中国科学技术大学
University of Science and Technology of China

程序设计II
第10讲运算符重载与类型转换

计算机学院 黄章进
zhuang@ustc.edu.cn

内容

- 10.1 基本概念
- 10.2 输入和输出运算符
- 10.3 算术和关系运算符
- 10.4 赋值运算符
- 10.5 下标运算符
- 10.6 自增和自减运算符
- 10.7 成员访问运算符
- 10.8 函数调用运算符
- 10.9 类型转换运算符

基本概念

- 重载运算符函数的声明形式：

返回类型 operator 运算符（*形参表*）

{

// 函数体

}

Operators That May Be Overloaded

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Operators That Cannot Be Overloaded

::	.*	.	?:

基本概念

- 只能重载已有的运算符
  - 不允许重载作用域(::)、成员访问(.*与 .) 、和条件(?:)等4个运算符
  - 有四个运算符(+, -, *, &)既是一元运算符也是二元运算符
- 重载运算符的优先级和结合律和内置运算符一致
- 不能改变运算对象数量，且重载运算符的运算对象至少有一个是类类型

```
// error: cannot redefine the built-in operator for ints
int operator+(int, int);
```

基本概念

- 重载运算符函数的形参数与运算符的运算对象数一样多
  - 除了函数调用运算符operator()外，其他重载运算符不能含有默认实参
- 运算符可以重载为类的非静态成员函数
  - 成员运算符函数的（显式）形参数比运算符的运算对象数少1个
    - ✓第一个（左侧）运算对象绑定到隐式的this
- 运算符函数也可以重载为非成员函数
  - 至少有一个类类型的形参

直接调用重载的运算符函数

基本概念

- 能像调用普通函数一样直接调用非成员运算符函数

```
// equivalent calls to a nonmember operator function
data1 + data2;          // normal expression
operator+(data1, data2); // equivalent function call
```

- 也可以显式地调用成员运算符函数

```
data1 += data2;          // expression-based "call"
data1.operator+=(data2);    // equivalent call to a
member operator function
```

有些运算符不应被重载

基本概念

- 使用重载运算符本质上是函数调用，内置运算符指定的运算对象的求值顺序不适用于重载运算符
  - 逻辑与(&&)和逻辑或(||)的短路求值属性
  - 逗号(,)运算符的求值顺序
- 通常情况下，不应重载逗号(,)、取地址(&)、逻辑与(&&)和逻辑或(||)运算符
  - 重载后运算符的行为异于常态

使用与内置类型一致的含义

基本概念

- 在确定类需要哪些操作后，操作是定义成普通函数还是重载运算符呢？
- 在逻辑上与运算符相关的操作适合定义成重载的运算符：
  - 如果类执行IO操作，定义移位运算符使其与内置类型的IO保持一致
  - 如果类有检查相等性的操作，定义operator==；通常也应该有operator!=
  - 如果类有排序操作，定义operator<；通常也应有其他所有的关系运算符
  - 重载运算符的返回类型通常应与其内置版本的返回类型兼容
    - ✓逻辑和关系运算符应返回bool，算法运算符应返回类类型的值，(复合)赋值运算符应返回左侧运算对象的引用

重载为成员或非成员

基本概念

- 运算符重载为成员还是普通非成员函数的一些准则：
  - 赋值(=)、下标([])、调用(())和成员访问箭头(->)运算符必须定义为成员函数
  - 复合赋值运算符一般来说应是成员函数
  - 改变对象状态或与给定类型密切相关的运算符，如自增(++)、自减(--)和解引用(*)运算符通常应该是成员函数
  - 具有对称性的运算符可能类型转换任意一侧的运算对象，例如算术、相等性、关系和位运算符等，通常应定义为普通的非成员函数

输入和输出运算符

- IO标准库分别使用>>和<<执行输入和输出操作
  - IO库定义了用它们读写内置类型的版本
- 与iostream库兼容的输入输出运算符必须是普通的非成员函数
  - IO运算符通常需读写类的非公有数据成员，所以常声明为友元
  - 输出运算符负责打印对象的内容而非控制格式，通常不应打印换行符
  - 输入运算符必须处理一些输入可能失败的情形，而输出运算符一般不需要

