第十二讲

运算符重载与自动类型转换

潘建瑜@MATH.ECNU

目录页

Contents

1 运算符重载

2 自动类型转换

1 运算符重载

- 为什么要“运算符重载”
- 哪些运算符可以重载
- 如何实现运算符重载

  —— 成员函数与非成员函数
- 重载赋值运算 = 与 [ ]

运算符重载

为什么要运算符重载

预定义的运算符只针对基本数据类型，若要对类的对象进行类似的运算，需要通过重新定义运算符来实现。

- 运算符重载实质就是函数重载

  —— 对已有的运算符赋予多重含义，使得同一个运算符作用于不同类型的数据时表现出不同的行为。

```
int x=1, y=2, z;
z = x + y;              // 普通数据类型的加法
string str1("hello"), str2("Math"), str3;
str3 = str1 + str2;     // string 类对象的加法
Point A(1,2), B(3,4), C;
C = A + B;              // Point 类对象的加法
```

运算符重载基本规则

运算符重载基本规则

- 只能重载已有的运算符
- 重载不改变运算符的优先级和结合率
- 运算符重载不改变运算符的操作数的数目
- 重载的功能通常与已有的功能类似
- 重载是针对新数据类型（类）的需要，因此至少有一个操作数是新类型数据

不能被重载的运算符（4+1）	`.` `.*` `::` `?:` `sizeof`

如何实现运算符重载

- 运算符重载的一般形式 （与函数重载类似）

类型说明符 operator运算符(形参列表)

{ 函数体; } // 在类的声明中定义

类型说明符 类名::operator运算符(形参列表)

{ 函数体; } // 如果是在类外部定义

这里的类型说明符可以是类名或基本数据类型

```
Complex Complex::operator+(Complex & c2)  // 在类外部定义
{
  return Complex(real+c2.real, imag+c2.imag);
}
```

实现方式一：成员函数

通过 **成员函数** 实现运算符重载

- 运算符重载可以通过成员函数实现
- 运算符重载为成员函数时，形参个数少一个：目标对象自动作为第一个操作数/左操作数
- 如果是单目运算，则不需要形参（后置 ++ 和后置 -- 除外）

- 优点：可以自由访问本类的数据成员
- 若是双目运算，则左操作数就是目标对象本身
- 若是单目运算，则目标对象就是操作数，不需要其它对象

举例：有理数的加法

ex12_overload_member_01.cpp

```
class Rational
{
  public:
    Rational() { x=0; y=1; }
    Rational(int x, int y) { this->x=x; this->y=y; }
    Rational operator+(const Rational & p);
  private:
    int x, y;
};

Rational Rational::operator+(const Rational & p)
{
    int newx = x*p.y + y*p.x;
    int newy = y*p.y;
    return Rational(newx, newy);
}
```

对象作为形参时，通常采用引用

运算符重载：双目运算

双目运算符的重载（成员函数方式）

对象 A ⊙ 对象 B

```
类型说明符 operator⊙(const 类名 &);
```

→ 声明

```
类型说明符 类名::operator⊙(const 类名 & p)
{ 函数体; }
```

→ 定义

（假定在类外部定义）

注意：只有一个形参（对象引用，通常为 常引用）

运算符重载：前置单目运算

前置单目运算符的重载（成员函数方式）

++，--，-，！

⊙ 对象 A

类型说明符 operator⊙();

→ 声明

类型说明符 类名::operator⊙()
{ 函数体; }

→ 定义

（假定在类外部定义）

注意：不需要形参

运算符重载：后置单目运算

后置单目运算符的重载（成员函数方式）

++，--

对象 A ⊙

类型说明符 operator⊙(int);

→ 声明

类型说明符 类名::operator⊙(int dummy)
{ 函数体; }

→ 定义

（假定在类外部定义）

注意：此处形参只是为了区分前置和后置，
在运算中不起任何作用，也成为伪参数。

举例：重载单目运算符 ++

```
class Clock
{
  public:
    Clock(int hour = 0, int minute = 0, int second = 0);
    void showTime() const  { cout<<hour<<":"<<minute<<":"<<second<<endl;}
    Clock operator++();   // 前置单目运算符重载
    Clock operator++(int);    // 后置单目运算符重载
  private:
    int hour, minute, second;
};

Clock::Clock(int hour, int minute, int second)  // 构造函数
{
    if(0<=hour && hour<24 && 0<=minute && minute<60 && 0<=second && second<60)
      { this->hour = hour; this->minute = minute; this->second = second; }
    else
      cout<<"Time error!"<<endl;
}
```

