3 操作符重载

3.1 操作符重载的一般原则

- 操作符的通用语法
  - 双目操作符：<左操作数><操作符><右操作数>，简单表示为L#R
  - 单目操作符：<操作数><操作符>或<操作符><操作数>，简单表示为O#或#O
- 重载操作符的操作数中至少有一个是类类型或枚举类型
- 重载操作符的操作数中至少有一个不是内建类型，如int，char，double等及其指针或引用
- 操作符的优先级不会因重载而发生改变
- 操作符的操作数个数不会因重载而发生改变
- 除函数操作符（()）外的所有操作符，其操作符函数都不能带有缺省参数
- 并不是所有操作符都能重载，如::、.*、.和?:等操作符无法重载

3.2 操作符标记与操作符函数

- 将一个操作符标记（#）应用于一个或多个类类型的操作数时，编译器将调用与这个操作符标记相关联的操作符函数（operator#）。例如：

操作符标记	操作符函数
=	operator=
+	operator+
+=	operator+=
++	operator++
<<	operator<<
[]	operator[]

- 操作符函数的定义通常包括全局函数和成员函数两种形式

3.3 插入/提取操作符

- 一般而言，如果一个插入/提取操作符函数已经以全局函数形式被定义了，那么表达式

```
L#R;
```

将被编译器解释为如下函数调用：

```
operator#(L,R);
```

可见，以全局函数形式定义的插入/提取操作符函数，应带有两个参数，分别是该操作符的左操作数和右操作数

```
// io.cpp
```

```cpp

// 插入/提取操作符

// cin>><操作数>被编译为operator>>(cin,<操作数>)
// cout<<<操作数>被编译为operator<<(cout,<操作数>)

#include <iostream>
#include <iomanip>

using namespace std;

class Complex {
public:
    Complex(double real = 0, double imag = 0) : real(real), imag(imag)
{}

    double getReal(void) const {
        return real;
    }

    void setReal(double real) {
        this->real = real;
    }

    double getImag(void) const {
        return imag;
    }

    void setImag(double imag) {
        this->imag = imag;
    }

private:
    double real, imag;
};

// 输入流提取操作符函数
istream& operator>>(istream& is, Complex& c) {
    double real, imag;
    is >> real >> imag;

    c.setReal(real);
    c.setImag(imag);

    return is;
}

// 输出流插入操作符函数
ostream& operator<<(ostream& os, const Complex& c) {
    return os << '(' << c.getReal() << showpos << c.getImag() <<
noshowpos << "i)";
}

int main(void) {
    Complex c;

```

```
    cin >> c; // 被编译为operator>>(cin,c)
    cout << c << endl; // 被编译为operator<<(cout,c)<<endl

    return 0;
}
```

- 为了能在全局函数形式的操作符函数中直接访问操作数的私有成员，不妨将其声明为操作数类型的友元

```
// friendio.cpp

// 将插入/提取操作符函数声明为操作数类型的友元

#include <iostream>
#include <iomanip>

using namespace std;

class Complex {
public:
    Complex(double real = 0, double imag = 0) : real(real), imag(imag) {}

private:
    double real, imag;

    // 将插入/提取操作符函数声明为Complex类的友元

    friend istream& operator>>(istream&, Complex&);
    friend ostream& operator<<(ostream&, const Complex&);
};

istream& operator>>(istream& is, Complex& c) {
    // 友元可以直接访问类的私有成员
    return is >> c.real >> c.imag;
}

ostream& operator<<(ostream& os, const Complex& c) {
    // 友元可以直接访问类的私有成员
    return os << '(' << c.real << showpos << c.imag << noshowpos << "i)";
}

int main(void) {
    Complex c;

    cin >> c;
    cout << c << endl;

    return 0;
}
```

