5 多态

5.1 非虚的世界

- 对象的自恰性：对于同样的函数调用，每个对象会做出自己恰当的响应

```cpp
// selfconst.cpp

// 对于同样的函数调用，每个对象会做出自己恰当的响应

#include <iostream>

using namespace std;

class Shape {
public:
    Shape(double x, double y) : x(x), y(y) {}

    void draw(void) const {
        cout << "Shape(" << x << ',' << y << ')' << endl;
    }

protected:
    double x, y;
};

class Rect : public Shape {
public:
    Rect(double x, double y, double w, double h) : Shape(x, y), w(w), h(h) {}

    void draw(void) const {
        cout << "Rect(" << x << ',' << y << ',' << w << ',' << h << ')' << endl;
    }

private:
    double w, h;
};

class Circle : public Shape {
public:
    Circle(double x, double y, double r) : Shape(x, y), r(r) {}

    void draw(void) const {
        cout << "Circle(" << x << ',' << y << ',' << r << ')' << endl;
    }

private:
    double r;
};

int main(void) {
    Rect rect(1, 2, 3, 4);
```

```
    rect.draw();

    Circle circle(5, 6, 7);
    circle.draw();

    return 0;
}
```

- 指针游戏
  - 通过指向子类对象的基类指针调用函数

```
// bdb.cpp

// 通过指向子类对象的基类指针调用函数

#include <iostream>

using namespace std;

class Shape {
public:
    Shape(double x, double y) : x(x), y(y) {}

    void draw(void) const {
        cout << "Shape(" << x << ',' << y << ')' << endl;
    }

protected:
    double x, y;
};

class Rect : public Shape {
public:
    Rect(double x, double y, double w, double h) : Shape(x, y), w(w), h(h) {}

    void draw(void) const {
        cout << "Rect(" << x << ',' << y << ',' << w << ',' << h << ')' << endl;
    }

private:
    double w, h;
};

class Circle : public Shape {
public:
    Circle(double x, double y, double r) : Shape(x, y), r(r) {}

    void draw(void) const {
        cout << "Circle(" << x << ',' << y << ',' << r << ')' << endl;
    }

private:
```

```
    double r;
};

int main(void) {
    // 通过基类类型的指针，只能调用基类的成员函数，
    // 虽然该指针实际指向的，是一个子类类型的对象

    Rect rect(1, 2, 3, 4);
    Shape* pshape = &rect;
    pshape->draw();

    // 通过基类类型的引用，只能调用基类的成员函数，
    // 虽然该引用实际引用的，是一个子类类型的对象

    Circle circle(5, 6, 7);
    Shape& rshape = circle;
    rshape.draw();

    return 0;
}
```

- 通过指向基类对象的子类指针调用函数

```
// dbd.cpp

// 通过指向基类对象的子类指针调用函数

#include <iostream>

using namespace std;

class Shape {
public:
    Shape(double x, double y) : x(x), y(y) {}

    void draw(void) const {
        cout << "Shape(" << x << ',' << y << ')' << endl;
    }

protected:
    double x, y;
};

class Rect : public Shape {
public:
    Rect(double x, double y, double w, double h) : Shape(x, y),
w(w), h(h) {}

    void draw(void) const {
        cout << "Rect(" << x << ',' << y << ',' << w << ',' << h <<
')' << endl;
    }

private:
    double w, h;
};
```

```
32
33  class Circle : public Shape {
34  public:
35      Circle(double x, double y, double r) : Shape(x, y), r(r) {}
36
37      void draw(void) const {
38          cout << "Circle(" << x << ',' << y << ',' << r << ')' <<
    endl;
39      }
40
41  private:
42      double r;
43  };
44
45  int main(void) {
46      // 通过指向基类对象的子类指针，虽然可以调用子类的成员函数，
47      // 但凡是涉及到对子类特有部分的访问，其结果都是不可预知的
48
49      Shape shape1(1, 2);
50      Rect* rect = static_cast<Rect*>(&shape1);
51      rect->draw();
52
53      // 通过引用基类对象的子类引用，虽然可以调用子类的成员函数，
54      // 但凡是涉及到对子类特有部分的访问，其结果都是不可预知的
55
56      Shape shape2(5, 6);
57      Circle& circle = static_cast<Circle&>(shape2);
58      circle.draw();
59
60      return 0;
61  }
```

