14 重载运算与类型转换

1，基本概念

重载的运算符由关键字operator和其后要定义的运算符号共同组成。

重载运算符函数的参数数量与该运算符作用的运算对象数量一样多。

如果一个运算符函数是成员函数，则它的第一个（左侧）运算对象绑定到隐式的this指针上，因此成员运算符函数的（显式）参数数量比运算符的运算对象总数少一个。

有四个符号（+,-,\*,&）既是一元运算符也是二元运算符，所有这些运算符都能被重载，从参数数量我们可以推断到底定义的是哪种运算符。

可以像调用普通函数一样直接调用运算符函数。

data1+data2和operator+(data1,data2)等价。

data1 += data2等价于data1.operator += (data2)。

如果类含有算术运算符或者位运算符，则最好也提供对应的复合赋值运算符。

具有对称性的运算符可能转换任意一端的运算对象，例如算术，相等性，关系和位运算等，因此它们通常应该是普通的非成员函数。

当我们把运算符定义成成员函数时，它的左侧运算对象必须是运算符所属类的一个对象。

例子：string u = "hi" + s;//如果+是string的成员，则错误。如果+定义成了普通的非成员函数，则正确

2，输入和输出运算符

通常情况下，输出运算符的第一个形参是一个非常量ostream对象的引用。

第二个形参一般来说是一个常量的引用，该常量就是我们想要打印的类类型。

为了与其他输出运算符保持一致，operator<<一般要返回它的ostream形参。

例子：ostream &operator<<(ostream &os, const Sales\_data &item)

输入输出运算符必须是非成员函数，而不能是类的成员函数。否则，它们的左侧运算对象将是我们的类的一个对象。

IO运算符通常需要读写类的非公有数据成员，所以IO运算符一般被声明为友元。

通常情况下，输入运算符的第一个形参是运算符将要读取的流的引用，第二个形参是将要读入到的对象的引用。

例子：istream &operator>>(istream &is, Sales\_data &item)

输入运算符必须处理输入可能失败的情况，而输出运算符不需要。

3，算术和关系运算符

我们通常把算术和关系运算符定义成非成员函数以允许对左侧或右侧的运算对象进行转换。

因为这些运算符一般不需要改变运算对象的状态，所以形参都是常量的引用。

如果类定义了算术运算符，则它一般也会定义一个对应的复合赋值运算符。

如果同时定义了算术运算符和相关的复合赋值运算符，通常使用复合赋值实现算术运算符。

如果类定义了operator==，则这个类也应该定义operator!=。

相等运算符和不等运算符中的一个应该把工作委托给另一个。

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。

4，赋值运算符

接受花括号内的元素列表作为赋值参数。

StrVec &operator=(std::initializer\_list<std::string>);

赋值运算符必须定义成类的成员，复合赋值运算符通常情况下也应该这样做。这两类运算符都应该返回左侧运算对象的引用。

5，下标运算符

下标运算符必须是成员函数。

下标运算符通常以返回元素的引用作为返回值。

如果一个类包含下标运算符，则它通常会定义两个版本：一个返回普通引用，另一个是类的常量成员并且返回常量引用。

例子：std::string& operator[](std::size\_t n)和const std::string& operator[](std::size\_t n) const

当StrVec是非常量时，我们可以给元素赋值；而当我们对常量对象取下标时，不能为其赋值。

6，递增和递减运算符

定义递增和递减运算符的类应该同时定义前置版本和后置版本。这些运算符通常应该被定义成类的成员。

后置运算符应该返回对象的原值，返回的形式是一个值而非引用。

StrBlobPtr& operator++();//前置运算符

StrBlobPtr operator++(int)//后置运算符

后置版本接受一个额外的int类型形参。我们无须使用int形参，所以无须为其命名。

7，成员访问运算符

解引用运算符(\*)和箭头运算符(->)。

例子：std::string& operator\*() const

std::string\* operator->() const

8，函数调用运算符

Struct absInt {int operator()(int val) const{}};

int i = -42;absInt absObj;int ui = absObj(i);

如果定义了调用运算符，则该类的对象被称作函数对象。

函数对象常常作为泛型算法的实参，

for\_each(vs.begin(), vs.end(), PrintString(cerr,'\n'))

8.1 lambda是函数对象

lambda是函数对象。

8.2 标准库定义的函数对象

标准库定义了一组表示算术运算符，关系运算符和逻辑运算符的类，每个类分别定义了一个执行命名操作的调用运算符。

例如，plus类定义了一个函数调用运算符用于对一对运算对象执行+操作，modulus定义一个调用运算符执行二元的%操作，equal\_to执行==，等等。

这些类都被定义成模版的形式，我们可以为其制定具体的应用类型。

plus<string>令string的加法运算符作用于string对象。

默认情况下排序算法使用operator<将序列按照升序排列。如果要执行降序排列的话，我们可以传入一个greater类型的对象。

sort(svec.begin(), svec.end(), greater<string>());

8.3 可调用对象与function

C++语言中有几种可调用对象：函数，函数指针，lambda表达式，bind创建的对象以及重载了函数调用运算符的类。

两个不同类型的可调用对象却可能共享同一种调用形式。调用形式指明了调用返回的类型以及传递给调用的实参类型。

一种调用形式对应一个函数类型。例如:int(int,int)

对于几个可调用对象共享同一种调用形式的情况，有时我们会希望把它们看成具有相同的类型。

int add(int i, int j){}

auto mod = [](int i, int j){};

struct divide{int operator()(int i, int j){}};

上面这些可调用对象，尽管它们的类型各不相同，但是共享一种调用形式：int(int,int)。

可以使用function的新的标准库类型解决上述问题，

function<int(int,int)> f1 = add;

function<int(int,int)> f2 = divede();

function <int(int,int)> f3 = [](int i, int j){};

然后定义map：map<string, function<int(int,int)>> binops;

9，重载，类型转换与运算符

9.1 类型转换运算符

类型转换运算符是类的一种特殊成员函数，它负责将一个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;

不允许转换成数组或者函数类型，但允许转换成指针或者引用类型。

一个类型转换函数必须是类的成员函数，它不能声明返回类型，形参列表也必须为空。类型转换函数通常应该是const。

class SmallInt{

public:

operator int() const{return val;}

};

SmallInt si;int num = si + 3;

在实践中，类很少提供类型转换运算符。但是，对于类来说，定义向bool的类型转换还是比较普遍的现象。

显式的类型转换运算符：explicit operator int() const{return val;}

static\_cast<int>(si) + 3;

有一个例外，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应用于它。

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。

9.2 避免有二义性的类型转换

如果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。否则的话，可能会有二义性。

最典型的例子是算数运算符，最好只定义最多一个与算术类型有关的转换规则。

9.3 函数匹配与重载运算符

如果我们对同一个类既提供了转换目标是算术类型的类型转换，也提供了重载的运算符，则将遇到重载运算符与内置运算符的二义性问题。