- 友元声明可以出现在一个类的私有、保护或公有部分，它允许被声明者直接访问该类的所有数据成员和成员函数，无论其访问控制属性是私有的、保护的还是公有的。

```cpp
// friend.cpp

// 友元函数和友元类

#include <iostream>
#include <iomanip>

using namespace std;

class Complex {
public:
    Complex(double real = 0, double imag = 0) : real(real), imag(imag) {}

private:
    double real, imag;

    // 友元函数

    friend istream& operator>>(istream&, Complex&);
    friend ostream& operator<<(ostream&, const Complex&);

    // 友元类

    friend class Sum;
};

istream& operator>>(istream& is, Complex& c) {
    return is >> c.real >> c.imag;
}

ostream& operator<<(ostream& os, const Complex& c) {
    return os << '(' << c.real << showpos << c.imag << noshowpos << "i)";
}

// Complex类的友元类，可以直接访问Complex类的私有成员
class Sum {
public:
    Sum(const Complex& c1, const Complex& c2) :
        c(c1.real + c2.real, c1.imag + c2.imag) {}

private:
    Complex c;

    // 友元函数可以在声明的同时给出定义
    friend void show(const Sum& sum) {
        cout << sum.c << endl;
    }
};

int main(void) {
    Complex c1, c2;
    cin >> c1 >> c2;

    Sum sum(c1, c2);
```

```
    show(sum);

    return 0;
}
```

3.4 双目操作符

- 一般而言，如果一个双目操作符函数已经以成员函数形式被定义了，那么表达式

```
L#R;
```

将被编译器解释为如下函数调用：

```
L.operator#(R);
```

可见，以成员函数形式定义的双目操作符函数，只有一个参数，即该操作符的右操作数，而其左操作数将作为此操作符函数的调用对象

```
// membinary.cpp

// 成员函数形式的双目操作符函数

// <左操作数><运算符><右操作数>被编译为<左操作数>.operator<运算符>(<右操作数>)

#include <iostream>
#include <iomanip>

using namespace std;

class Complex {
public:
    Complex(double real = 0, double imag = 0) : real(real), imag(imag)
{}

    // 双目操作符函数

    // 注意返回值还是返回引用，要与操作符在表达式中的语义保持一致

    const Complex operator+(const Complex& c) const {
        return Complex(real + c.real, imag + c.imag);
    }

    const Complex operator-(const Complex& c) const {
        return Complex(real - c.real, imag - c.imag);
    }

    Complex& operator+=(const Complex& c) {
        real += c.real;
        imag += c.imag;
        return *this;
    }

    Complex& operator-=(const Complex& c) {
        real -= c.real;
```

```
        imag -= c.imag;
        return *this;
    }

private:
    double real, imag;

    friend istream& operator>>(istream& is, Complex& c) {
        return is >> c.real >> c.imag;
    }

    friend ostream& operator<<(ostream& os, const Complex& c) {
        return os << '(' << c.real << showpos << c.imag << noshowpos << "i)";
    }
};

int main(void) {
    Complex c1, c2, c3, c4;
    cin >> c1 >> c2 >> c3 >> c4;

    cout << c1 << '+' << c2 << '=' << c1 + c2 << endl;
    cout << c1 << '-' << c2 << '=' << c1 - c2 << endl;

    // (c1 + c2) = c3; // 禁止
    // (c1 - c2) = c4; // 禁止

    cout << c1 << "+=" << c2 << endl;
    c1 += c2;
    cout << c1 << endl;

    cout << c1 << "-=" << c2 << endl;
    c1 -= c2;
    cout << c1 << endl;

    cout << '(' << c1 << "+=" << c2 << ")=" << c3 << endl;
    (c1 += c2) = c3; // 允许
    cout << c1 << endl;

    cout << '(' << c1 << "-=" << c2 << ")=" << c4 << endl;
    (c1 -= c2) = c4; // 允许
    cout << c1 << endl;

    return 0;
}
```

- 以成员函数形式定义的双目操作符函数，被其左操作数调用，并传入右操作数实参。因此，该操作符函数显然应该被定义为左操作数类型的成员函数，其参数为右操作数类型的对象
- 插入/提取操作符左操作数的类型通常为ostream/istream，因此即使能以成员函数形式定义这两个操作符函数，也应该将其定义为ostream/istream类的成员函数。如果不想修改这两个标准库提供的预定义类，就只能以全局函数形式定义这两个操作符函数，同时将其声明为被插入/提取对象类型的友元，以使其更接近于成员函数形式

