

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

```
1 | L#R;
```

将被编译器解释为如下函数调用:

```
1 | 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

```
1 | // io.cpp
```

```

2
3 // 插入/提取操作符
4
5 // cin>><操作数>被编译为operator>>(cin,<操作数>)
6 // cout<<<操作数>被编译为operator<<(cout,<操作数>)
7
8 #include <iostream>
9 #include <iomanip>
10
11 using namespace std;
12
13 class Complex {
14 public:
15     Complex(double real = 0, double imag = 0) : real(real), imag(imag)
16     {}
17
18     double getReal(void) const {
19         return real;
20     }
21
22     void setReal(double real) {
23         this->real = real;
24     }
25
26     double getImag(void) const {
27         return imag;
28     }
29
30     void setImag(double imag) {
31         this->imag = imag;
32     }
33 private:
34     double real, imag;
35 };
36
37 // 输入流提取操作符函数
38 istream& operator>>(istream& is, Complex& c) {
39     double real, imag;
40     is >> real >> imag;
41
42     c.setReal(real);
43     c.setImag(imag);
44
45     return is;
46 }
47
48 // 输出流插入操作符函数
49 ostream& operator<<(ostream& os, const Complex& c) {
50     return os << '(' << c.getReal() << showpos << c.getImag() <<
51     noshowpos << "i)";
52 }
53
54 int main(void) {
55     Complex c;

```

```

56     cin >> c; // 被编译为operator>>(cin,c)
57     cout << c << endl; // 被编译为operator<<(cout,c)<<endl
58
59     return 0;
60 }

```

- 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

```

1 // friendio.cpp
2
3 // 将插入/提取操作符函数声明为操作数类型的友元
4
5 #include <iostream>
6 #include <iomanip>
7
8 using namespace std;
9
10 class Complex {
11 public:
12     Complex(double real = 0, double imag = 0) : real(real), imag(imag)
13 {}
14 private:
15     double real, imag;
16
17     // 将插入/提取操作符函数声明为Complex类的友元
18
19     friend istream& operator>>(istream&, Complex&);
20     friend ostream& operator<<(ostream&, const Complex&);
21 };
22
23 istream& operator>>(istream& is, Complex& c) {
24     // 友元可以直接访问类的私有成员
25     return is >> c.real >> c.imag;
26 }
27
28 ostream& operator<<(ostream& os, const Complex& c) {
29     // 友元可以直接访问类的私有成员
30     return os << '(' << c.real << showpos << c.imag << noshowpos <<
31     "i)";
32 }
33
34 int main(void) {
35     Complex c;
36
37     cin >> c;
38     cout << c << endl;
39
40     return 0;
41 }

```

- 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。

```

1 // friend.cpp
2
3 // 友元函数和友元类
4
5 #include <iostream>
6 #include <iomanip>
7
8 using namespace std;
9
10 class Complex {
11 public:
12     Complex(double real = 0, double imag = 0) : real(real), imag(imag)
13     {}
14 private:
15     double real, imag;
16
17     // 友元函数
18
19     friend istream& operator>>(istream&, Complex&);
20     friend ostream& operator<<(ostream&, const Complex&);
21
22     // 友元类
23
24     friend class Sum;
25 };
26
27 istream& operator>>(istream& is, Complex& c) {
28     return is >> c.real >> c.imag;
29 }
30
31 ostream& operator<<(ostream& os, const Complex& c) {
32     return os << '(' << c.real << showpos << c.imag << noshowpos <<
33     "i)";
34 }
35 // Complex类的友元类, 可以直接访问Complex类的私有成员
36 class Sum {
37 public:
38     Sum(const Complex& c1, const Complex& c2) :
39         c(c1.real + c2.real, c1.imag + c2.imag) {}
40 private:
41     Complex c;
42
43     // 友元函数可以在声明的同时给出定义
44     friend void show(const Sum& sum) {
45         cout << sum.c << endl;
46     }
47 };
48
49
50 int main(void) {
51     Complex c1, c2;
52     cin >> c1 >> c2;
53
54     Sum sum(c1, c2);

```

```
55     show(sum);
56
57     return 0;
58 }
```

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