- 通过指向子类对象的基类指针调用子类的函数

```
1   // bdd.cpp
2
3   // 通过指向子类对象的基类指针调用子类的函数
4
5   #include <iostream>
6
7   using namespace std;
8
9   class Shape {
10  public:
11      Shape(double x, double y) : x(x), y(y) {}
12
13      void draw(void) const {
14          cout << "Shape(" << x << ',' << y << ')' << endl;
15      }
16
17  protected:
18      double x, y;
19  };
20
21  class Rect : public Shape {
22  public:
```

```
    Rect(double x, double y, double w, double h) : Shape(x, y), w(w), h(h) {}

    void draw(void) const {
        cout << "Rect(" << x << ',' << y << ',' << w << ',' << h << ')' << endl;
    }

    double area(void) const {
        return w * h;
    }

private:
    double w, h;
};

class Circle : public Shape {
public:
    Circle(double x, double y, double r) : Shape(x, y), r(r) {}

    void draw(void) const {
        cout << "Circle(" << x << ',' << y << ',' << r << ')' << endl;
    }

    double area(void) const {
        return 3.14159 * r * r;
    }

private:
    double r;
};

int main(void) {
    // 通过指向子类对象的基类指针，无法调用子类的成员函数，
    // 限制其能力的是它本身的类型，而非其所指向对象的类型，

    Rect rect(1, 2, 3, 4);
    Shape* pshape = &rect;
    // cout << pshape->area() << endl; // 编译错误
    cout << static_cast<Rect*>(pshape)->area() << endl; // 向下造型

    // 通过引用子类对象的基类引用，无法调用子类的成员函数，
    // 限制其能力的是它本身的类型，而非其所引用对象的类型

    Circle circle(5, 6, 7);
    Shape& rshape = circle;
    // cout << rshape.area() << endl; // 编译错误
    cout << static_cast<Circle&>(rshape).area() << endl; // 向下造型

    return 0;
}
```

5.2 虚函数与多态

5.2.1 通过指向子类对象的基类指针调用虚函数

基于虚函数的运行时多态：

- 基类中带有virtual关键字的成员函数为虚函数
- 子类中的某个成员函数，如果与基类中的虚函数拥有完全相同的函数原型，那么该成员函数也是一个虚函数，并对基类中的版本构成覆盖
- 通过一个指向子类对象的基类指针，或一个引用子类对象的基类引用，调用基类中的虚函数，实际被执行的是子类中的覆盖版本，而非基类中的原始版本
- 这种使同一种类型表现出多种行为特征的语言特性，称为多态性

```
// bdv.cpp

// 通过指向子类对象的基类指针调用虚函数

#include <iostream>

using namespace std;

class Shape {
public:
    Shape(double x, double y) : x(x), y(y) {}

    // 基类中带有virtual关键字的成员函数为虚函数

    virtual void draw(void) const {
        cout << "Shape(" << x << ',' << y << ')' << endl;
    }

protected:
    double x, y;
};

class Rect : public Shape {
public:
    Rect(double x, double y, double w, double h) : Shape(x, y), w(w), h(h)
{}

    // 子类中的某个成员函数，如果与基类中的虚函数拥有完全相同的函数
    // 原型，那么该成员函数也是一个虚函数，并对基类中的版本构成覆盖

    void draw(void) const {
        cout << "Rect(" << x << ',' << y << ',' << w << ',' << h << ')' <<
endl;
    }

private:
    double w, h;
};

class Circle : public Shape {
public:
```

```
    Circle(double x, double y, double r) : Shape(x, y), r(r) {}

    // 子类中的某个成员函数，如果与基类中的虚函数拥有完全相同的函数
    // 原型，那么该成员函数也是一个虚函数，并对基类中的版本构成覆盖

    void draw(void) const {
        cout << "Circle(" << x << ',' << y << ',' << r << ')' << endl;
    }

private:
    double r;
};

void drawShape(const Shape* shape) {
    shape->draw(); // 同一种类型表现出多种行为特征，谓之多态
}

void drawShapes(const Shape* shapes[]) {
    for (size_t i = 0; shapes[i]; ++i)
        shapes[i]->draw(); // 同一种类型表现出多种行为特征，谓之多态
}

int main(void) {
    // 通过一个指向子类对象的基类指针，调用基类中的虚函数，
    // 实际被执行的是子类中的覆盖版本，而非基类中的原始版本

    Rect rect(1, 2, 3, 4);
    Shape* pshape = &rect;
    pshape->draw();

    // 通过一个引用子类对象的基类引用，调用基类中的虚函数，
    // 实际被执行的是子类中的覆盖版本，而非基类中的原始版本

    Circle circle(5, 6, 7);
    Shape& rshape = circle;
    rshape.draw();

    drawShape(&rect);
    drawShape(&circle);

    const Shape* shapes[] {&rect, &circle, NULL};
    drawShapes(shapes);

    return 0;
}
```

