - 运行期类型识别

- typeid
  - 可以用 dynamic_cast 操作符获取对象的类型，但并不总是获取对象的确切类型
  - 在需要确切类型信息的时候，可以使用 typeid 操作符：

    typeid(object)
  - typeid 返回 type_info 对象的引用，可以通过 type_info 对象的 name() 成员函数查看对象的类型信息， name() 函数返回 const char* 类型的字符串
  - 如果一个多态类型的指针或引用的操作对象的值是 0， typeid() 将抛出 bad_typeid 异常
  - 示例：

```
void f(D* p) {
    cout << typeid(*p).name() << endl;
}
```

类层次结构 - 指向成员的指针

- 指向成员的指针的类型
  - 数据成员指针的类型：
    - 如成员 lb 的指针类型为：

      ```
      int Rectangle::*
      ```

    - 可以定义指针，并赋值

      ```
      int Rectangle::* np;
      np = &Rectangle::lb;
      ```

  - 成员函数指针的类型：
    - 如成员函数 move() 的指针类型：

      ```
      int (Rectangle::*) ()
      ```

    - 可以定义指针，并赋值

      ```
      int (Rectanle::* func) () = 0;
      func = &Rectangle::move;
      ```

```
class Rectangle {
    int lb, rt;
public:
    void move();
    void rotate();
    void mirror();

    typedef void (Rectangle::*Op)();
    void repeat(Op op, const int&);
};
```

类层次结构 - 指向成员的指针

- 使用指向成员的指针
  - 示例

```
void Rectangle::repeat(Op op, const int& times) {
    for (int i = 0; i < times; ++i) {
        (this->*op)();  // 不要忘记(this->*op)前后的括号！
    }
}

int main() {
    Rectangle r;
    r.repeat(&Rectangle::move, 3);
}
```

类层次结构 - 指向成员的指针

- 静态成员的指针
  - 由于类的静态成员不专属于某个对象，所以指向静态成员的指针是普通指针。

类层次结构 - Bjarne's Advices

- Bjarne's Advices
  - 利用常规的多继承表述特征的合并
  - 利用多继承完成实现细节与接口分离
  - 用虚基类表达在类层次结构里对某些类共同的东西
  - 避免显式转换
  - 在必须漫游类层次结构的场合，使用 dynamic_cast
  - 尽量使用 dynamic_cast 而不是 typeid
  - 尽量使用 private 而不是 protected，不要声明 protected 数据成员
  - 如果某个类定义了 operator delete()，也应该为其提供一个虚析构函数
  - 在构造和析构期间不要调用虚函数