举例：重载单目运算符 ++ （续）

ex12_overload_member_02.cpp

```
Clock  Clock::operator++()  // 前置单目运算符重载函数
{
   second++;
   if(second >= 60)
   { second -= 60; minute++;
     if(minute >= 60)
     { minute -= 60; hour = (++hour) % 24; }
   }
   return *this;
}

Clock Clock::operator++(int)  // 后置单目运算符重载，注意形参表中的整型参数
{
   Clock old = *this;
   ++(*this); // 调用前置++
   return old;
}
```

实现方式二：非成员函数

通过 非成员函数 实现运算符重载

- 一般在相关类中将其声明为 友元函数，以便直接访问私有数据
- 形参个数与操作数相同
- 所有操作数都通过 参数传递 实现

```
Complex operator+(const Complex & c1, const Complex & c2) // 非成员函数方式
{
    return complex(c1.real+c2.real, c1.imag+c2.imag);
}
```

举例：有理数的加法（非成员函数方式）

ex12_overload_nonmember_01.cpp

```
class Rational
{
  public:
    Rational() { x = 0; y = 1; }
    Rational(int x, int y) { this->x = x; this->y = y; }
    friend Rational operator-(const Rational &p1, const Rational &p2);
  private:
    int x, y;
};
Rational operator-(const Rational &p1, const Rational &p2)
{
    int newx = p1.x*p2.y – p1.y*p2.x;
    int newy = p1.y*p2.y;
    return Rational(newx, newy);
}
```

重载赋值运算符“ = ”

什么要重载赋值运算符

用户可以自定义复制构造函数，但赋值运算“=”仍然调用缺省复制构造函数。

如果要修改赋值运算的工作方式，则需重载赋值号。

ex12_overload_assignment.cpp

```
class Rational
{
  public:
   ... ...
    Rational & operator=(const Rational &p);  // 赋值运算是双目运算
   ... ...
};
```

- 注意：返回的通常是对象的引用，不是 void，因为 C++ 允许多重赋值，如 a=b=c

重载运算符“ [ ] ”

什么要重载 [ ]

在数组中，可以通过 [ ] 来引用指定位置的元素。

在 Rational 中，我们希望用 r[0] 表示分子，r[1] 表示分母，能否实现？

```
int Rational::operator[](int idx)
{
    if (idx == 0)
        return x;
    else
        return y;
}
```

```
Rational a(4,5);
cout << r[0] << "/" << r[1] << endl;  // OK
r[0] = 3; // ???
```

左值

什么是左值?

能出现在赋值号左边的量称为左值 (Lvalue)

怎样使得 r[0] 能出现在赋值号左边?

→→→ 返回 r[0] 的引用 （左值引用）

```
int & Rational::operator[](int idx)
{
    if (idx == 0)
        return x;
    else
        return y;
}
```

ex12_overload_left_value.cpp

重载运算符注意事项

运算符重载的注意事项

- 运算符 [ ]、=、 ->、 ( ) **必须以成员函数方式** 重载
- 运算符 <<、 >> **必须以非成员函数方式** 重载
  (这两个运算符的重载涉及到输入输出，将在文件流中介绍)
- 算术运算符和关系运算符 **建议以非成员函数方式** 重载，以便实现一些简单的自动类型转换
- 所有返回值需要出现在赋值号左边的重载，**返回的必须是一个引用**（即以左值方式返回），如赋值运算符 +=， -=，等等。

2 自动类型转换

- 为什么要“自动类型转换”
- 怎么实现自动类型转换
- 实现方式：成员函数与非成员函数
- 注意事项

为什么要自动类型转换

Q：对象与基本数据类型变量怎么运算?

Q：不同类的对象之间能不能做运算?

→ 自动（隐式）类型转换

- 对象 → 基本数据类型
- 基本数据类型 → 对象
- 对象 → 对象 （不同类的对象）

基本数据类型 → 对象

例：有理数与整型数据的加法运算

```
Rational a(1,2), b;
int c=3;
b = a + c;  // 怎么实现？
```

- 解决方法：通过构造函数，将整型数据自动转换为有理数，然后运算。

```
class Rational
{
  public:
    Rational() { x=0; y=1; }
    Rational(int x, int y) { this->x=x; this->y=y; }
    Rational(int x) { this->x=x; y=1; };
    ... ...
  private:
    int x, y;
};
```

ex12_overload_conversion_01.cpp

对象 → 基本数据类型

例：有理数与双精度数的加法运算

```
Rational a(1,2);
double b = 0.8, c;
c = a + b;  // 怎么实现？
```