书店销售数据类


```
class Sales_data {
public:
    Sales_data() {};
    Sales_data(const std::string &s, unsigned n, double p):
                bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(const std::string &s): bookNo(s) { }
    std::string isbn() const { return bookNo; }
private:
    double avg_price() const
        { return units_sold ? revenue/units_sold : 0; }
    std::string bookNo;
    unsigned units_sold;
    double revenue;
};
```

重载输出运算符<<

输入和输出运算符

- 输出运算符的第一个形参是非常量ostream对象的引用
- 第二个形参是要打印类类型的const引用
- operator<<一般返回ostream的引用

```
ostream &operator<<(ostream &os, const Sales_data
&item)
{
    os << item.isbn() << " " << item.units_sold << " "
       << item.revenue << " " << item.avg_price();
    return os;
}
```

重载输入运算符>>

输入和输出运算符

- 第一个形参是非常量istream对象的引用；第二个形参是要读入的(非常量)对象的引用；返回istream的引用

```
istream &operator>>(istream &is, Sales_data &item)
{
    double price;  // no need to initialize; we'll read into price
before we use it
    is >> item.bookNo >> item.units_sold >> price;
    if (is)        // check that the inputs succeeded
        item.revenue = item.units_sold * price;
    else
        item = Sales_data(); // input failed: give the object the default
state
    return is;
}
```

输入时的错误

- 执行输入运算符时可能发生如下错误
  - 当流含有错误类型的数据时，读取操作可能失败
  - 当读取操作到达文件末尾或者遇到输入流的其他错误时，也会失败

```
if (is)       // check that the inputs succeeded
    item.revenue = item.units_sold * price;
else
    item = Sales_data(); // input failed: give the object the default state
```

算术和关系运算符

运算符重载和类型转换

- 通常把算术和关系运算符定义为非成员函数，以允许对左侧或右侧运算对象进行类型转换
  - 形参通常都是const的引用
    - ✓算术和关系运算符不改变运算对象的状态
  - 算法运算符一般返回类类型的值
  - 关系运算符返回bool值

算术运算符

算术和关系运算符

- 算术运算符对它的两个运算对象进行计算得到一个新值
  - 新值存于局部变量，返回其副本作为结果
- 如果类定义了算术运算符，一般也会定义对应的复合赋值运算符
  - 常使用复合赋值来定义算术运算符

```
// assumes that both objects refer to the same book
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs;  // copy data members from lhs into sum
    sum += rhs;            // add rhs into sum
    return sum;
}
```

相等运算符

算术和关系运算符

- 只有当所有对应的成员都相等时，才认为两个对象相等

```
bool operator==(const Sales_data &lhs, const Sales_data
&rhs)
{
    return lhs.isbn() == rhs.isbn() &&
           lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue;
}
bool operator!=(const Sales_data &lhs, const Sales_data
&rhs)
{
    return !(lhs == rhs);
}
```

相等运算符

- 一些设计准则
  - 如果类含有判断两个对象是否相等的操作，则应把函数定义成operator==而非一个普通的命名函数
  - 如果类定义了operator==，它也应该定义operator!=
    - ✓相等和不相等运算符中的一个应把工作委托给另外一个
  - 如果类定义了operator==，则该运算符应能判断给定对象是否包含等价的数据
    - ✓相等运算符应该具有传递性

关系运算符

算术和关系运算符

- 定义了相等运算符的类也常常（但不总是）包含关系运算符
- 标准库中关联容器和一些算法要用到小于运算符，所以定义operator<会比较有用
- 通常情况下，关系运算符应该
  - 定义逻辑合理的顺序关系，使其与关联容易中对关键字的要求一致
  - 定义的顺序关系应与类中（若有）相等运算符相容
    - ✓如果两个对象是!=的，则一个应<另外一个

赋值运算符

- 赋值运算符必须定义为成员函数
  - 拷贝赋值运算符把类的一个对象赋值给该类的另一个对象
  - 类还可以定义其他赋值运算符，以使用别的类型作为右侧运算对象
  - 必须返回对*this的引用