5.2.2 虚函数覆盖的条件

对基类中的虚函数，子类可以用一个具有相同签名（即函数名相同，参数表相同，常限定相同）的成员函数进行覆盖。为使覆盖有效需要满足以下条件：

1. 如果基类中的虚函数返回一个基本类型的数据，那么该函数在子类中的覆盖版本也必须返回相同类型的数据
2. 如果基类中的虚函数返回一个类类型的指针或引用，那么该函数在子类中的覆盖版本可以返回其子类类型的指针或引用——类型协变

3. 无论基类中的虚函数位于公有、保护还是私有部分，该函数在子类中的覆盖版本都可以出现在包括公有、保护及私有在内的任何部分

```cpp
// override.cpp

// 虚函数覆盖需要满足的条件

#include <iostream>

using namespace std;

class X {};
class Y : public X {};

class A {
public:
    virtual int foo(int) {
        cout << "A::foo() invoked" << endl;
        return 0;
    }

    virtual X* bar(int) {
        cout << "A::bar() invoked" << endl;
        return NULL;
    }

    virtual int hum(int) {
        cout << "A::hum() invoked" << endl;
        return 0;
    }
};

class B : public A
{
public:
    // 不构成覆盖，常限定不同
    int foo(int) const {
        cout << "B::foo(1) invoked" << endl;
        return 0;
    }

    // 不构成覆盖，参数表不同
    int foo(int, bool) {
        cout << "B::foo(2) invoked" << endl;
        return 0;
    }

    // 编译错误，返回类型不同
    // bool foo(int) {
    //     cout << "B::foo(3) invoked" << endl;
    //     return true;
    // }

    // 构成覆盖，返回类型是指针或引用，且为基类版本返回类型的子类
    Y* bar(int) {
```

```
        cout << "B::bar() invoked" << endl;
        return NULL;
    }

private:
    // 构成覆盖，访问控制属性与虚函数覆盖无关
    int hum(int) {
        cout << "B::hum() invoked" << endl;
        return 0;
    }
};

int main(void) {
    B b;
    A* a = &b;

    a->foo(0);
    a->bar(0);
    a->hum(0);

    return 0;
}
```

5.2.3 多态性的条件

5.2.3.1 多态性必须借助指针或引用才能表现出来

```
// ptref.cpp

// 多态性必须借助指针或引用才能表现出来

#include <iostream>

using namespace std;

class Shape {
public:
    Shape(double x, double y) : x(x), y(y) {}

    virtual void draw(void) const {
        cout << "Shape(" << x << ',' << y << ')' << endl;
    }

protected:
    double x, y;
};

class Rect : public Shape {
public:
    Rect(double x, double y, double w, double h) : Shape(x, y), w(w), h(h)
{}

    void draw(void) const {
        cout << "Rect(" << x << ',' << y << ',' << w << ',' << h << ')' <<
endl;
```

```
    }

private:
    double w, h;
};

int main(void) {
    Rect rect(1, 2, 3, 4);

    // 通过指针调用虚函数--多态
    Shape* pshape = &rect;
    pshape->draw();

    // 通过引用调用虚函数--多态
    Shape& rshape = rect;
    rshape.draw();

    // 通过对象调用虚函数--非多态
    Shape shape = rect; // 对象截切
    shape.draw();

    return 0;
}
```

