第14章 重载运算与类型转换

14.1 基本概念

1. **重载运算符**是具有特殊名字的函数：由关键字 `operator` 和其后要定义的运算符号共同组成。包含返回类型、参数列表以及函数体。
2. 除了重载的函数调用运算符 `operator()` 之外，其他重载运算符不能含有默认实参。
3. 当一个重载的运算符是成员函数时， `this` 绑定到左侧运算对象。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。
4. 无法改变**内置类型运算对象**的符号含义。
5. 只能重载已有的运算符，而**无权发明新的运算符号**。
6. 重载的运算符其**优先级**和**结合律**与对应的内置运算符保持一致。
7. 直接调用一个重载的运算符函数：

```
// 等价的表达
data1 + data2;
operator+(data1, data2);
data1.operator+(data2);
```

8. 通常情况下，不应该重载逗号 `,` 、取地址 `&` 、逻辑与 `&&` 和逻辑或 `||` 运算符。会无法保留求值顺序和/或短路属性。
9. 建议只有当操作的含义对于用户来说清晰明了时才使用运算符。如果用户对运算符可能有几种不同的理解，则使用这样的运算符将产生二义性。

14.2 输入和输出运算符

14.2.1 重载输出运算符 `<<`

1. 通常，输出运算符应该主要负责打印对象的内容而非控制格式，输出运算符不应该打印换行符。
2. 如果希望为类自定义IO运算符，则必须将其定义成非成员函数。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。

14.2.2 重载输入运算符 >>

1. 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。
2. 当读取操作发生错误时，输入运算符应该负责从错误中恢复。

14.3 算术与关系运算符

1. 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。

14.3.1 相等运算符 ==

1. 如果某个类在逻辑上有相等性的含义，则该类应该定义 `operator==`，这样做可以使得用户更容易使用标准库算法来处理这个类。

14.3.2 关系运算符 < > <= >=

1. 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。如果类同时还包含 ==，则当且仅当 < 的定义和 == 产生的结果一致时才定义 < 运算符。

14.4 赋值运算符 =

1. 可以重载赋值运算符。不论形参的类型是什么，赋值运算符都必须定义为成员函数。
2. 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。

14.5 下标运算符 []

1. 下标运算符必须是成员函数。
2. 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。

```
class StrVec{
public:
    // 普通版本
    std::string& operator[](std::size_t n)
        { return elements[n]; }
    // 常量成员，返回常量引用
    const std::string& operator[](std::size_t n) const
        { return elements[n]; }

private:
    std::string *elements;
}
```

14.6 递增和递减运算符 ++ --

1. 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。
2. 为了与内置版本保持一致，前置运算符应该返回递增或递减后对象的引用。
3. 后置版本接受一个额外的（不被使用）int 类型的形参。让编译器区分前后置版本。
4. 为了与内置版本保持一致，后置运算符应该返回对象的原值（递增或递减之前的值，需额外拷贝一份原值的副本），返回的形式是一个值而非引用。

```
class StrBlobPtr{
public:
    // 前置版本
    StrBlobPtr& operator++();
    StrBlobPtr& operator--();

    // 后置版本
    // 不会用到int形参，无须为其命名
    StrBlobPtr operator++(int);
    StrBlobPtr operator--(int);
}

StrBlobPtr p;
// 前置版本调用
++p;
p.operator++();
// 后置版本调用
p++;
p.operator++(0);
```

14.7 成员访问运算符 * ->

1. 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，尽管并非必须如此
2. 箭头运算符永远不能丢掉访问成员这个最基本的含义，当重载箭头时，可以改变的是箭头从哪个对象当中获取成员，而箭头获取成员这一事实则永远不变。
3. 重载的箭头运算符必须返回类的指针或者自定义了箭头运算符的某个类的对象， `point->mem` 的执行过程如下所示：
   (1) 如果 `point` 是指针，则应用内置的箭头运算符，表达式等价于 `(*point).mem` 。首先解引用该指针，然后从所得的对象中获取指定的成员。如果 `point` 所指的类型没有 `mem` 的成员，程序会发生错误。
   (2) 如果 `point` 是定义了 `operator->` 的类的一个对象，则使用 `point.operator->()` 的结果来获取 `mem` 。其中，如果该结果是一个指针，则执行第1步；如果该结果本身含有重载的 `operator->()` ，则重复调用当前步骤。最终，当这一过程结束时，程序或者返回了所需的内容，或者返回一些表示程序错误的信息。