3.5 单目操作符

- 一般而言，如果一个单目操作符函数已经以成员函数的形式被定义了，那么表达式

```
#o;
```

将被编译器解释为如下函数调用：

```
o.operator#();
```

可见，以成员函数形式定义的单目操作符函数，没有任何参数，而该操作符唯一的一个操作数将作为此操作符函数的调用对象

```
// memunary.cpp

// 成员函数形式的单目操作符函数

// <运算符><操作数>被编译为<操作数>.operator<运算符>()

#include <iostream>
#include <iomanip>

using namespace std;

class Complex {
public:
    Complex(double real = 0, double imag = 0) : real(real), imag(imag)
{}

    // 单目操作符函数

    const Complex operator-(void) const {
        return Complex(-real, -imag);
    }

private:
    double real, imag;

    friend istream& operator>>(istream& is, Complex& c) {
        return is >> c.real >> c.imag;
    }

    friend ostream& operator<<(ostream& os, const Complex& c) {
        return os << '(' << c.real << showpos << c.imag << noshowpos <<
"i)";
    }
};

int main(void) {
    Complex c;
    cin >> c;

    cout << '-' << c << '=' << -c << endl;

```

```
    return 0;
}
```

3.6 自增减操作符

操作符表达式	操作符函数调用
++O	O.operator++()
O++	O.operator++(0)
--O	O.operator--()
O--	O.operator--(0)

- 前缀操作符表达式的值是操作数本身（左值），因此可以重复使用

```
++++O; // 允许
```

- 后缀操作符表达式的值是操作数自增减前的历史值（右值），因此不能重复使用

```
O++++; // 禁止
```

```
// memself.cpp

// 成员函数形式的自增减操作符函数

// ++<操作数>被编译为<操作数>.operator++()
// <操作数>++被编译为<操作数>.operator++(0)

// --<操作数>被编译为<操作数>.operator--()
// <操作数>--被编译为<操作数>.operator--(0)

#include <iostream>

using namespace std;

class Integer {
public:
    Integer(int n = 0) : n(n) {}

    // 自增减操作符函数

    // 注意自增减操作符函数的参数和返回值，因前后缀不同而有差异

    // 前++操作符函数
    Integer& operator++(void) {
        ++n;
        return *this;
    }

    // 后++操作符函数
    const Integer operator++(int) {
        Integer i = *this;
```

```
        ++n;
        return i;
    }

    // 前--操作符函数
    Integer& operator--(void) {
        --n;
        return *this;
    }

    // 后--操作符函数
    const Integer operator--(int) {
        Integer i = *this;
        --n;
        return i;
    }

private:
    int n;

    friend istream& operator>>(istream& is, Integer& i) {
        return is >> i.n;
    }

    friend ostream& operator<<(ostream& os, const Integer& i) {
        return os << i.n;
    }
};

int main(void) {
    Integer i;
    cin >> i;

    cout << "++" << i << '=' << ++i << ", i=" << i << endl;
    cout << i << "++=" << i++ << ", i=" << i << endl;

    cout << "--" << i << '=' << --i << ", i=" << i << endl;
    cout << i << "--=" << i-- << ", i=" << i << endl;

    cout << "++++" << i << '=' << ++++i << ", i=" << i << endl; // 允许
    cout << "----" << i << '=' << ----i << ", i=" << i << endl; // 允许

    // i++++; // 禁止
    // i----; // 禁止

    return 0;
}
```

3.7 成员还是友元

- 一个操作符的左右操作数不一定都是相同类型的对象，这就涉及到将该操作符函数定义为谁的成员，谁的友元
    - 一个操作符函数被声明为哪个类的友元，取决于该函数参数对象的类型
    - 一个操作符函数被定义为哪个类的成员，取决于该函数调用对象的类型

```cpp
// memfriend.cpp

// 一个操作符函数被声明为哪个类的友元，取决于该函数参数对象的类型
// 一个操作符函数被定义为哪个类的成员，取决于该函数调用对象的类型

#include <iostream>

using namespace std;

class Mail;

class Mailbox { // 调用对象的类型
public:
    Mailbox(const char* account) : account(account) {}

    const Mailbox& operator<<(const Mail&) const; // 成员声明

private:
    string account;
};

class Mail { // 参数对象的类型
public:
    Mail (const char* title, const char* content) : title(title), content(content) {}

    friend const Mailbox& Mailbox::operator<<(const Mail&) const; // 友元声明

private:
    string title, content;
};

// 插入操作符函数
const Mailbox& Mailbox::operator<<(const Mail& mail) const {
    cout << '[' << mail.title << '|' << mail.content << "]->" << account << endl;
    return *this;
}

int main(void) {
    Mail m1("Hello", "Good lucky for you!"), m2("Notify", "You are fired.");
    Mailbox mb("zhangjw@tedu.cn");

    mb << m1 << m2;

    return 0;
}
```

