基本概念

引言

对于类的操作可以定义函数,但如果操作的逻辑和运算符类似

通常为运算符指定新的含义和转换规则,使其可以作用于类

运算符重载的实质是函数重载

函数重载通常功能类似,参数不同返回类型不同,所以运算符重载应该**尽量保留原用法和特性**

重载运算符特点

- 类似函数,但是参数必须是某个类的成员或者至少拥有一个类类型的运算对象(用于内置类型无法重载)
- 运算对象数量(**参数个数**),结合律,优先级与内置类型的运算符一致.
- 当被定义为类的成员时,类对象的隐式*this绑定到第一个运算对象 (此时运算对象数量显式不一致)
- 能像运算符一样使用,也可以函数调用的方式使用

```
data1+data2   普通表达式
operator+(data1,data2) 等价函数调用

为类的成员
data1+=data2   基于调用的表达式
data1.operator+=(data2) 成员运算符的等价调用
```

重载的形式

1. 成员函数

- 必须：赋值=、下标[]、调用()和成员访问箭头->运算符
- 一般：复合赋值运算符+=, -=、递增递减++, --和解引用*运算符 (运算对象都是类)

2. 非成员函数(友元)

- 具有对称性的运算符可能转换任意一端的运算对象，比如算数、相等性、关系和位运算符等

不应被重载的运算符

1. : : 作用域运算符在编译的时候解析
2. . 成员运算符
3. .*引用指向类成员的指针
4. ? : 条件运算符
   (**辅助记忆:1234都带点号**)
5. sizeof 许多指针都依赖sizeof

不建议重载(求值顺序):&& 和 || 和 ,

运算符--输入输出

```
<<  >>
```

1. 输入运算符需要处理输入失败的情况
2. IO运算符需要读写类的非公有数据成员,声明为友元

运算符--算术与关系

代码复用:

- 用复合赋值运算符(+=)实现算数运算符(+)
  (+相比+=性能上没有优势,可读性也是前者更好)
- 相等(==)和不相等(!=)二选一

运算符-赋值

1. 必须是**成员函数**,返回左侧对象的引用
2. 编译系统为每个类提供了默认的赋值运算符,把类的所有数据成员进行赋值操作。**但出现以下情况需要显性重载赋值运算符**

- 当用类A类型的值为类A的对象赋值，且类A的数据成员中**含有指针**或进行**析构**,由于重复释放一块内存，会导致程序崩溃
- 用非类A类型的值为类A的对象赋值时

复合赋值运算符

一般为成员函数，返回其左侧运算对象的引用

运算符-下标

1. **必须**是成员函数
2. 通常定义两个版本:一个返回普通引用,一个返回常量引用,否则无法访问const对象的任何元素

运算符-递增和递减

1. 通常是成员函数
2. 需要检查下标有效性
3. 后置相比前置多了一个参数int
4. 前置运算符返回引用,后置运算符返回值

运算符-成员访问

1. 常用于迭代器和智能指针类
2. 箭头运算符(->)必须是类的成员，解引用运算符(*)通常也是类的成员
3. 箭头运算符返回值的两种情况:

```
(*point).mem                          //point是类指针
point.operator()->mem              //point是重载了operator->类
```

若返回类型是指针，对其解引用并获取指定成员。没有该成员则编译器产生一个错误。

若返回类型是类或类的引用，则将递归应用该操作符。编译器检查返回对象是否具有成员箭头，如果有，就应用那个操作符；否则产生一个错误。这个过程继续下去，直到出现第一种情况，或者返回某些其他值,在后一种情况下，代码出错.

