第七章 运算符重载

模块7.2：实例应用

主讲教师：同济大学计算机科学与技术学院 陈宇飞
同济大学计算机科学与技术学院 龚晓亮

目录

- 单目运算符的重载
- 双目运算符的重载
- 流运算符的重载
- 不同类型间数据的转换

1.1 单目运算符的重载

例1：对复数的-重载，规则为实部虚部全部换符号

- 成员函数的实现：

```
Complex operator-();
```

- 友元函数的实现：

```
friend Complex operator-(Complex &a);
```

```
class Complex {
 private:
  double real;
  double imag;
public:
  Complex(double r=0, double i=0) {
     real = r; imag = i;
  }
  void display(){
  cout << real <<"+" << imag <<"i" <<endl;
  }
  Complex operator-();
};
```

```
Complex Complex::operator-() //成员函数
{  Complex c1;
   c1.real = -real;  //实部取反
   c1.imag = -imag;  //虚部取反
   return c1;
}
int main()
{  Complex c1(3,4),c2;
   c2 = -c1;
   c2.display();
}    //输出为-3+-4i形式
```

1.1 单目运算符的重载

思考右侧运算符重载程序：

```
……//略
Complex Complex::operator-() //成员函数
{
   Complex c1;
   c1.real = -real; //实部取反
   c1.imag = -imag; //虚部取反
   return c1;
}
```

- 返回类型能否为引用Complex & ？

➢分析：

若只修改函数返回类型为Complex &，不可以

因为不能返回自动变量的引用

```
……//略
Complex & Complex::operator-() //成员函数
{
   real = -real; //实部取反
   imag = -imag; //虚部取反
   return *this;
}
```

➢进一步思考：

若改为右侧程序，不可以

```
c2 = -c1;
```

-的语义不希望改变对象自身，所以要返回临时对象，而不是this

```
class Complex {
 private:
  double real;
  double imag;
public:
  Complex(double r=0, double i=0) {
     real = r; imag = i;
  }
  void display(){
  cout << real <<"+" << imag <<"i" <<endl;
  }
  friend Complex operator-(Complex &a);
};
```

```
//该友元函数返回值不可以是Complex &
Complex operator-(Complex &a) //友元函数
{   Complex c1;
    c1.real = -a.real; //实部取反
    c1.imag = -a.imag; //虚部取反
    return c1;
}
int main()
{   Complex c1(3,4),c2;
    c2 = -c1;
    c2.display();
}  //输出为-3+-4i形式
```

1.1 单目运算符的重载

例2：对++/--的前后缀运算符重载，规则为只对实部++/--，虚部不动

- 思考语义：程序中前缀++/--返回引用，后缀++/--返回对象
  - 前缀是自身先++/--，再自身参与运算，因此返回引用，即对象自身，且不需调用复制构造
  - 后缀是保存旧值，自身++/--，再旧值参与运算，因此返回对象，返回时调用复制构造产生临时对象
- 实现：
  - 前缀：正常方式
  - 后缀：多一个int型参数，不访问，仅进行区别

引申思考：i++和++i哪个效率更高？

```
class Complex {    //成员函数的实现方式
  private:
    double real;  double imag;
 public:
    Complex(double r=0, double i=0) {
        real = r; imag = i;
    }
    void display(){
        cout<<real<<"+"<<imag<<"i"<<endl;
    }
    Complex& operator++();   //前缀
    Complex  operator++(int); //后缀
};
Complex& Complex::operator++() //前缀
{   real++; //前后缀无所谓
    return *this; //返回自身
}
```

```
Complex Complex::operator++(int) //后缀
{  Complex c1(*this); //用对象自身初始化c1
   real++;    //前后缀无所谓，注意不是c1.real
   return c1; //返回++前的值，符合后缀语义
}
int main()
{  Complex c1(3,4), c2;
   c2 = c1++;
   c1.display();
   c2.display();
   c2 = ++c1;
   c1.display();
   c2.display();
}
```

```
class Complex {      //友元函数的实现方式
  private:
     double real;
     double imag;
  public:
     Complex(double r=0, double i=0) {
        real = r; imag = i;
     }
     void display(){
        cout << real <<"+" << imag <<"i" <<endl;
     }
     friend Complex& operator++(Complex &a);      //前缀
     friend Complex  operator++(Complex &a, int); //后缀
 };
Complex& operator++(Complex &a) //前缀
{    a.real++; //前后缀无所谓
     return a; //全局函数没有this指针
}
```

```
Complex operator++(Complex &a, int) //后缀
{  Complex c1(a); //用对象a初始化c1
   a.real++; //前后缀无所谓，注意不是c1.real
   return c1; //返回++前的值，符合后缀语义
}
int main()
{  Complex c1(3,4), c2;
   c2 = c1++;
   c1.display();
   c2.display();
   c2 = ++c1;
   c1.display();
   c2.display();
}
```