5.2.3.2 子类调基类靠继承，基类调子类靠多态

```
// this.cpp

// 子类调基类靠继承，基类调子类靠多态

#include <iostream>

using namespace std;

class Base {
public:
    Base(void) {
        // 在构造函数中调用虚函数没有多态性，因为这时子类对象
        // 的特有部分尚未初始化，调用子类的覆盖版本是有风险的

        bar();
    }

    ~Base(void) {
        // 在析构函数中调用虚函数没有多态性，因为这时子类对象
        // 的特有部分已经被析构，调用子类的覆盖版本是有风险的

        bar();
    }

    void foo(void) const {
        // 虽然没有显式地通过指向子类对象的基类指针，或引用子
        // 类对象的基类引用，调用基类中的虚函数，但实际上是通
        // 过this指针完成的函数调用，而this指针就是一个指向子
        // 类对象的基类指针，表现出多态性，即基类调子类靠多态
```

```
30
31          bar(); // this->bar();
32      }
33
34      virtual void bar(void) const {
35          cout << "Base::bar()" << endl;
36      }
37  };
38
39  class Derived : public Base {
40  public:
41      void bar(void) const {
42          cout << "Derived::bar()" << endl;
43      }
44
45      void hum(void) const {
46          foo(); // 子类调基类靠继承
47      }
48  };
49
50  int main (void) {
51      Derived d;
52
53      d.hum();
54
55      return 0;
56  }
```

5.2.4 操作符函数的多态性

- 全局函数形式的操作符函数，无法通过虚函数展现其多态性
- 成员函数形式的操作符函数，和其它成员函数一样，可被声明为虚函数，并表现出多态性

```
1   // virop.cpp
2
3   // 操作符函数的多态性
4
5   #include <iostream>
6
7   using namespace std;
8
9   class Shape {
10  public:
11      Shape(double x, double y) : x(x), y(y) {}
12
13      virtual ostream& operator>>(ostream& os) const {
14          return os << "Shape(" << x << ',' << y << ')';
15      }
16
17  protected:
18      double x, y;
19  };
20
21  class Rect : public Shape {
```

```
public:
    Rect(double x, double y, double w, double h) : Shape(x, y), w(w), h(h) {}

    ostream& operator>>(ostream& os) const {
        return os << "Rect(" << x << ',' << y << ',' << w << ',' << h << ')';
    }

private:
    double w, h;
};

class Circle : public Shape {
public:
    Circle(double x, double y, double r) : Shape(x, y), r(r) {}

    ostream& operator>>(ostream& os) const {
        return os << "Circle(" << x << ',' << y << ',' << r << ')';
    }

private:
    double r;
};

int main(void) {
    Rect rect(1, 2, 3, 4);
    Shape* pshape = &rect;
    *pshape >> cout << endl;

    Circle circle(5, 6, 7);
    Shape& rshape = circle;
    rshape >> cout << endl;

    return 0;
}
```