▶ 解决方法：将有理数转换为双精度数，然后参与运算。

→ 重载类型转换函数

```
operator 类型转换函数名()
{ 函数体 };                    // 在类内部定义
```

```
类名::operator 类型转换函数名()
{ 函数体 };                    // 在类外部定义
```

注意：没有返回数据类型！

举例：重载类型转换函数

ex12_overload_conversion_02.cpp

```
class Rational
{
  public:
    Rational() { x=0; y=1; }
    Rational(int x, int y) { this->x=x; this->y=y; }
    operator double();
  private:
    int x, y;
};

Rational::operator double()
{
   return double(x)/y;
}
```

自动类型转换的注意点

一个类可以重载类型转换函数实现 对象到基本数据类型 的转换，也可以通过构造函数实现 基本数据类型到对象 的转换，但两者不能并存！

若重载了类型转换函数，则建议用 **非成员函数方式** 实现运算符重载，并且形参使用常引用

```
Rational operator+(const Rational & r1, const Rational & r2)
```

第十二讲上机作业

1、设计一个名为 `Complex` 的类，表示复数，这个类包括：

- 两个私有 `float` 数据成员：`x, y,` 分别表示实部与虚部；
- 一个不带形参的构造函数 `Complex()`, 用于创建缺省复数 `0,` 即 `x=y=0`
- 一个带形参的构造函数: `Complex(float x, float y)`
- 成员函数 `void Display()`, 输出复数，如 `2+3i,4-5i`
- 以成员函数方式重载复合复制运算 `+=`
- 以非成员函数方式重载复数的除法运算 `/`

实现这个类，并在主函数中测试这个类：创建复数 `z1=2.1-5.4i, z2=7.5+3.2i,` 计算 `z3=z2/z1` 和 `z3+=z1,` 并在屏幕上输出结果。（程序取名 **hw12_01.cpp**）

第十二讲上机作业

2、使 用非成员函数 方式重载复数的加法运算，使之能执行下面的运算

（程序取名 **hw12_02.cpp**）

```
Complex a(2.1,5.7), b(7.5,8), c, d;
c = a + b;
d = b + 5.6;
e = 4.1 + a;
```

3、使用 成员函数 方式重载有理数的比较运算，即 >, ==, <

（程序取名 **hw12_03.cpp**）

```
Rational a(4,5), b(2,3);

cout << "a>b? "  << (a>b ? "true" : "false") << endl;
cout << "a==b? " << (a==b ? "true" : "false") << endl;
cout << "a<b? "  << (a<b ? "true" : "false") << endl;
```

第十二讲上机作业

4、设计一个名为 `PolyInt` 的类，表示整系数多项式（即系数为整数），这个类包括：

- 一个私有 `int` 整型数据: `int n`, 表示多项式的次数，
  一个私有 `int` 整型指针: `int *a`, 表示多项式的系数，即 $p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$
  其中 `a` 指向系数向量 $[a_0, a_1, a_2, \cdots, a_n]$ 的首地址
- 不带形参的构造函数 `PolyInt()`, 用于创建 `0` 多项式 $p(x)$ =0, 即 `n=0, a[0]=0`
- 带一个形参的构造函数 `PolyInt(int n)`， 用于创建系数全部为 `0` 的 `n` 次多项式
- 带两个形参的构造函数: `PolyInt(int n, int a[])` (提示: 在构造函数中用 `new` 为对象申请存储空间)
- 成员函数 `void Display()`, 按数组方式输出多项式，如 $p(x) = 1 + 2x + 4x^3$ 输出为 `[1,2,0,4]`
- 以成员函数方式重载赋值运算 `=`，实现深度拷贝 `PolyInt & operator=(const PolyInt & p)`
- 使用非成员函数方式实现两个整系数多项式的加法运算:
  `PolyInt operator+(const PolyInt & p, const PolyInt & q)`
- 析构函数：`~PolyInt()`，释放构造函数中由 `new` 申请的存储空间。

实现这个类，并在主函数中测试这个类：创建多项式 $p(x) = 1 + 2x + 4x^3$ 和 $q(x) = 2 + 4x^2 + 3x^3 + 5x^5$，计算 `s(x)=p(x)+q(x)`，并输出 `p(x)`，`q(x)` 和 `s(x)`。

（程序取名 **hw12_04.cpp**）

第十二讲上机作业

5、用向量类实现第 4 题中的 `PolyInt`，即数据成员 `a` 是 `vector` 对象。

（程序取名 **hw12_05.cpp**）