3.8 类型转换操作符函数与自定义类型转换

- 通过类型转换操作符函数，可以实现自定义类型转换
- 类型转换操作符函数只能被定义为源类型的成员函数，而不能被定义为全局函数
- 当需要进行类型转换时（构造、赋值、函数调用等），编译器将产生如下函数调用：

```
源类型对象.operator目标类型();
```

该函数返回一个目标类型的对象

```
// opconv.cpp

// 通过类型转换操作符函数，可以实现自定义类型转换

#include <iostream>

using namespace std;

class Point3D {
public:
    Point3D(int x = 0, int y = 0, int z = 0) : x(x), y(y), z(z) {}

private:
    int x, y, z;

    friend ostream& operator<<(ostream& os, const Point3D& p) {
        return os << '(' << p.x << ',' << p.y << ',' << p.z << ')';
    }
};

class Point2D {
public:
    Point2D(int x = 0, int y = 0) : x(x), y(y) {}

    // 类型转换操作符函数

    // Point2D -> Point3D
    operator Point3D(void) const {
        return Point3D(x, y);
    }

private:
    int x, y;

    friend ostream& operator<<(ostream& os, const Point2D& p) {
        return os << '(' << p.x << ',' << p.y << ')';
    }
};

void foo(Point3D p) {
    cout << p << endl;
}

Point3D bar(void) {
    Point2D p(6, 7);
    return p; // 从函数返回过程中的类型转换
}

int main(void) {
    Point2D p1(1, 2);
    cout << p1 << endl;
```

```
52
53      Point3D p2 = p1; // 构造过程中的类型转换
54      cout << p2 << endl;
55
56      Point3D p3(3, 4, 5);
57      cout << p3 << endl;
58      p3 = p1; // 赋值过程中的类型转换
59      cout << p3 << endl;
60
61      foo(p1); // 向函数传参过程中的类型转换
62
63      cout << bar() << endl;
64
65      return 0;
66  }
```

- 通过构造函数实现自定义类型转换。以源类型对象为参数，通过目标类型中特定的构造函数，创建一个目标类型的对象

```
1   // consconv.cpp
2
3   // 通过构造函数实现自定义类型转换
4
5   #include <iostream>
6
7   using namespace std;
8
9   class Point2D;
10
11  class Point3D {
12  public:
13      Point3D(int x = 0, int y = 0, int z = 0) : x(x), y(y), z(z) {}
14
15      // 类型转换构造函数
16
17      // Point2D -> Point3D
18      Point3D(Point2D const& p);
19
20  private:
21      int x, y, z;
22
23      friend ostream& operator<<(ostream& os, const Point3D& p) {
24          return os << '(' << p.x << ',' << p.y << ',' << p.z << ')';
25      }
26  };
27
28  class Point2D {
29  public:
30      Point2D(int x = 0, int y = 0) : x(x), y(y) {}
31
32  private:
33      int x, y;
34
35      friend ostream& operator<<(ostream& os, const Point2D& p) {
36          return os << '(' << p.x << ',' << p.y << ')';
37      }
```

```
    friend class Point3D;
};

Point3D::Point3D(Point2D const& p) : x(p.x), y(p.y), z(0) {}

void foo(Point3D p) {
    cout << p << endl;
}

Point3D bar(void) {
    Point2D p(6, 7);
    return p; // 从函数返回过程中的类型转换
}

int main(void) {
    Point2D p1(1, 2);
    cout << p1 << endl;

    Point3D p2 = p1; // 构造过程中的类型转换
    cout << p2 << endl;

    Point3D p3(3, 4, 5);
    cout << p3 << endl;
    p3 = p1; // 赋值过程中的类型转换
    cout << p3 << endl;

    foo(p1); // 向函数传参过程中的类型转换

    cout << bar() << endl;

    return 0;
}
```