5.2.5 纯虚函数、抽象类和纯抽象类

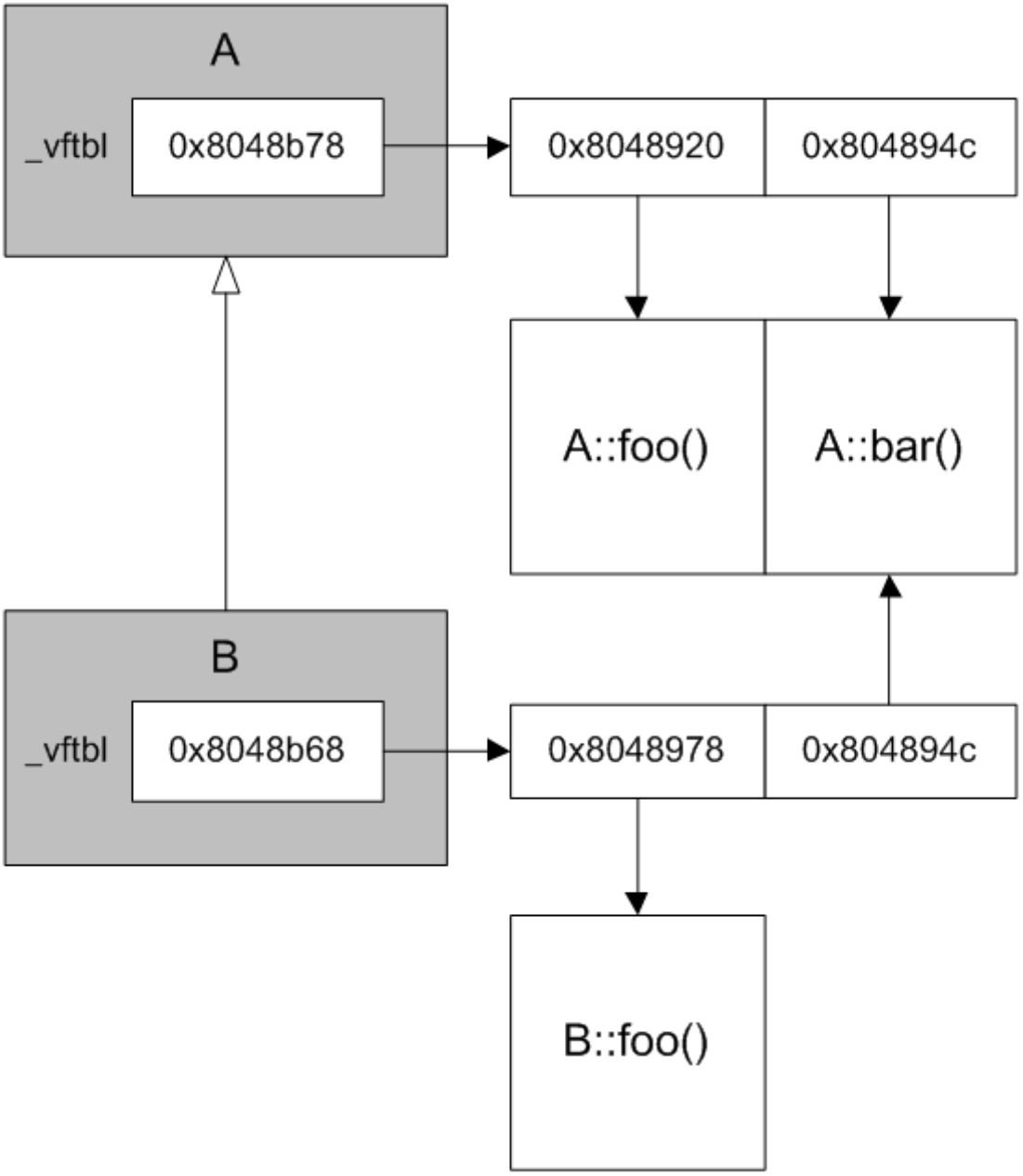
- 在成员函数声明的末尾加上“=0”，即将其声明为纯虚函数
- 纯虚函数亦称抽象函数或抽象方法，代表类的抽象行为，无需亦无法给出函数定义
- 至少拥有一个纯虚函数的类称为抽象类，抽象类表示某种抽象概念，不应亦不能实例化为对象
- 抽象类的子类，如果没有为基类中的所有纯虚函数提供覆盖实现，那么它也是一个抽象类
- 全部由纯虚函数构成的抽象类称为纯抽象类或接口类

```
// abstract.cpp

// 纯虚函数、抽象类和纯抽象类

#include <iostream>

using namespace std;

```

```cpp
class Shape {
public:
    Shape(double x, double y) : x(x), y(y) {}

    virtual void draw(void) const = 0;

protected:
    double x, y;
};

class Rect : public Shape {
public:
    Rect(double x, double y, double w, double h) : Shape(x, y), w(w), h(h) {}

    void draw(void) const {
        cout << "Rect(" << x << ',' << y << ',' << w << ',' << h << ')' << endl;
    }

private:
    double w, h;
};

class Circle : public Shape {
public:
    Circle(double x, double y, double r) : Shape(x, y), r(r) {}

    void draw(void) const {
        cout << "Circle(" << x << ',' << y << ',' << r << ')' << endl;
    }

private:
    double r;
};

int main(void) {
    Rect rect(1, 2, 3, 4);
    Shape* pshape = &rect;
    pshape->draw();

    Circle circle(5, 6, 7);
    Shape& rshape = circle;
    rshape.draw();

    // Shape shape(8, 9); // 编译错误

    return 0;
}
```

