重载运算符与类型转换

基本概念

- 重载运算符是具有特殊名字的函数

```
[返回类型] operator[重载的运算符]（参数列表）{ }
```

  - 函数的参数数量与该运算符作用的运算对象数量相同

> 当重载的运算符是 **成员函数** 时，this绑定到左侧对象。
>
> 成员运算符函数的（显式）参数数量比运算对象的数量少一个

  - 除了重载的调用运算符operator()之外，其他重载运算符不能含义默认实参
- 对于一个运算符函数，它或者是类的成员，或者至少含有一个类类型的参数
- 当运算符作用于内置类型的运算对象时，我们无法改变该运算符的含义
- 可以重载大多数（不是全部）运算符

表 14.1：运算符

可以被重载的运算符					
+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]
不能被重载的运算符					
	::	.*	.	? :	

  - 只能重载已有的运算符，而无权发明新的运算符
  - 对于一个重载的运算符来说，其优先级和结合律和其内置运算符保持一致
  - 对于 `&&、||、，、&` 运算符，重载版本无法保留求值顺序或短路求值属性，应此也不建议重载

> 因为使用重载的运算符本质是是一次函数调用

如何确定重载的版本是类成员还是普通函数

- `=`、`[]`、`()`、`->`运算符必须是成员
- 复合赋值运算符**一般来说**应该是成员
- 给定对象状态的运算符或与给定类型密切相关的运算符，如`++、--、*`，通常应该是成员
- 具有对称性的运算符可能转换为任意一端的运算符，如算术、相等性、关系和位运算符，通常应该是普通函数

> 如果是成员函数，那么左侧的运算对象必须是所属类的一个对象

重载基础运算符

输入和输出运算符

重载输出运算符`<<`

第一个参数：非常量`ostream`的引用；

第二个参数：输出类型的常量引用;

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;
}
```

> 输出运算符尽量减少格式化操作，也不应该打印换行符

- **输入输出运算符必须是非成员函数**，但是一般被声明为友元来直接访问类内成员

  > 因为左侧是标准的输入输出成员；

重载输入运算符`>>`

第一个参数：运算符将要读取的流的引用；

第二个参数：将要读到的（非常量）对象的引用

返回某个给定流的引用

```
istream &operator>>(istream &is, Sale_data &item) {
    is>>xx>>xx;
    if (is)
        xxx;
    else
        xxx;
}
```

> 输出运算符必须处理输入可能失败的情况——从错误中恢复原来的状态

算术和关系运算符

重载的算术和关系运算符设计为非成员函数来允许左、右侧的运算对象进行转换；

参数都是常量的引用

> 因为这些运算符都不用改变运算对象的状态

- 如果类同时定义了算术运算符和相关的复合赋值运算符，则通常情况下应该使用复合赋值来实现算术运算符

相等运算符

- 如果类定义了 `operator==` ，那这个应该定义 `opeartor!=`

  > 但是不用具体实现两个，只需实现一个，另一个调用否定形式即可

  ```
  bool operator=(const T &a, const T &b) {
      ...
  }
  bool operator!=(const T&a, const T &b) {
      return !(a == b);           // 使用另一个
  }
  ```

关系运算符

> 特别地，因为关联容器和一些算法要用到小于运算符，所有定义operator<会比较有用

原则：

- 如果存在唯一一种逻辑可靠的 < 定义，则应该考虑为这个类定义<运算符；

  > 实际开发需要考虑，如果只是做题不用考虑

- 如果类同时含义 ==，则当且仅当 < 的定义和 == **产生的结果一致**时才定义< 运算符

> 如果 A == B，那么A不小于B，B也不小于;
>
> 如果 A != B， 那么要么A < B，要么 B < A;

赋值运算符

除了拷贝赋值和移动赋值运算符之外，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

```
/*
    v = {"a","b"}
*/
StrVec &StrVec::operator=(initializer_list<string> il) {
    ....
}
```

- 重载的赋值运算符必须先释放当前内存空间，在创建一片新的空间保存数据

下标运算符

- 下标运算符通常以所访问元素的引用作为返回对象
- 最好同时定义下标运算符的常量版本和非常量版本

```
string &operator[](size_t n) {
    return elements[n];
}
cosnt string &operator[](size_t n) {        // 常量版本
    return elements[n]
}
```

递增和递减运算符

> 因为递增和递减运算符改变的了所操作对象的状态，所以建议将其设置为成员函数

注意前置和后置版本都要定义

```
StrBlobPtr& StrBlobPtr::operator++() {
    // 检查给定的索引值curr是否有效
    check(curr,"已经指向容器尾后位置，不能增加");
    ++curr;
    return *this;
}

StrBlobPtr& StrBlobPtr::operator--() {
    // 直接递减，将产生一个无效下标
    --curr;
    check(curr,"往前移动");
    return *this;
}
```

- 注意到，前置和后置运算符，符号相同并且运算对象的数量和类型也相同，如何区分呢?
    - **解决：** 后置版本接受一个额外的（不被使用的）int类型的形参。当我们使用后置运算符时，编译器为这个形参提供一个值为0的实参