```
1 | L#R;
```

将被编译器解释为如下函数调用：

```
1 | 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

```
1 // membinary.cpp
2
3 // 成员函数形式的双目操作符函数
4
5 // <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>)
6
7 #include <iostream>
8 #include <iomanip>
9
10 using namespace std;
11
12 class Complex {
13 public:
14     Complex(double real = 0, double imag = 0) : real(real), imag(imag)
15     {}
16
17     // 双目操作符函数
18
19     // 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
20
21     const Complex operator+(const Complex& c) const {
22         return Complex(real + c.real, imag + c.imag);
23     }
24
25     const Complex operator-(const Complex& c) const {
26         return Complex(real - c.real, imag - c.imag);
27     }
28
29     Complex& operator+=(const Complex& c) {
30         real += c.real;
31         imag += c.imag;
32         return *this;
33     }
34
35     Complex& operator-=(const Complex& c) {
36         real -= c.real;
```

```

36     imag -= c.imag;
37     return *this;
38 }
39
40 private:
41     double real, imag;
42
43     friend istream& operator>>(istream& is, Complex& c) {
44         return is >> c.real >> c.imag;
45     }
46
47     friend ostream& operator<<(ostream& os, const Complex& c) {
48         return os << '(' << c.real << showpos << c.imag << noshowpos <<
49         "i)";
50     };
51
52 int main(void) {
53     Complex c1, c2, c3, c4;
54     cin >> c1 >> c2 >> c3 >> c4;
55
56     cout << c1 << '+' << c2 << '=' << c1 + c2 << endl;
57     cout << c1 << '-' << c2 << '=' << c1 - c2 << endl;
58
59     // (c1 + c2) = c3; // 禁止
60     // (c1 - c2) = c4; // 禁止
61
62     cout << c1 << "+=" << c2 << endl;
63     c1 += c2;
64     cout << c1 << endl;
65
66     cout << c1 << "-=" << c2 << endl;
67     c1 -= c2;
68     cout << c1 << endl;
69
70     cout << '(' << c1 << "+=" << c2 << ")=" << c3 << endl;
71     (c1 += c2) = c3; // 允许
72     cout << c1 << endl;
73
74     cout << '(' << c1 << "-=" << c2 << ")=" << c4 << endl;
75     (c1 -= c2) = c4; // 允许
76     cout << c1 << endl;
77
78     return 0;
79 }

```

- 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

```
1 | #0;
```

将被编译器解释为如下函数调用：

```
1 | 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

```
1 // memunary.cpp
2
3 // 成员函数形式的单目操作符函数
4
5 // <运算符><操作数>被编译为<操作数>.operator<运算符>()
6
7 #include <iostream>
8 #include <iomanip>
9
10 using namespace std;
11
12 class Complex {
13 public:
14     Complex(double real = 0, double imag = 0) : real(real), imag(imag)
15     {}
16
17     // 单目操作符函数
18
19     const Complex operator-(void) const {
20         return Complex(-real, -imag);
21     }
22 private:
23     double real, imag;
24
25     friend ostream& operator>>(ostream& is, Complex& c) {
26         return is >> c.real >> c.imag;
27     }
28
29     friend ostream& operator<<(ostream& os, const Complex& c) {
30         return os << '(' << c.real << showpos << c.imag << noshowpos <<
31         "j)";
32     }
33 };
34
35 int main(void) {
36     Complex c;
37     cin >> c;
38
39     cout << '-' << c << '=' << -c << endl;
```

```
40 |     return 0;
41 | }
```

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