5.2.6 动态绑定

5.2.6.1 动态绑定的实现机理——虚函数表




```cpp
// vptr.cpp

// 虚函数表

#include <string.h>

#include <iostream>

using namespace std;

typedef void (*VFUN)(void*);
typedef VFUN* VPTR;

class A {
```

```
15  public:
16      A(int n) : n(n) {
17          VPTR vptr = *(VPTR*)this;
18          cout << "A::A()   : " << vptr << "->[" <<
19              (void*)vptr[0] << ',' <<
20              (void*)vptr[1] << ']' << endl;
21      }
22
23      ~A(void) {
24          VPTR vptr = *(VPTR*)this;
25          cout << "A::~A()  : " << vptr << "->[" <<
26              (void*)vptr[0] << ',' <<
27              (void*)vptr[1] << ']' << endl;
28      }
29
30      virtual void foo(void) {
31          cout << "A::foo() : " << n << endl;
32      }
33
34      virtual void bar(void) {
35          cout << "A::bar() : " << n << endl;
36      }
37
38  protected:
39      int n;
40  };
41
42  class B : public A {
43  public:
44      B(int n) : A(n) {
45          VPTR vptr = *(VPTR*)this;
46          cout << "B::B()   : " << vptr << "->[" <<
47              (void*)vptr[0] << ',' <<
48              (void*)vptr[1] << ']' << endl;
49      }
50
51      ~B(void) {
52          VPTR vptr = *(VPTR*)this;
53          cout << "B::~B()  : " << vptr << "->[" <<
54              (void*)vptr[0] << ',' <<
55              (void*)vptr[1] << ']' << endl;
56      }
57
58      void foo(void) {
59          cout << "B::foo() : " << n << endl;
60      }
61  };
62
63  int main (void) {
64      cout << "-------- 1 --------" << endl;
65
66      A a(100);
67      VPTR vptr = *(VPTR*)&a;
68      cout << "main()   : " << vptr << "->[" <<
69          (void*)vptr[0] << ',' <<
70          (void*)vptr[1] << ']' << endl;
```

```
    vptr[0](&a);
    vptr[1](&a);

    cout << "-------- 2 --------" << endl;

    B b(200);
    vptr = *(VPTR*)&b;
    cout << "main()   : " << vptr << "->[" <<
        (void*)vptr[0] << ',' <<
        (void*)vptr[1] << ']' << endl;
    vptr[0](&b);
    vptr[1](&b);

    cout << "-------- 3 --------" << endl;

    a = b;
    vptr = *(VPTR*)&a;
    cout << "main()   : " << vptr << "->[" <<
        (void*)vptr[0] << ',' <<
        (void*)vptr[1] << ']' << endl;
    vptr[0](&a);
    vptr[1](&a);

    cout << "-------- 4 --------" << endl;

    memcpy(&a, &b, sizeof(a));
    vptr = *(VPTR*)&a;
    cout << "main()   : " << vptr << "->[" <<
        (void*)vptr[0] << ',' <<
        (void*)vptr[1] << ']' << endl;
    vptr[0](&a);
    vptr[1](&a);
    a.foo();
    A& ra = a;
    ra.foo();

    return 0;
}
```

```
A* pa = new B;
```

当编译器看到下面这条语句时：

```
pa->foo();
```

它并不知道pa所指向对象的真实身份，编译器所能做的就是用一段代码替代上面这行调用语句。这段代码将依次执行下列操作：

1. 首先弄清指针pa所指向对象的真实身份
2. 然后通过这个对象的虚函数表指针_vftbl访问其虚函数表，并找到与foo标识符相对应的虚函数代码入口地址
3. 根据入口地址，调用该函数

5.2.6.2 动态绑定对性能的影响

- 虚拟函数表的存在必然会增加内存方面的开销，但这种开销事实上并不是太大
- 通过动态绑定调用多态函数较之通过静态绑定调用普通函数，要多出几个步骤。这几个步骤必然要消耗掉几个CPU周期，对整个程序的执行效率稍有影响
- 动态绑定妨碍编译器通过内联来优化代码

5.2.7 好莱坞模式——职责分离

```
// PDF解析器，专注于解析PDF文件

class PDFParser {
public:
    ...
    void parse(const char* filename, ...) {
        ...
        drawText(...);
        ...
        drawRect(...);
        ...
        drawCircle(...);
        ...
        drawImage(...);
        ...
    }
    ...
protected:
    ...
    virtual void drawText(...) = 0;
    virtual void drawRect(...) = 0;
    virtual void drawCircle(...) = 0;
    virtual void drawImage(...) = 0;
    ...
};

// PDF渲染器，专注于绘制PDF文件中的可视元素

class PDFRender : public PDFParser {
public:
    ...
    void display(const char* filename, ...) {
        ...
        parse(filename, ...);
        ...
    }
    ...
protected:
    ...
    void drawText(...) {
        ...
    }

    void drawRect(...) {
        ...
```

```
    }

    void drawCircle(...) {
        ...
    }

    void drawImage(...) {
        ...
    }
    ...
};

int main(void) {
    ...
    PDFRender render(...);
    render.display("cppprimier.pdf", ...);
    ...
    return 0;
}
```