赋值运算符

运算符重载和类型转换

- string库重载了多个赋值运算符函数

```
// illustration of assignment operators for class string
class string {
public:
    string& operator=(const string &);      // s1 = s2;
    string& operator=(const char *);        // s1 = "str";
    string& operator=(char);                // s1 = 'c';
    // ....
};
```

  - 支持如下的赋值操作：

```
string car ("Volks");
car = "Studebaker"; // string = const char*
string model;
model = 'T'; // string = char
```

复合赋值运算符

- 复合赋值运算符不要求是类的成员函数，但倾向于把包括复合赋值在内的所有赋值运算符定义在类内
  - 复合赋值运算符返回左侧运算对象的引用

```
// member binary operator: left-hand operand is bound to the implicit this pointer
// assumes that both objects refer to the same book
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```

下标运算符

- 表示容器的类通过位置来访问元素，通常会定义下标运算符operator[]
- 下标运算符必须是非静态成员函数
  - 下标形参可以是任意类型
- 下标运算符返回所访问元素的引用
  - 通常定义两个版本
    - ✓非常量版本返回普通引用
    - ✓常量版本返回常量引用

下标运算符

运算符重载和类型转换

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
    char& operator[](std::size_t n)
        { return ps[n]; }       // n should be a valid index
    const char& operator[](std::size_t n) const
        { return ps[n]; }
    // other members as before
private:
    std::string *ps;
    int    i;
};
```

自增和自减运算符

- 迭代器类通常会实现自增(++)运算符和自减(--)运算符
- C++不要求它们必须是成员函数，但常被定义为成员函数
- 通常同时定义前置版本和后置版本
  - 前置运算符返回自增或自减后对象的引用
  - 后置运算符返回对象的原值（自增或自减之前的值）

定义前置自增/自减运算符

自增和自减运算符

```
class HasPtr {
public:
    // increment and decrement
    HasPtr& operator++();       // prefix operators
    HasPtr& operator--();
    // other members as before
private:
    std::string *ps;
    int    i;
};
```

定义前置自增/自减运算符

自增和自减运算符

```
// prefix: return a reference to the incremented / decremented object
HasPtr& HasPtr::operator++()
{
    // if i already points past the end of the container, can't increment it
    ++i;       // advance the current state
    return *this;
}

HasPtr& HasPtr::operator--()
{
    // if i is zero, decrementing it will yield an invalid subscript
    --i;       // move the current state back one element
    return *this;
}
```

区分前置和后置运算符

自增和自减运算符

- 后置版本接受一个额外（不被使用）的int类型的形参
- 使用后置运算符时，编译器为这个形参提供一个值为0的实参

```
class HasPtr {
public:
    // increment and decrement
    HasPtr& operator++(int);       // postfix operators
    HasPtr& operator--(int);
    // other members as before
};
```

定义后置版本

自增和自减运算符

```
// postfix: increment/decrement the object but return the unchanged value
HasPtr HasPtr::operator++(int)
{
    HasPtr ret = *this;   // save the current value
    ++*this;     // advance one element; prefix ++ checks the increment
    return ret;  // return the saved state
}
HasPtr HasPtr::operator--(int)
{
    HasPtr ret = *this;  // save the current value
    --*this;     // move backward one element; prefix -- checks the decrement
    return ret;  // return the saved state
}
```

显式调用后置运算符

自增和自减运算符

- 后置版本必须在自增/自减前保存对象的当前状态
  - 保存后，可以调用各自的前置版本完成实际的工作
- 通过函数调用方式调用后置版本时，必须为整型形参传递一个值

```
HasPtr hp("Hello World!");
hp++;
hp.operator++(0);  // call postfix operator++
++hp;
hp.operator++();   // call prefix operator++
```

成员访问运算符

- 解引用(*)和箭头(->)运算符常用在迭代器类及智能指针类中
  - 箭头运算符必须是类的成员
  - 解引用运算符通常也是(非必须)类的成员

成员访问运算符

运算符重载和类型转换

```
class HasPtr {
public:
    std::string& operator*() const
    {
       return *ps;   // (*ps) is the string to which this object points
    }
    std::string* operator->() const
    { // delegate the real work to the dereference operator
      return & this->operator*(); // equivalent to return ps;
    }
    // other members as before
private:
    std::string *ps;
    int    i;
 };
```