目录

- 单目运算符的重载
- 双目运算符的重载
- 流运算符的重载
- 不同类型间数据的转换

2.1 双目运算符的重载

例3：对复数的+重载，需考虑复数+复数，复数+double，double+复数

- 两个操作数都是对象（复数+复数）：

```
friend Complex operator+(Complex &a, Complex &b);
Complex operator+(Complex &b);
```

- 一个操作数是对象（复数+double，double+复数）：

(1) 复数+double：
```
friend Complex operator+(Complex &a, double b);
Complex operator+(double b);
```

(2) double+复数：
```
friend Complex operator+(double a, Complex &b);
```

- 双目运算符--两个操作数都是对象：友元函数、成员函数均可

```
class Complex {
    …
    friend Complex operator+(Complex &a, Complex &b);
};
Complex operator+(Complex &a, Complex &b)//友元函数
{  Complex c;
    c.real=a.real+b.real;
    c.imag=a.imag+b.imag;
    return c;
}
int main()
{  Complex c1(3,4),c2(4,5),c3;
    c3 = c1+c2;
    c3.display();
}
```

//全局友元函数，没有this指针，通过对象.成员的形式调用，返回时调用复制构造函数

• 双目运算符--两个操作数都是对象：友元函数、成员函数均可

```
class Complex {
    …
    Complex operator+(Complex &b);
};
Complex Complex::operator+(Complex &b)//成员函数
{   Complex c;
    c.real=real+b.real;
    c.imag=imag+b.imag;
    return c;
}
int main()
{   Complex c1(3,4),c2(4,5),c3;
    c3 = c1+c2;
    c3.display();
}
```

//成员函数，有this指针，直接成员的形式调用，返回时调用复制构造函数

• 双目运算符--一个操作数是对象：复数+double使用友元、成员均可

```
class Complex {
    …
    friend Complex operator+(Complex &a, double b);
};
Complex operator+(Complex &a, double b)  //友元函数
{  Complex c;
    c.real=a.real+b;//实部相加
    c.imag=a.imag; //虚部不变
    return c;
}
int main()
{    Complex c1(3,4),c2;
     c2 = c1 + 4;  //正确
     c2 = 4 + c1;  //编译错
}
```

- 双目运算符--一个操作数是对象：复数+double使用友元、成员均可

```
class Complex {
    …
    Complex operator+(double b);
};
Complex Complex::operator+(double b)   //成员函数
{   Complex c;
    c.real=real+b;//实部相加
    c.imag=imag; //虚部不变
    return c;
}
int main()
{    Complex c1(3,4),c2;
     c2 = c1 + 4;  //正确
     c2 = 4 + c1;  //编译错
}
```

- 双目运算符--一个操作数是对象：两个友元函数重载

```
class Complex { ...
    friend Complex operator+(Complex &a, double b);
    friend Complex operator+(double a, Complex &b);
};
Complex operator+(Complex &a, double b) //复数+double
{   Complex c;
    c.real=a.real+b; //实部相加
    c.imag=a.imag;  //虚部不变
    return c;
}
Complex operator+(double a, Complex &b) //double+复数
{   Complex c;
    c.real=a+b.real;//实部相加
    c.imag=b.imag; //虚部不变
    return c;
}
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

• 双目运算符--一个操作数是对象：两个友元函数重载

```
class Complex {  //引申修改1：将double换成double &
    ...
    friend Complex operator+(Complex &a, double &b);
    friend Complex operator+(double &a, Complex &b);
};
Complex operator+(Complex &a, double &b) //复数+double
{   Complex c;
    c.real=a.real+b; //实部相加
    c.imag=a.imag;  //虚部不变
    return c;
}
Complex operator+(double &a, Complex &b) //double+复数
{   Complex c;
    c.real=a+b.real;//实部相加
    c.imag=b.imag; //虚部不变
    return c;
}
```

原因：引用是变量的别名，因此若函数形参为引用，则实参只能是变量，不能是常量/表达式

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  c2 = c1 + 4; //错误
  c2.display();
  c3 = 5 + c1; //错误
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

- 双目运算符--一个操作数是对象：两个友元函数重载