```
1 | ++++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

```
1 | O++++; // 禁止
```

```
1 // memself.cpp
2
3 // 成员函数形式的自增减操作符函数
4
5 // ++<操作数>被编译为<操作数>.operator++()
6 // <操作数>++被编译为<操作数>.operator++(0)
7
8 // --<操作数>被编译为<操作数>.operator--()
9 // <操作数>--被编译为<操作数>.operator--(0)
10
11 #include <iostream>
12
13 using namespace std;
14
15 class Integer {
16 public:
17     Integer(int n = 0) : n(n) {}
18
19     // 自增减操作符函数
20
21     // 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
22
23     // 前++操作符函数
24     Integer& operator++(void) {
25         ++n;
26         return *this;
27     }
28
29     // 后++操作符函数
30     const Integer operator++(int) {
31         Integer i = *this;
```



```

32     ++n;
33     return i;
34 }
35
36 // 前--操作符函数
37 Integer& operator--(void) {
38     --n;
39     return *this;
40 }
41
42 // 后--操作符函数
43 const Integer operator--(int) {
44     Integer i = *this;
45     --n;
46     return i;
47 }
48
49 private:
50     int n;
51
52     friend istream& operator>>(istream& is, Integer& i) {
53         return is >> i.n;
54     }
55
56     friend ostream& operator<<(ostream& os, const Integer& i) {
57         return os << i.n;
58     }
59 };
60
61 int main(void) {
62     Integer i;
63     cin >> i;
64
65     cout << "++" << i << '=' << ++i << ", i=" << i << endl;
66     cout << i << "++=" << i++ << ", i=" << i << endl;
67
68     cout << "--" << i << '=' << --i << ", i=" << i << endl;
69     cout << i << "--=" << i-- << ", i=" << i << endl;
70
71     cout << "++++" << i << '=' << ++++i << ", i=" << i << endl; // 允许
72     cout << "----" << i << '=' << ----i << ", i=" << i << endl; // 允许
73
74     // i++++; // 禁止
75     // i----; // 禁止
76
77     return 0;
78 }

```

3.7 成员还是友元

- 一个操作符的左右操作数不一定是相同类型的对象，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

```

1 // memfriend.cpp
2
3 // 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
4 // 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
5
6 #include <iostream>
7
8 using namespace std;
9
10 class Mail;
11
12 class Mailbox { // 调用对象的类型
13 public:
14     Mailbox(const char* account) : account(account) {}
15
16     const Mailbox& operator<<(const Mail&) const; // 成员声明
17
18 private:
19     string account;
20 };
21
22 class Mail { // 参数对象的类型
23 public:
24     Mail (const char* title, const char* content) : title(title),
25     content(content) {}
26
27     friend const Mailbox& Mailbox::operator<<(const Mail&) const; // 友元声明
28
29 private:
30     string title, content;
31 };
32 // 插入操作符函数
33 const Mailbox& Mailbox::operator<<(const Mail& mail) const {
34     cout << '[' << mail.title << '|' << mail.content << "]"->" << account <<
35     endl;
36     return *this;
37 }
38
39 int main(void) {
40     Mail m1("Hello", "Good lucky for you!"), m2("Notify", "You are fired.");
41     Mailbox mb("zhangjw@tedu.cn");
42
43     mb << m1 << m2;
44
45     return 0;
46 }

```

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：

```
1 | 源类型对象.operator目标类型();
```

该函数返回一个目标类型的对象

```
1 // opconv.cpp
2
3 // 通过类型转换操作符函数，可以实现自定义类型转换
4
5 #include <iostream>
6
7 using namespace std;
8
9 class Point3D {
10 public:
11     Point3D(int x = 0, int y = 0, int z = 0) : x(x), y(y), z(z) {}
12
13 private:
14     int x, y, z;
15
16     friend ostream& operator<<(ostream& os, const Point3D& p) {
17         return os << '(' << p.x << ',' << p.y << ',' << p.z << ')';
18     }
19 };
20
21 class Point2D {
22 public:
23     Point2D(int x = 0, int y = 0) : x(x), y(y) {}
24
25     // 类型转换操作符函数
26
27     // Point2D -> Point3D
28     operator Point3D(void) const {
29         return Point3D(x, y);
30     }
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     }
38 };
39
40 void foo(Point3D p) {
41     cout << p << endl;
42 }
43
44 Point3D bar(void) {
45     Point2D p(6, 7);
46     return p; // 从函数返回过程中的类型转换
47 }
48
49 int main(void) {
50     Point2D p1(1, 2);
51     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     }