运算符-函数调用

1. 特点:拥有成员函数和成员变量(**具有状态**)
2. 必须是成员函数,一个类可以定义多个不同版本的调用运算符，相互在参数数量或类型上有所区别
3. 若类定义了函数调用运算符，该类的对象被称为函数对象(function object)，即调用对象类似于调用函数
4. lambda表达式被编译器翻译成一个未命名类的未命名对象,可以简化函数对象的定义
    - 通过值而非引用捕获时不含默认构造函数,使用这个对象必须提供实参
    - lambda表达式产生的类不含默认构造函数、赋值运算符以及默认析构函数；是否含有默认的拷贝/移动构造函数则通常要视捕获的数据成员类型而定

标准库定义的函数对象

1. 标准库定义了一组表示算术,关系和逻辑运算符的类，这些类都被定义成模板的形式，可以用形参指定具体的应用类型
2. 表示运算符的函数对象类常用于**替换算法中的默认运算符。**

可调用对象与function

使用map和function标准库模板(**使得可调用对象可以不同**)表示不同类型却共享调用形式(**返回对象和参数类型相同**)的可调用对象

```
map<string, function<int(int, int)>> binops;
// 可以把所有可调用对象都添加到这个map中:
binops = {
    {"+", add},                                 // 函数指针
    {"-", std::minus<int>()},                   // 标准库函数对象
    {"/", divide()},                            // 用户定义的函数对象
    {"*", [](int i, int j) {return i * j;}}, // 未命名的lambda
    {"%", mod},                                 // 命名了的lambda对象
}

// function类型重载了调用运算符，该运算符接受它自己的实参然后将其传递给存好的调用对象:
binops["+"](10, 5);  // 调用add(10, 5)
binops["-"](10, 5);  // 调用minus<int>对象的调用运算符
binops["/"](10, 5);  // 调用divide对象的调用运算符
binops["*"](10, 5);  // 调用lambda函数对象
binops["%"](10, 5);  // 调用lambda函数对象
```

类类型转换运算符

1. 必须是类的成员函数,不声明返回类型,参数列表为空，将一个类类型转换为其他类型
2. 通常不改变内容,所以定义为const成员

   ```
   operator type() const
   ```

3. 转换的类型要作为函数的返回类型，故不允许转换为数组或者函数类型，但允许转换为指针（包括数组指针以及函数指针）或者引用类型

类类型转换运算符

4. 显式的类型转换运算符必须通过显示的强制类型转换调用

```
class SmallInt{
public:
    explicit operator int() const { return val; }  // 编译器不会自动执行这一类型转换
    // ... 其他与之前一致
};
SmallInt si = 3;  // 正确：SmallInt 的构造函数不是显示的
si + 3;  // 错误：此刻需要隐式的类类型向算数类型转换，但类的运算符是显式的
static_int<int>(si) + 3;  // 正确：显示地请求类型转换
```

- 存在一个例外，如果表达式被用作条件，则编译器会将显式的类型转换自动转成隐式的执行。
- 向 bool 的类型转换通常用在条件部分，因此 operator bool() 一般定义成 explicit 的

避免有二义性的类型转换

若类中包含一个或多个类型转换，则必须确保在类类型和目标类型之间只存在唯一一种转换方式，否则，可能会有二义性

1. 不要为类提供相同的类型转换，比如A类定义了接受B类型对象的转换构造函数，同时B类定义了转换目标是A类的类型转换运算符
2. 不要定义多个转换规则，比如在类中最多只定义一个与算数类型有关的转换规则(算术类型互项转换会造成结果1)

函数匹配与重载运算符

1. 在表达式中使用重载的运算符时，无法判断正在使用的是成员函数还是非成员函数。
2. 若一个类既提供了转换目标是算数类型的类型转换，也提供了重载运算符，则将遇到重载运算符与内置运算符的二义性问题

```
class SmallInt{
friend SmallInt operator+(const SmallInt &, const SmallInt &);
public:
    SmallInt(int = 0);  // 转换源为 int 的类型转换
    operator int() const { return val; }  // 转换目标是 int 的类型转换
private:
    std::size_t val;
};
SmallInt s1, s2;
SmallInt s3 = s1 + s2;  // 正确：使用重载的 operator+
int i = s3 + 0;  // 错误：二义性错误
```