```
class Complex {  //引申修改2：将double换成const double &
    ...
    friend Complex operator+(Complex &a, const double &b);
    friend Complex operator+(const double &a, Complex &b);
};
Complex operator+(Complex &a, const double &b) //复数+double
{   Complex c;
    c.real=a.real+b; //实部相加
    c.imag=a.imag;  //虚部不变
    return c;
}
Complex operator+(const double &a, Complex &b) //double+复数
{   Complex c;
    c.real=a+b.real;//实部相加
    c.imag=b.imag; //虚部不变
    return c;
}
```

若函数形参为常引用，则实参可以是变量/常量/表达式

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

- 双目运算符---一个操作数是对象：友元和成员函数重载

```
class Complex {
   ...
   Complex operator+(double b);
   friend Complex operator+(double  a, Complex &b);
};
Complex Complex::operator+(double b) //成员函数实现复数+double
{  Complex c;
   c.real=real+b; //实部相加
   c.imag=imag;  //虚部不变
   return c;
}
Complex operator+(double a, Complex &b) //友元函数实现double+复数
{  Complex c;
   c.real=a+b.real;//实部相加
   c.imag=b.imag; //虚部不变
   return c;
}
```

无法做到两个成员函数重载：
因为5+c1无法表示为成员函数形式
原因：第1个参数(左值)不是类

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

2.1 双目运算符的重载

- 关于 + 交换律的说明 ：

➢两个对象+，即定义两个类的+重载后，无论c1+c2还是c2+c1，都调用同一重载函数但本质不同：c1.operator+(c2)/c2.operator+(c1)结果相同，可以理解为交换律存在

➢对象+其它类型，例如 Complex+double，交换律不存在，交换律的表面现象是通过多个函数的重载来实现的

➢同理适用于其它存在交换律的运算符

目录

- 单目运算符的重载
- 双目运算符的重载
- 流运算符的重载
- 不同类型间数据的转换

3.1 流运算符的重载

- 形式：

```
istream& operator >> (istream &, 自定义类 &);
ostream& operator << (ostream &, 自定义类 &);
```

  - >> 和 << 已被多次重载：

    最初：位移运算符

    istream类：重载为一个输入工具

    ostream类：重载为一个输出工具

  - >> 本身是istream类(系统定义类)的成员函数，因此希望对istream类的 >> 运算符重载，使其能输入自定义类的内容
  - << 本身是ostream类(系统定义类)的成员函数，因此希望对ostream类的 << 运算符重载，使其能输出自定义类的内容

3.1 流运算符的重载

- 重载<<运算符（>>的实现类似）：

```
Time trip;  //trip是已定义的Time类对象

cout << trip;  //需要重载<<
```

  - 必须使用友元函数的方式来实现

假设：使用Time的成员函数来重载<<（Time对象将是第一个操作数）

则意味着使用形式为： trip << cout

结论：使用友元函数

3.1 流运算符的重载

- 重载<<运算符：
  - 第一种重载版本

```
void operator<<(ostream & os, const Time & t)
{
      os << t.hours << " hours, " << t.minutes << " minutes";
}
```

```
cout << trip; //可以
cout << "Trip time: " << trip << " (Tuesday)\n";   //不可以
```

ostream &

void

3.1 流运算符的重载

- 重载<<运算符：
  - 第二种重载版本

```
ostream & operator<<(ostream & os, const Time & t){
        os << t.hours << " hours, " << t.minutes << " minutes";
        return os;
}
```

```
cout << trip; //可以
cout << "Trip time: " << trip << " (Tuesday)\n";  //可以
```

ostream &

ostream &

```
//mytime3.h -- Time class with friends
class Time {
    …
    friend ostream & operator<<(ostream & os, const Time& t);
};
```

头文件
类的声明

```
//mytime3.cpp -- implementing Time methods
#include"mytime3.h"
ostream & operator<<(ostream & os, const Time& t);{…}
…
```

源程序文件
函数的实现

```
//usetime3.cpp -- using the fourth draft of the Time class
//compile usetime3.cpp and mytime3.cpp together
#include"mytime3.h"
int main()
{   Time aida(3, 35);
    Time tosca(2, 48);
    cout << aida << "; " << tosca << endl;  //operator<<
    ...
}
```

源程序文件
调用函数

完整程序见primer书

目录

- 单目运算符的重载
- 双目运算符的重载
- 流运算符的重载
- 不同类型间数据的转换

4.1 不同类型间数据的转换

- 内置类型
  - 隐式转换：

    int + double; //按照转换规则（高程基础第二章）

    int = double; //以左值为准进行转换
  - 显式转换：　（强制类型转换）

    C方式：　(int)89.5

    C++方式：　int(89.5)
- 自定义类型，能否进行类型转换？

• 例：设计一种合适的类型，以两种方式（磅和英石）来表示重量