5.3 运行时类型信息——RTTI

5.3.1 动态类型转换

```
// dynamiccast.cpp

// 将指向子类对象的基类指针动态转换（dynamic_cast）为子类指针

#include <iostream>

using namespace std;

class A {
public:
    virtual void foo(void) {}
};

class B : public A {}; // 多态继承
class C : public A {}; // 多态继承
class D {};

int main(void) {
    B b;

    A* pa = &b;
    cout << "pa=" << pa << endl;

    B* pb = dynamic_cast<B*>(pa); // pa实际指向B类对象，转换成功
    cout << "pb=" << pb << endl;

    C* pc = dynamic_cast<C*>(pa); // pa没有指向C类对象，转换失败，安全
    cout << "pc=" << pc << endl;

    D* pd = dynamic_cast<D*>(pa); // pa没有指向D类对象，转换失败，安全
```

```
    cout << "pd=" << pd << endl;

    try {
        A& ra = b;
        D& rd = dynamic_cast<D&>(ra);
    }
    catch (bad_cast& ex) {
        cout << ex.what() << endl;
    }

    pb = static_cast<B*>(pa); // B是A的子类，转换成功
    cout << "pb=" << pb << endl;

    pc = static_cast<C*>(pa); // C是A的子类，转换成功，危险！
    cout << "pc=" << pc << endl;

    // pd = static_cast<D*>(pa); // 编译错误，D不是A的子类，安全
    // cout << "pd=" << pd << endl;

    // 编译期、运行期均不检查，永远成功，危险！

    pb = reinterpret_cast<B*>(pa);
    cout << "pb=" << pb << endl;

    pc = reinterpret_cast<C*>(pa);
    cout << "pc=" << pc << endl;

    pd = reinterpret_cast<D*>(pa);
    cout << "pd=" << pd << endl;

    return 0;
}
```

5.3.2 typeid操作符

```
// typeid.cpp

// typeid操作符

#include <typeinfo>
#include <iostream>

using namespace std;

template<typename T>
void typeName(const char* lable) {
    const type_info& ti = typeid(T);
    cout << lable << ": " << ti.name() << endl;
}

template<typename T>
void typeName(const T& t, const char* lable) {
    const type_info& ti = typeid(t);
    cout << lable << ": " << ti.name () << endl;
}
```

```cpp

class X {};

class Y : public X {}; // 普通继承

class A {
public:
    virtual void foo(void) {}
};

class B : public A {}; // 多态继承

int main(void) {
    typeName<void>("void");
    typeName<void*>("void*");
    typeName<void**>("void**");

    typeName<char>("char");
    typeName<unsigned char>("unsigned char");
    typeName<short>("short");
    typeName<unsigned short>("unsigned short");
    typeName<int>("int");
    typeName<unsigned int>("unsigned int");
    typeName<long>("long");
    typeName<unsigned long>("unsigned long");
    typeName<long long>("long long");
    typeName<unsigned long long>("unsigned long long");
    typeName<bool>("bool");
    typeName<float>("float");
    typeName<double>("double");
    typeName<long double>("long double");

    // 一个指向以函数指针为参数同时返回函数指针的函数的指针
    typeName<long long*(*(*)(int*(*)(short*)))(char*)>(
        "long long*(*(*)(int*(*)(short*)))(char*)");
    // long long*(
    //   *(*)(
    //     int*(*)(short*)
    //   )
    // )(char*)

    typeName<X>("X");
    typeName<Y>("Y");

    Y y;
    X* px = &y;
    X& rx = y;

    typeName(y, "y");
    typeName(px, "px");
    typeName(*px, "*px");
    typeName(rx, "rx");

    if (typeid(*px) == typeid(Y))
        cout << "typeid(*px) == typeid(Y)" << endl;
    else
```

```
        cout << "typeid(*px) != typeid(Y)" << endl;

    if (typeid(rx) == typeid(Y))
        cout << "typeid(rx) == typeid(Y)" << endl;
    else
        cout << "typeid(rx) != typeid(Y)" << endl;

    typeName<A>("A");
    typeName<B>("B");

    B b;
    A* pa = &b;
    A& ra = b;

    typeName(b, "b");
    typeName(pa, "pa");
    typeName(*pa, "*pa");
    typeName(ra, "ra");

    if (typeid(*pa) == typeid(B))
        cout << "typeid(*pa) == typeid(B)" << endl;
    else
        cout << "typeid(*pa) != typeid(B)" << endl;

    if (typeid(ra) == typeid(B))
        cout << "typeid(ra) == typeid(B)" << endl;
    else
        cout << "typeid(ra) != typeid(B)" << endl;

    return 0;
}
```