```
StrBlobPtr StrBlobPtr::operator++(int) {          // 后置版本，返回值
    // 无须检查有效性
    StrBlobPtr ret = *this;
    ++*this;              // 调用重载的前置版本实现重载后置递增
    ++curr;
    return ret;
}
```

> 因为不会用到int形参，所以可以不用给它名字

> 编译器分辨++i还是i++，看符号顺序就行了；占位参数只是表达了用户重载的意愿，到底是重载++i还是重载i++

成员访问运算符

> 箭头运算符必须是成员函数，解引用一般是成员函数

解引用运算符

```
class StrBlopStr {
    public:
        std::string& operator*() const {
            auto p = chekc(curr, "....");
            return (*p)[curr];
        }
}
```

箭头运算符

```
std::string* operator->() const {
    reutrn &this->operator*();  //  将实际工作委托给解引用符
}
```

- 两个运算符都定义为const成员，保证不改变调用的对象状态
- 箭头运算符返回的一定是一个成员

重载函数调用运算符——`()`

- 函数调用运算算符必须是成员函数；

  可以定义多版本，依照参数数量或类型区分

- 如果类重载了调用运算符，则该类的对象称作 **函数对象**

  > 对象的"行为像函数一样"

  - 函数对象类通常含有一些数据成员，用来定制调用运算符的操作
  - 函数对象通常作为泛型算法的实参

    ```
    for_each(vs.begin(), vs.end(), PrintString(cerr,'\n');      //
    for_each，接受一个调用对象，并对每一个元素调用该对象
    ```

lambda是函数对象

- 编写一个lambda后，编译器将该表达式翻译成一个未命名类的未命名对象；

  在lambda表达式产生的类中含有一个重载的函数调用运算符

  > - 这个重载的函数调用运算符的逻辑就是lambda表达式函数体内逻辑

- 如果通过值捕获，那么该未命名类将生成对应的数据成员，用其捕获的成员来初始化数据成员；‘

  如果是引用捕获，编译器可以直接使用该引用而无需存储

  > 因为引用捕获必须保证所引用的对象必须存在

```
[sz](const string &s) { return a.size() >= sz};

class SizeComp {    // 名字是我们取的
    SizeComp(size_t n ): sz(n) { }      // 初始化
    bool operator()(const string &s) const {
        return s.size() >= sz;      // sz必须是数据成员才行
    }
    private:
        size_t sz;
}
```

- lambda表达式产生的类不含默认构造函数、赋值运算符及默认析构函数；

  是否含默认拷贝/移动构造函数则要视捕获的数据成员类型而定

  > ？？ 按什么规则呢

标准库定义的函数对象

- 标准库定义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。

  定义在 `<function>` 中

表 14.2：标准库函数对象

算术	关系	逻辑
plus<Type>	equal_to<Type>	logical_and<Type>
minus<Type>	not_equal_to<Type>	logical_or<Type>
multiplies<Type>	greater<Type>	logical_not<Type>
divides<Type>	greater_equal<Type>	
modulus<Type>	less<Type>	
negate<Type>	less_equal<Type>	

- 这些类都被定义为模版的形式，可以指定具体的应用类型

```
std::plus<int> int_add;
int sum = int_add(10,20);       // sum = 10+20=30
```

- 表达式运算符的函数对象类常用来替换算法中的默认运算符

```
sort(a.begin(), a.begin(), greater<int>());      // 给sort传入一个可调用对象，来代替<运算符
```

- 标准库规定其函数对象对于指针同样使用适用

```
// 定义指针的比较
vector<string*> nameTable;
sort(nameTable.begin(), nameTable.end(), [](string *a, string *b) { return a < b;});
sort(nameTable.begin(), nameTable.end(), less<string*>());
```

> 正常来说，比较两个无关的指针将产生为定义的行为

可调用对象与function

> c++的可调用对象：函数；函数指针；lambda表达式，bind创建的对象；重载了函数调用运算符的类

- 和其他对象一样，可调用对象也有类型；

  > 函数及函数指针的类型有其返回值类型和实参类型决定；
  >
  > lambda表达式有它唯一的（未命名的）类类型

  但是，**不同类型可能具有相同的调用形式**。调用形式指明类调用的返回类型及传递给调用的实参类型。**一个调用形式对应一个函数类型**

```
int(int,int)          // 调用类型，接受两个int，返回一个int
```

- 标准库定义了一个名为 `function` 的新类型来存储具有存储具有特定形式的调用对象

  在头文件 `<function>` 中

操作	含义
function f	定义一个可以存储可调用对象的空function，这些可调用对象的调用形式应该与函数类型T相同
function f (nullptr)	显示构造一个空的function
function f (obj)	在f中存储可调用对象obj的副本
f	将f作为条件：当f中含有一个可调用对象时为真；否则为假
f (args)	调用f中的对象，参数是args
定义为*function*的成员的类型	
result_type	该function类型的可调用对象返回的类型
argument_type	返回第一个参数的类型
first_argument_type	返回第一个参数的类型
second_argument_typ	返回第二个参数的类型