- 在源类型中定义类型转换操作符函数，同时在目标类型中定义类型转换构造函数，通常优先选择后者

```
// opcons.cpp

// 同时提供类型转换操作符函数和类型转换构造函数

#include <iostream>

using namespace std;

class Point2D;

class Point3D {
public:
    Point3D(int x = 0, int y = 0, int z = 0) : x(x), y(y), z(z) {}

    // 类型转换构造函数

    // Point2D -> Point3D
    Point3D(Point2D const& p);

```

```
private:
    int x, y, z;

    friend ostream& operator<<(ostream& os, const Point3D& p) {
        return os << '(' << p.x << ',' << p.y << ',' << p.z << ')';
    }
};

class Point2D {
public:
    Point2D(int x = 0, int y = 0) : x(x), y(y) {}

    // 类型转换操作符函数

    // Point2D -> Point3D
    operator Point3D(void) const {
        cout << "Point2D::operator Point3D()" << endl;

        return Point3D(x, y);
    }

private:
    int x, y;

    friend ostream& operator<<(ostream& os, const Point2D& p) {
        return os << '(' << p.x << ',' << p.y << ')';
    }

    friend class Point3D;
};

Point3D::Point3D(Point2D const& p) : x(p.x), y(p.y), z(0) {
    cout << "Point3D::Point3D(Point2D)" << endl;
}

void foo(Point3D p) {
    cout << p << endl;
}

Point3D bar(void) {
    Point2D p(6, 7);
    return p; // 从函数返回过程中的类型转换
}

int main(void) {
    Point2D p1(1, 2);
    cout << p1 << endl;

    Point3D p2 = p1; // 构造过程中的类型转换
    cout << p2 << endl;

    Point3D p3(3, 4, 5);
    cout << p3 << endl;
    p3 = p1; // 赋值过程中的类型转换
    cout << p3 << endl;

```

```
    foo(p1); // 向函数传参过程中的类型转换

    cout << bar() << endl;

    return 0;
}
```

- 为类型转换构造函数添加explicit修饰符，指明该函数仅用于显式类型转换，以此防止误转换

```
// expcons.cpp

// 为类型转换构造函数添加explicit修饰符

#include <iostream>

using namespace std;

class Point2D;

class Point3D {
public:
    Point3D(int x = 0, int y = 0, int z = 0) : x(x), y(y), z(z) {}

    // 类型转换构造函数

    // Point2D -> Point3D
    explicit Point3D(Point2D const& p);

private:
    int x, y, z;

    friend ostream& operator<<(ostream& os, const Point3D& p) {
        return os << '(' << p.x << ',' << p.y << ',' << p.z << ')';
    }
};

class Point2D {
public:
    Point2D(int x = 0, int y = 0) : x(x), y(y) {}

private:
    int x, y;

    friend ostream& operator<<(ostream& os, const Point2D& p) {
        return os << '(' << p.x << ',' << p.y << ')';
    }

    friend class Point3D;
};

Point3D::Point3D(Point2D const& p) : x(p.x), y(p.y), z(0) {}

void foo(Point3D p) {
    cout << p << endl;
}

```

```
Point3D bar(void) {
    Point2D p(6, 7);
    // return p; // 编译错误
    return Point3D(p);
}

int main(void) {
    Point2D p1(1, 2);
    cout << p1 << endl;

    // Point3D p2 = p1; // 编译错误
    Point3D p2(p1);
    cout << p2 << endl;

    Point3D p3(3, 4, 5);
    cout << p3 << endl;
    // p3 = p1; // 编译错误
    p3 = (Point3D)p1;
    cout << p3 << endl;

    // foo(p1); // 编译错误
    foo(static_cast<Point3D>(p1));

    cout << bar() << endl;

    return 0;
}
```

- 为类型转换操作符函数添加explicit修饰符，指明该函数仅用于显式类型转换，依次防止误转换

```
// expop.cpp

// 为类型转换操作符函数添加explicit修饰符

#include <iostream>

using namespace std;

class Point3D {
public:
    Point3D(int x = 0, int y = 0, int z = 0) : x(x), y(y), z(z) {}

private:
    int x, y, z;

    friend ostream& operator<<(ostream& os, const Point3D& p) {
        return os << '(' << p.x << ',' << p.y << ',' << p.z << ')';
    }
};

class Point2D {
public:
    Point2D(int x = 0, int y = 0) : x(x), y(y) {}

    // 类型转换操作符函数