```

```

38     friend class Point3D;
39 };
40 };
41
42 Point3D::Point3D(Point2D const& p) : x(p.x), y(p.y), z(0) {}
43
44 void foo(Point3D p) {
45     cout << p << endl;
46 }
47
48 Point3D bar(void) {
49     Point2D p(6, 7);
50     return p; // 从函数返回过程中的类型转换
51 }
52
53 int main(void) {
54     Point2D p1(1, 2);
55     cout << p1 << endl;
56
57     Point3D p2 = p1; // 构造过程中的类型转换
58     cout << p2 << endl;
59
60     Point3D p3(3, 4, 5);
61     cout << p3 << endl;
62     p3 = p1; // 赋值过程中的类型转换
63     cout << p3 << endl;
64
65     foo(p1); // 向函数传参过程中的类型转换
66
67     cout << bar() << endl;
68
69     return 0;
70 }

```

- 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

```

1 // opcons.cpp
2
3 // 同时提供类型转换操作符函数和类型转换构造函数
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     // 类型转换操作符函数
33
34     // Point2D -> Point3D
35     operator Point3D(void) const {
36         cout << "Point2D::operator Point3D()" << endl;
37
38         return Point3D(x, y);
39     }
40
41 private:
42     int x, y;
43
44     friend ostream& operator<<(ostream& os, const Point2D& p) {
45         return os << '(' << p.x << ',' << p.y << ')';
46     }
47
48     friend class Point3D;
49 };
50
51 Point3D::Point3D(Point2D const& p) : x(p.x), y(p.y), z(0) {
52     cout << "Point3D::Point3D(Point2D)" << endl;
53 }
54
55 void foo(Point3D p) {
56     cout << p << endl;
57 }
58
59 Point3D bar(void) {
60     Point2D p(6, 7);
61     return p; // 从函数返回过程中的类型转换
62 }
63
64 int main(void) {
65     Point2D p1(1, 2);
66     cout << p1 << endl;
67
68     Point3D p2 = p1; // 构造过程中的类型转换
69     cout << p2 << endl;
70
71     Point3D p3(3, 4, 5);
72     cout << p3 << endl;
73     p3 = p1; // 赋值过程中的类型转换
74     cout << p3 << endl;
75

```

```

76     foo(p1); // 向函数传参过程中的类型转换
77
78     cout << bar() << endl;
79
80     return 0;
81 }

```

- 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

```

1 // expcons.cpp
2
3 // 为类型转换构造函数添加explicit修饰符
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     explicit 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     }
38
39     friend class Point3D;
40 };
41
42 Point3D::Point3D(Point2D const& p) : x(p.x), y(p.y), z(0) {}
43
44 void foo(Point3D p) {
45     cout << p << endl;
46 }
47

```

```

48 Point3D bar(void) {
49     Point2D p(6, 7);
50     // return p; // 编译错误
51     return Point3D(p);
52 }
53
54 int main(void) {
55     Point2D p1(1, 2);
56     cout << p1 << endl;
57
58     // Point3D p2 = p1; // 编译错误
59     Point3D p2(p1);
60     cout << p2 << endl;
61
62     Point3D p3(3, 4, 5);
63     cout << p3 << endl;
64     // p3 = p1; // 编译错误
65     p3 = (Point3D)p1;
66     cout << p3 << endl;
67
68     // foo(p1); // 编译错误
69     foo(static_cast<Point3D>(p1));
70
71     cout << bar() << endl;
72
73     return 0;
74 }

```

- 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

```

1 // expop.cpp
2
3 // 为类型转换操作符函数添加explicit修饰符
4
5 #include <iostream>
6
7 using namespace std;
8
9 class Point3D {
10 public:
11     Point3D(int x = 0, int y = 0, int z = 0) : x(x), y(y), z(z) {}
12
13 private:
14     int x, y, z;
15
16     friend ostream& operator<<(ostream& os, const Point3D& p) {
17         return os << '(' << p.x << ',' << p.y << ',' << p.z << ')';
18     }
19 };
20
21 class Point2D {
22 public:
23     Point2D(int x = 0, int y = 0) : x(x), y(y) {}
24
25     // 类型转换操作符函数
26