```
function<int(int,int)>  f1;              // function是一个模版，它可以表示接受两个int,返回一个int的可调用对象

f1 = add;                // 函数指针
f1 = divide();           // 函数对象类的对象
f1 = [](int a, int b) { return a + b; }     // lambda
```

> 只要可调用对象的调用形式和function要求的相同，那f1都可以保存该可调用对象

- 不能直接将重载过的函数名存入function类型的对象中

```
int add (int a, int b) {
   return a + b;
}
Sale add (Sale a, Sale b);

function<int(int,int)> f = add;      // 不知道add是哪一个
```

  - 可以用函数指针来解决该问题

```
int (*fp) (int, int ) = add;         // 确定版本
f = add;
```

重载、类型转换与运算符

转换构造函数[1]和**类型转换运算符**共同定义了类类型转换，这样的转换有时也被称作用户定义的类型转换

- 转换构造函数：其他类型 ---> 类类型
- 类型转换函数：类类型 ---> 其他类型

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，它负责将一个类转换为其他的类型。

```
operator Type() const;
```

只要该类型可以作为函数的返回类型，就可以面向该类型（除void外）进行定义。

- 特点：
  - 没有显示的返回类型；
  - 没有形参；
  - 必须定义为类的成员函数
  - 通常应该是const
- 用户定义的类型转换，可以在标准（内置）类型转换之前或之后，并与其一起使用

> 在实践中，类很少定义类型转换运算符。
>
> 但，对于类来说，定义向bool的类转换比较常见，但是一般定义为显式的。防止bool类型转换为其他算术类型

显式的类型转换运算符

```
explicit operator int() const { return val; };

si + 3;     // 错误，si不支持隐式的转换
static_cast<int>(si) + 3;       // 正确：显示地请求类型转换
```

- 例外：如果表达式出现在下列位置，则显示的类型转换会将被隐式地执行：
  - if、while及do语句的条件部分
  - for语句头的条件部分
  - 逻辑非、或、与运算符的运算对象
  - 条件运算符的条件表达式

> 这也是为什们可以将IO对象作为条件表达式

```
while(cin >> value) {   // cin>>读完后，发生的隐式类型转换

}
```

> bool类型常用在条件部分，所以一般定义为显式的，不仅不影响其作用，而且更加安全

类型转换的二义性

- 两个类提供相同的类型转换

```
class A {
    A(const &B);        // 将B转换为A
}

class B {
    operator A() const;     // 将B转换为A
}

// 这两个用户定义的类型转换，导致B转换为A的时候不知道是由哪一个进行的
```

- 类定义了多个转换规则，而这些转换涉及的类型本身可以通过其他类型转换联系到一起

```
class A {
    operator int() const;
    operator double const;
}

long lg;
A a(lg);        // 歧义；因为long->int,和long->double 转换等级是一样的，所有区分不了
```

> 但是，如果转换等级[2]不一样，则是可以区分使用哪一个的

原则：除了显示的向bool类型的转换外，我们应尽量避免定义类型转换的函数，并尽可能限制那些“显然正确”的非显式构造函数

- 对于重载的函数，如果两个或多个类型转换都提供了同一种匹配，则会产生匹配模糊

> 编译器认为这两种类型装换是同级的

```
struct C {
    C(int);
}
struct D {
    D(int);          // 两个类都支持int默认转换构造函数
}
void manip(const C&);
void manip(const D&);    // 两个重载的函数

manip(10);       // 二义性，因为不知道这个10对应的是哪个类的转换构造函数

-----------------------------------------------------------------
// 即使在调用重载函数时，需要额外的标准类型转换，仍然会有二义性
struct D {
    D(double);
}
manip(10);       // 对应manip(C(10))或manip(D(double(10)))。看起来有不同，但是这里不考虑标准类型转换的级别
```

> 在调用重载函数时，如果需要额外的标准类型转换，则该转换的级别只有当所有可行函数**都请求于同一个类定义的类型转换**才有用
>
> ```
> class Test {
> public:
>  Test(int t) {
>      std::cout<<"int";
>  }
>  Test(double t) {
>      std::cout<<"float";
>  }
>  int a;
>
> };
> int main(){
>  Test t(1.2);        // 成立； double->int 多了一步
>   return 0;
> }
> ```

- 如果对同一个类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，则会遇到重载运算符和内置运算符的二义性问题

```
class smallInt {
```

```
public:
//    smallInt operator+(const smallInt& s) {       // 成员函数形式，也会导致下面的表达式
//        return { a + s.a};
//    }
    friend smallInt operator+(const smallInt& s, const smallInt& s2) {      // 非成员函数，所以有两个参数，并且定义为友元
        return { s2.a + s.a};
    }
    operator int() const {
        return a;
    }
       smallInt()=default;
    smallInt(int);
    int a;
};
int main(){
    smallInt s1{1};
    smallInt s2{2};
    smallInt s3;
    s3 = s1 + s2;
    int t = s1 + 2;     // 二义性，因为可以先将s1变为int,或者先将2变成smallint，两种选择
    std::cout<<t;
     return 0;
}
```

---

1. 只有一个参数的构造函数，它支持隐式的类类型转换 ↩

2. 类型转换等级，第6章 ↩