```

```
    // Point2D -> Point3D
    explicit operator Point3D(void) const {
        return Point3D(x, y);
    }

private:
    int x, y;

    friend ostream& operator<<(ostream& os, const Point2D& p) {
        return os << '(' << p.x << ',' << p.y << ')';
    }
};

void foo(Point3D p) {
    cout << p << endl;
}

Point3D bar(void) {
    Point2D p(6, 7);
    // return p; // 编译错误
    return Point3D(p);
}

int main(void) {
    Point2D p1(1, 2);
    cout << p1 << endl;

    // Point3D p2 = p1; // 编译错误
    Point3D p2(p1);
    cout << p2 << endl;

    Point3D p3(3, 4, 5);
    cout << p3 << endl;
    // p3 = p1; // 编译错误
    p3 = (Point3D)p1;
    cout << p3 << endl;

    // foo(p1); // 编译错误
    foo(static_cast<Point3D>(p1));

    cout << bar() << endl;

    return 0;
}
```

此特性要求编译器支持C++11及以上标准

3.9 解引用操作符（*和->）与智能指针

```
// userptr.cpp

// 简单的智能指针

#include <iostream>

```

```cpp
using namespace std;

class User {
public:
    User(const string& name = "Anonymous", int age = -1) : name(name), age(age) {
        cout << "User::User() invoked" << endl;
    }

    ~User(void) {
        cout << "User::~User() invoked" << endl;
    }

    void print(void) const {
        cout << *this << endl;
    }

    friend ostream& operator<< (ostream& os, const User& user) {
        return os << '(' << user.name << ',' << user.age << ')';
    }

private:
    string name;
    int age;
};

class UserPtr {
public:
    // 类型转换构造函数
    explicit UserPtr(User* user = NULL) : user(user) {}

    // 拷贝构造函数
    UserPtr(UserPtr& up) : user(up.release()) {}

    // 拷贝赋值操作符函数
    UserPtr& operator=(UserPtr& up) {
        if (&up != this)
            reset(up.release()); // 永远只有一个智能指针持有对象的地址
        return *this;
    }

    // 析构函数
    ~UserPtr(void) {
        if (user)
            delete user; // 析构时销毁对象
    }

    // 解引用操作符函数
    User& operator*(void) const {
        return *user;
    }

    // 间接成员访问操作符函数
    User* operator->(void) const {
        return user;
    }
```

```
62
63  private:
64      User* release(void) {
65          User* ret = user;
66          user = NULL;
67          return ret;
68      }
69
70      void reset(User* user) {
71          if (user != this->user) {
72              delete this->user;
73              this->user = user;
74          }
75      }
76
77  private:
78      User* user;
79  };
80
81  int main(void) {
82      UserPtr up1(new User("Zaphod", 49)); // 类型转换构造
83      cout << *up1 << endl; // 解引用
84      up1->print(); // 间接成员访问
85
86      UserPtr up2 = up1; // 拷贝构造
87      cout << *up2 << endl; // 解引用
88      up2->print();// 间接成员访问
89
90      UserPtr up3(new User("Antony", 37)); // 类型转换构造
91      cout << *up3 << endl; // 解引用
92      up3->print(); // 间接成员访问
93
94      up3 = up2; // 拷贝赋值
95      cout << *up3 << endl; // 解引用
96      up3->print(); // 间接成员访问
97
98      return 0;
99  } // 析构
```

```
1   // autoptr.cpp
2
3   // 标准库的智能指针
4
5   #include <memory>
6
7   #include <iostream>
8
9   using namespace std;
10
11  class User {
12  public:
13      User(const string& name = "Anonymous", int age = -1) : name(name),
    age(age) {
14          cout << "User::User() invoked" << endl;
15      }
```

```
16
17      ~User(void) {
18          cout << "User::~User() invoked" << endl;
19      }
20
21      void print(void) const {
22          cout << *this << endl;
23      }
24
25      friend ostream& operator<< (ostream& os, const User& user) {
26          return os << '(' << user.name << ',' << user.age << ')';
27      }
28
29  private:
30      string name;
31      int age;
32  };
33
34  int main(void) {
35      auto_ptr<User> up1(new User("Zaphod", 49)); // 类型转换构造
36      cout << *up1 << endl; // 解引用
37      up1->print(); // 间接成员访问
38
39      auto_ptr<User> up2 = up1; // 拷贝构造
40      cout << *up2 << endl; // 解引用
41      up2->print();// 间接成员访问
42
43      auto_ptr<User> up3(new User("Antony", 37)); // 类型转换构造
44      cout << *up3 << endl; // 解引用
45      up3->print(); // 间接成员访问
46
47      up3 = up2; // 拷贝赋值
48      cout << *up3 << endl; // 解引用
49      up3->print(); // 间接成员访问
50
51      // auto_ptr<User> up4(new User[3]); // 导致崩溃
52
53      return 0;
54  } // 析构
```