5.4 虚析构函数

5.4.1 通过基类指针析构子类对象的问题

当delete操作符作用于一个指向子类对象的基类指针时，如果基类的析构函数没有被声明为虚函数，那么实际被调用的将仅仅是基类的析构函数，而基类的析构函数并不会自动调用子类的析构函数，因此实际被销毁的将仅仅是这个子类对象中基类子对象所创建的动态资源，而该子类对象自己的动态资源将失去被释放的机会，形成泄漏

```
// delbd.cpp

// 通过基类指针析构子类对象的问题

#include <iostream>

using namespace std;

class X {
public:
    X(void) : data(new int[256]) {
        cout << "X::X() invoked" << endl;
    }
```

```
14
15      ~X(void) {
16          cout << "X::~X() invoked" << endl;
17          delete[] data;
18      }
19
20  private:
21      int* data;
22  };
23
24  class Y : public X {
25  public:
26      Y(void) : text(new char[1024]) {
27          cout << "Y::Y() invoked" << endl;
28      }
29
30      ~Y(void) {
31          cout << "Y::~Y() invoked" << endl;
32          delete[] text;
33      }
34
35  private:
36      char* text;
37  };
38
39  int main(void) {
40      Y* y = new Y;
41      delete y; // 先调Y类的析构函数，再调X类的析构函数
42
43      X* x = new Y;
44      delete x; // 只调X类的析构函数，不调Y类的析构函数
45
46      return 0;
47  }
```

5.4.2 虚析构方案

当delete操作符作用于一个指向子类对象的基类指针时，如果基类的析构函数被声明为虚函数，那么实际被调用的将是子类的析构函数，而子类的析构函数在释放完子类对象自己的动态资源后，会自动调用基类的析构函数，完成对子类对象中基类子对象所创建动态资源的释放，避免泄漏

```
1   // virdes.cpp
2
3   // 虚析构方案
4
5   #include <iostream>
6
7   using namespace std;
8
9   class X {
10  public:
11      X(void) : data(new int[256]) {
12          cout << "X::X() invoked" << endl;
13      }
14
```

```
    // 虚析构函数
    virtual ~X(void) {
        cout << "X::~X() invoked" << endl;
        delete[] data;
    }

private:
    int* data;
};

class Y : public X {
public:
    Y(void) : text(new char[1024]) {
        cout << "Y::Y() invoked" << endl;
    }

    ~Y(void) {
        cout << "Y::~Y() invoked" << endl;
        delete[] text;
    }

private:
    char* text;
};

int main(void) {
    Y* y = new Y;
    delete y; // 先调Y类的析构函数，再调X类的析构函数

    X* x = new Y;
    delete x; // 先调Y类的析构函数，再调X类的析构函数

    return 0;
}
```

5.4.3 空虚析构函数

- 子类的析构函数会自动调用基类的析构函数，但基类的析构函数不会自动调用子类的析构函数
- 将基类的析构函数声明为虚函数，可以保证子类的析构函数总被执行
- 在不定义析构函数的情况下，系统提供的缺省析构函数并非虚函数
- 有时必须为基类定义虚析构函数，即使它是个什么也不做的空函数，仅仅为了保证子类的析构函数被执行

```
// emptyvirdes.cpp

// 空虚析构函数

#include <iostream>

using namespace std;

class X {
public:
```

```
    // 空虚析构函数
    virtual ~X(void) {}

private:
    int data[256];
};

class Y : public X {
public:
    Y(void) : text(new char[1024]) {
        cout << "Y::Y() invoked" << endl;
    }

    ~Y(void) {
        cout << "Y::~Y() invoked" << endl;
        delete[] text;
    }

private:
    char* text;
};

int main(void) {
    X* x = new Y;
    delete x; // 一定会调Y类的析构函数

    return 0;
}
```

> 思考：构造函数可以是虚函数吗？静态成员函数可以是虚函数吗？