```



```

27     // Point2D -> Point3D
28     explicit operator Point3D(void) const {
29         return Point3D(x, y);
30     }
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     }
38 };
39
40 void foo(Point3D p) {
41     cout << p << endl;
42 }
43
44 Point3D bar(void) {
45     Point2D p(6, 7);
46     // return p; // 编译错误
47     return Point3D(p);
48 }
49
50 int main(void) {
51     Point2D p1(1, 2);
52     cout << p1 << endl;
53
54     // Point3D p2 = p1; // 编译错误
55     Point3D p2(p1);
56     cout << p2 << endl;
57
58     Point3D p3(3, 4, 5);
59     cout << p3 << endl;
60     // p3 = p1; // 编译错误
61     p3 = (Point3D)p1;
62     cout << p3 << endl;
63
64     // foo(p1); // 编译错误
65     foo(static_cast<Point3D>(p1));
66
67     cout << bar() << endl;
68
69     return 0;
70 }

```

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

```

1 // userptr.cpp
2
3 // 简单的智能指针
4
5 #include <iostream>
6

```

```

7  using namespace std;
8
9  class User {
10 public:
11     User(const string& name = "Anonymous", int age = -1) : name(name),
age(age) {
12         cout << "User::User() invoked" << endl;
13     }
14
15     ~User(void) {
16         cout << "User::~~User() invoked" << endl;
17     }
18
19     void print(void) const {
20         cout << *this << endl;
21     }
22
23     friend ostream& operator<< (ostream& os, const User& user) {
24         return os << '(' << user.name << ',' << user.age << ')';
25     }
26
27 private:
28     string name;
29     int age;
30 };
31
32 class UserPtr {
33 public:
34     // 类型转换构造函数
35     explicit UserPtr(User* user = NULL) : user(user) {}
36
37     // 拷贝构造函数
38     UserPtr(UserPtr& up) : user(up.release()) {}
39
40     // 拷贝赋值操作符函数
41     UserPtr& operator=(UserPtr& up) {
42         if (&up != this)
43             reset(up.release()); // 永远只有一个智能指针持有对象的地址
44         return *this;
45     }
46
47     // 析构函数
48     ~UserPtr(void) {
49         if (user)
50             delete user; // 析构时销毁对象
51     }
52
53     // 解引用操作符函数
54     User& operator*(void) const {
55         return *user;
56     }
57
58     // 间接成员访问操作符函数
59     User* operator->(void) const {
60         return user;
61     }

```

```

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),
14     age(age) {
15         cout << "User::User() invoked" << endl;
16     }

```

```

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 {

```

```

14     cout << "Square::operator()(int) invoked" << endl;
15
16     return n * n;
17 }
18
19 double operator()(double f) const {
20     cout << "Square::operator()(double) invoked" << endl;
21
22     return f * f;
23 }
24 };
25
26 int main(void) {
27     Square square;
28
29     // 在对象上应用函数语法
30
31     cout << square(2) << endl;
32     cout << square(2.5) << endl;
33
34     return 0;
35 }

```

3.11 new/delete操作符

```

1 // newdel.cpp
2
3 // new/delete操作符
4
5 #include <stdlib.h>
6
7 #include <iostream>
8
9 using namespace std;
10
11 // 全局作用域的new/delete操作符函数
12
13 void* operator new(size_t size) {
14     void* p = malloc(size);
15     cout << "::operator new(" << size << ") invoked -> " << p << endl;
16     return p;
17 }
18
19 void* operator new[](size_t size) {
20     void* p = malloc(size);
21     cout << "::operator new[](" << size << ") invoked -> " << p << endl;
22     return p;
23 }
24
25 void operator delete(void* p) {
26     cout << "::operator delete(" << p << ") invoked" << endl;
27     free(p);
28 }
29
30 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

```

```
86     double* d = new double;
87     cout << "d=" << d << endl;
88     delete d;
89
90     char* ca = new char[128];
91     cout << "ca=" << reinterpret_cast<void*>(ca) << endl;
92     delete[] ca;
93
94     A* a = new A;
95     cout << "a=" << a << endl;
96     delete a;
97
98     A* aa = new A[3];
99     cout << "aa=" << aa << endl;
100    delete[] aa;
101
102    B* b = new B;
103    cout << "b=" << b << endl;
104    delete b;
105
106    B* ba = new B[3];
107    cout << "ba=" << ba << endl;
108    delete[] ba;
109
110    return 0;
111 }
```