3.10 函数操作符

```
1   // func.cpp
2
3   // 函数操作符
4
5   #include <iostream>
6
7   using namespace std;
8
9   class Square {
10  public:
11      // 函数操作符函数
12
13      int operator()(int n) const {
```

```
        cout << "Square::operator()(int) invoked" << endl;

        return n * n;
    }

    double operator()(double f) const {
        cout << "Square::operator()(double) invoked" << endl;

        return f * f;
    }
};

int main(void) {
    Square square;

    // 在对象上应用函数语法

    cout << square(2) << endl;
    cout << square(2.5) << endl;

    return 0;
}
```

3.11 new/delete操作符

```
// newdel.cpp

// new/delete操作符

#include <stdlib.h>

#include <iostream>

using namespace std;

// 全局作用域的new/delete操作符函数

void* operator new(size_t size) {
    void* p = malloc(size);
    cout << "::operator new(" << size << ") invoked -> " << p << endl;
    return p;
}

void* operator new[](size_t size) {
    void* p = malloc(size);
    cout << "::operator new[](" << size << ") invoked -> " << p << endl;
    return p;
}

void operator delete(void* p) {
    cout << "::operator delete(" << p << ") invoked" << endl;
    free(p);
}

void operator delete[](void* p) {
```

```
31      cout << "::operator delete[](" << p << ") invoked" << endl;
32      free(p);
33  }
34
35  class A {
36  public:
37      A(void) {
38          cout << "A::A() invoked, this=" << this << endl;
39      }
40
41      ~A (void) {
42          cout << "A::~A() invoked, this=" << this << endl;
43      }
44  };
45
46  class B {
47  public:
48      // 类作用域的new/delete操作符函数
49
50      static void* operator new(size_t size) {
51          void* p = malloc(size);
52          cout << "B::operator new(" << size << ") invoked -> " << p << endl;
53          return p;
54      }
55
56      static void* operator new[](size_t size) {
57          void* p = malloc(size);
58          cout << "B::operator new[](" << size << ") invoked -> " << p <<
    endl;
59          return p;
60      }
61
62      static void operator delete(void* p) {
63          cout << "B::operator delete(" << p << ") invoked" << endl;
64          free(p);
65      }
66
67      static void operator delete[](void* p) {
68          cout << "B::operator delete[](" << p << ") invoked" << endl;
69          free(p);
70      }
71
72      B(void) {
73          cout << "B::B() invoked, this=" << this << endl;
74      }
75
76      ~B(void) {
77          cout << "B::~B() invoked, this=" << this << endl;
78      }
79  };
80
81  int main(void) {
82      int* n = new int;
83      cout << "n=" << n << endl;
84      delete n;
85
```

```
    double* d = new double;
    cout << "d=" << d << endl;
    delete d;

    char* ca = new char[128];
    cout << "ca=" << reinterpret_cast<void*>(ca) << endl;
    delete[] ca;

    A* a = new A;
    cout << "a=" << a << endl;
    delete a;

    A* aa = new A[3];
    cout << "aa=" << aa << endl;
    delete[] aa;

    B* b = new B;
    cout << "b=" << b << endl;
    delete b;

    B* ba = new B[3];
    cout << "ba=" << ba << endl;
    delete[] ba;

    return 0;
}
```