```
// stonewt.h  -  definition for the Stonewt class
class Stonewt{
  private:
     enum {Lbs_per_stn = 14};          //类特定常量，相当于static const int
     int stone;                        Lbs_per_Stn = 14
     double pds_left;
     double pounds;
  public:                              //3种构造函数：
     Stonewt(double lbs);              Stonewt blossom(132.5);//132.5 pounds
     Stonewt(int stn, double lbs);     Stonewt buttercup(10,2);//10stone,2pounds
     Stonewt();                        Stonewt bubbles;//default value
     ~Stonewt();
     void show_lbs() const;
     void show_stn() const;
};
```

// stonewt.cpp - stonewt methods 具体程序见primer书

4.1 不同类型间数据的转换

1）转换构造函数（某类型->类类型）

```
Stonewt(double lbs); //template for double-to-Stonewt conversion
```

- 只有接受一个参数的构造函数才能作为转换函数：

```
Stonewt(int stn, double lbs); //not a conversion function
```

如果第二个参数提供默认值，可用于转换int

```
Stonewt(int stn, double lbs = 0); //int-to-Stonewt conversion
```

- 单一个参数的构造函数，只有完成转换功能才称为转换构造函数：

```
Stonewt::Stonewt(double lbs){
    stone = 0; pds_left = 0; pounds = 0;
}  //not a conversion function
```

4.1 不同类型间数据的转换

1）转换构造函数（某类型->类类型）

```
Stonewt(double lbs); //template for double-to-Stonewt conversion
```

- 隐式的自动类型转换：

```
Stonewt myCat; //create a Stonewt object
myCat = 19.6; //use Stonewt(double) to convert 19.6 to Stonewt
```

- 显式的类型强制转换：（使用关键字explicit关闭隐式自动转换）

```
explicit Stonewt(double lbs);//只能显式类型转换
myCat = 19.6; //not valid
myCat = Stonewt(19.6)  //ok, an explicit conversion
myCat = (Stonewt) 19.6  //ok, old form for explicit typecast
```

4.1 不同类型间数据的转换

1）转换构造函数（某类型->类类型）

```
Stonewt(double lbs); //template for double-to-Stonewt conversion
```

- 隐式转换的时机：

1. 将Stonewt对象初始化为double值时；

2. 将double值赋给Stonewt对象时；

3. 将double值传递给接受Stonewt参数的函数时；

4. 返回值被声明为Stonewt的函数试图返回double值时；

5. 在上述任意一种情况下，使用可转换为double类型的内置类型时；

4.1 不同类型间数据的转换

1）转换构造函数（某类型->类类型）

```
Stonewt(double lbs); //template for double-to-Stonewt conversion
```

➢允许使用Stonewt(double)构造函数来转换其他数值类型：

```
Stonewt Jumb(7000);//uses Stonewt(double), converting int to double
Jumb = 7300; //uses Stonewt(double), converting int to double
```

二步转换的前提：不存在二义性

如果还存在Stonewt(long)，则编译器将拒绝执行上述语句

因为int可以转为long或者double，一次调用会出现二义性

```
//stonewt.h  -  definition for the Stonewt class
class Stonewt {
    …
    Stonewt(double lbs);
};
```

头文件
类的声明

```
//stonewt.cpp -- Stonewt methods
#include"stonewt.h"
Stonewt::Stonewt(double lbs){…}
…
```

源程序文件
函数的实现

```
//stone.cpp  -  user-defined conversions
//compile with stonewt.cpp
#include"stonewt.h"
int main()
{   Stonewt incognito = 275;
    Stonewt wolfe(285.7);
    Stonewt taft(21, 8);
    incognito = 276.8;
    taft = 325;
    display(422, 2);
    …
}
```

源程序文件
调用函数

完整程序见primer书

• 程序分析：

```
//stone.cpp  -  user-defined conversions
int main()
{    Stonewt incognito = 275;
     Stonewt wolfe(285.7);
     Stonewt taft(21, 8);
     incognito = 276.8;
     taft = 325;
     display(422, 2);
     …
}
void display(const Stonewt & st, int n){
     …
}
```

```
//隐式
等价形式：
Stonewt incognito(275);//隐式
Stonewt incognito = Stonewt(275)//显式
等价形式相比左侧标红程序的好处：
可以接受多个参数的构造函数
```

• 程序分析：