箭头运算符返回值的限定

成员访问运算符

- 重载的箭头运算符必须返回指向类对象的指针或者一个重载了operator->的类的对象
  - 箭头运算符是一种特殊的二元运算符，右侧运算对象是标识符，而不是值
  - 重载时，箭头运算符看作一元后置运算符
- 根据point的类型，point->men等价于

```
(*point).mem;          // point is a built-in pointer type
point.operator->()->mem; // point is an object of class type
```

函数调用运算符

- 如果类重载了函数调用运算符()，则该类的对象称作函数对象(function object)
- 函数调用运算符必须是非静态成员函数
  - 调用运算符函数可以重载

```
struct absInt {
    int operator()(int val) const {
        return val < 0 ? -val : val;
    }
};
```

  - 使用调用运算符的方式是像函数调用一样令函数对象作用于一个实参列表

```
int i = -42;
absInt absObj;      // object that has a function-call operator
int ui = absObj(i); // passes i to absObj.operator()
```

含有状态的函数对象类

函数调用运算符

- 函数对象类除了operator()之外可以包含其他成员
  - 数据成员被用于定制调用运算符中的操作

```
class PrintString {
public:
    PrintString(ostream &o = cout, char c = ' '):
        os(o), sep(c) { }
    void operator()(const string &s) const { os << s << sep; }
private:
    ostream &os;   // stream on which to write
    char sep;      // character to print after each output
};
```

含有状态的函数对象类

函数调用运算符

- 定义PrintString的对象时，分隔符和输出流可以使用默认值也可以提供值

```
string s("Hello World!");
PrintString printer;   // uses the defaults; prints to cout
printer(s);            // prints s followed by a space on cout
PrintString errors(cerr, '\n');
errors(s);             // prints s followed by a newline on
cerr
```

类型转换运算符

- 类型转换运算符(conversion operator)是特殊的成员函数，将类类型的值转换成其他类型
- 类型转换函数的一般形式为：

  operator *type*() const;

  - *type*表示能作为函数返回类型(void除外)的任意类型
  - 没有显式的返回类型(隐含为*type*)和形参
  - 通常被定义成const成员

定义含有类型转换运算符的类

类型转换运算符

- 定义一个表示0-255间整数的类

```
class SmallInt {
public:
    SmallInt(int i = 0): val(i)    // 转换构造函数
    {
        if (i < 0 || i > 255)
            throw std::out_of_range("Bad SmallInt value");
    }
    operator int() const { return val; }
private:
    std::size_t val;
};
```

  - 定义了SmallInt和int类型间的相互转换

定义含有类型转换运算符的类

类型转换运算符

- 尽管编译器一次只能执行一个用户定义的类型转换，但隐式的用户定义转换可以置于一个标准(内置)类型转换之前或之后一起应用

```
SmallInt si;
si = 4; // implicitly converts 4 to SmallInt then calls SmallInt::operator=
si + 3; // implicitly converts si to int followed by integer addition

// the double argument is converted to int using the built-in conversion
SmallInt si = 3.14; // calls the SmallInt(int) constructor
// the SmallInt conversion operator converts si to int;
si + 3.14; // that int is converted to double using the built-in conversion
```

运算符重载基本规则

- 一元运算符可以是不带参数的成员函数或带一个参数的非成员函数
  - 对前置一元运算符@，@x可以解释为x.operator@()或operator@(x)
  - 对后置一元运算符@，x @可以解释为x.operator@(int)或operator@(x, int)
- 二元运算符可以是带一个参数的成员函数或带两个参数的非成员函数
  - 对二元运算符@，x@y可以解释为x.operator@(y)或operator@(x, y)
- 重载的运算符应尽量模拟运算符对内置类型的行为

运算符重载基本规则

- operator=、operator[]、operator()、operator->和operator *type*只能定义为成员函数
- operator->的返回值必须是一个指针或能使用->的对象
- 重载operator++和operator--时带一个int形参表示后置，不带参数表示前置
- 除operator new和operator delete外，重载的运算符参数中至少要有一个非内置数据类型