```
//stone.cpp  -  user-defined conversions
int main()
{    Stonewt incognito = 275;
     Stonewt wolfe(285.7);
     Stonewt taft(21, 8);
     incognito = 276.8;          //将double转换为Stonewt
     taft = 325;                 //将int转换为double再转换为Stonewt
     display(422, 2);
     …
}
void display(const Stonewt & st, int n){
     …
}
```

• 程序分析：

```
//stone.cpp  -  user-defined conversions
int main()
{    Stonewt incognito = 275;
     Stonewt wolfe(285.7);
     Stonewt taft(21, 8);
     incognito = 276.8;
     taft = 325;
     display(422, 2);
     …
}
void display(const Stonewt & st, int n){
     …
}
```

查找匹配顺序：

(1)是否有Stonewt(int)？　　无

(2)是否有系统内置类型的转换？　有

int->double

(3)是否有用户定义的内部转换？　有

Stonewt(double)

//将int转换为double再转换为Stonewt

4.1 不同类型间数据的转换

2）类型转换函数（类类型->某类型）

```
operator double(); //template for Stonewt-to-double conversion
```

- 类型转换函数是用户定义的强制类型转换：(若已定义operator double();)
  - 显式使用：

```
Stonewt wolfe(285.7);
double host = double (wolfe);
double thinker = (double) wolfe;
```

  - 隐式使用：

```
Stonewt wells(20, 3);
double star = wells;
```

4.1 不同类型间数据的转换

2）类型转换函数（类类型->某类型）

```
operator double(); //template for Stonewt-to-double conversion
```

➢ 注意事项：

- 转换函数必须是类方法
- 转换函数不能指定返回类型
- 转换函数不能有参数
- 可以使用explicit关键词关闭隐式自动转换

```
//stonewt1.h -- revised definition for the Stonewt class
class Stonewt {
    …
    operator int() const;
    operator double() const;
};
```

头文件
类的声明

```
//stonewt1.cpp -- Stonewt class methods + conversion functions
#include"stonewt1.h"
Stonewt::operator int() const{…}
Stonewt::operator double() const{…}
…
```

源程序文件
函数的实现

```
//stone1.cpp  -  user-defined conversion functions
//compile with stonewt1.cpp
#include"stonewt1.h"
int main()
{   Stonewt poppins(9,2.8);
    double p_wt = poppins;
    …
    cout << "Poppins: " << int(poppins) << " pounds.\n";
    …
}
```

源程序文件
调用函数

完整程序见primer书

• 程序分析：

```
//已定义了：
    operator int() const;
    operator double() const;
//stone1.cpp
int main()
{   Stonewt poppins(9,2.8);
    double p_wt = poppins;  //隐式
    …
    cout << "Poppins: " << int(poppins) << " pounds.\n";   //显式
    …
}
```

```
//左侧程序将显式修改为隐式：
cout << "Poppins: " << poppins
     << " pounds.\n";  //不可以
原因：二义性转换
     （未指出转换为int还是double）
若只定义：operator double() const;
则正确，没有二义性
```

• 程序分析：

```
//已定义了：
    operator int() const;
    operator double() const;
//stone1.cpp
int main()
{   Stonewt poppins(9,2.8);
    double p_wt = poppins;  //隐式
    …
    cout << "Poppins: " << int(poppins) << " pounds.\n";   //显式
    …
}
```

```
//若有以下赋值：
long gone = poppins;  //不可以
原因：二义性
    （int和double均可被赋值给long）
若删除任一转换函数：
则正确，没有二义性
```

• 程序分析：

```
//已定义了：
    operator int() const;
    operator double() const;
//stone1.cpp
int main()
{   Stonewt poppins(9,2.8);
    double p_wt = poppins;  //隐式
    …
    cout << "Poppins: " << int(poppins) << " pounds.\n";   //显式
    …
}
```

```
//可使用显式的强制类型转换：
long gone = (double) poppins;
                //use double conversion
long gone = int (poppins);
                //use int conversion
```

• 程序分析：应谨慎的使用隐式转换函数

```
//手误程序：
int ar[20];
…
Stonewt temp(14,4);
…
int Temp = 1;
…
cout << ar[temp] << "!\n";
//used temp instead of Temp
```

```
//隐式的将temp转换为int，用作了数组索引（结果越界）：
结论：最好使用显式转换
      （使用explicit关键字）

class Stonewt{
   …
   explicit operator int() const;
   explicit operator double() const;
};
```

总结

- 单目运算符的重载（掌握）
- 双目运算符的重载（掌握）
- 流运算符的重载（熟悉）
- 不同类型间数据的转换（了解）