14.8 函数调用运算符 （）

1. 如果类重载了函数调用运算符，则我们可以像使用函数一样使用该类的对象。
2. 函数调用运算符必须是成员函数。一个类可以定义多个不同版本的调用运算符，相互之间应该在参数数量或类型上有所区别。
3. 如果类定义了调用运算符，则该类的对象称做**函数对象（function object）**。

```
class PrintString{
public:
    PrintString(ostream& o = cout) : os(o) {}
    void operator() (const string& s) const { os << s; }
private:
    ostream& os;
}

// 像调用函数一样，调用函数对象
PrintString printer;        // 使用默认值，打印到cout
printer(s);                 // 在cout中打印s
PrintString errors(cerr);   // 打印到cerr
errors(s);                  // 在cerr中打印s

// 函数对象常常作为泛型算法的实参
// 第三个参数是类型PrintString的一个临时对象
vector<string> vs;
for_each(vs.begin(), vs.end(), PrintString(cerr));
```

14.8.1 lambda是函数对象

1. 编写一个**lambda表达式**后，编译器会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。
2. 当一个lambda表达式通过引用捕获变量时，将由程序负责确保lambda执行时引用所引的对象确实存在。因此编译器可以直接使用该引用而无须在lambda产生的类中将其存储为数据成员。
3. 相反，通过值捕获的变量被拷贝到lambda中。因此，这种lambda产生的类必须为每个值捕获的变量建立对应的**数据成员**，**同时创建构造函数**，令其使用捕获的变量的值来初始化数据成员。
4. 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。它是否含有默认拷贝函数/移动构造函数则通常要视捕获的数据成员类型而定。

```
auto wc = find_if(word.begin(), word.end(),
            [sz](const string& a)
                { return a.size() >= sz; });

// 该lambda表达式产生的类将形如：
class SizeComp{
public:
    SizeComp(size_t n) : sz(n) {}   // 该形参对应捕获的变量
    // 该调用运算符的返回值类型、形参和函数体都与lambda一致
    bool operator() (const string& s) const
        { return s.size() >= sz; }
private:
    size_t sz;  // 该数据成员对应通过值捕获的变量
}

// 等价的调用：
auto wc = find_if(word.begin(), word.end(), SizeComp(sz));
```

14.8.2 标准库定义的函数对象

1. 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。(`plus<Type>` 、 `equal_to<Type>` 、 `logical_and<Type>` 等)
2. 这些类都被定义成模板的形式，我们可以为其指定具体的应用类型，这里的类型即调用运算符的形参类型。
3. 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。

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

14.8.3 可调用对象与function

1. c++中有五种可调用对象：**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**。
2. 和其他对象一样，可调用的对象也有**类型**。例如，每个lambda有它自己唯一的（未命名）类类型；函数即函数指针的类型则由其返回值类型和实参类型决定，等等。
3. 两个不同类型的可调用对象却可能共享同一种**调用形式（call signature）**。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)` 是一个函数类型，它接受两个int、返回一个int。

```
// 下列不同类型的可调用对象，但是共享了同一种调用形式int(int, int):
// 普通函数
int add(int i, int j) { return i + j;}
// lambda，其产生一个未命名的函数对象
auto mod = [](int i, int j) { return i % j; };
// 函数对象
struct divide {
    int operator() (int i, int j) { return i / j; }
}

// 构建从运算符到函数指针的映射关系，其中函数接受两个int，返回一个int
map<string, int(*)(int, int)> binops;
binops.insert({ "+", add }); // 正确：add是一个指向正确类型函数的指针
binops.insert({ "%", mod }); // 错误：mod不是一个函数指针
```

4. 可以使用标准库的 `function` 类型解决函数指针不匹配的问题。
5. function是一个模板，当创建一个具体的function类型时我们必须提供额外的信息，在尖括号内指定类型： `function<int(int, int)>` 。

```
// 将之前的binops类型该为使用function<>
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
```

6. 不能直接将重载函数的名字存入 `function` 类型对象中：

```
int add(int i, int j) { return i + j;}
Sales_data add(const Sales_data&, const Sales_data&);
binops.insert( {"+", add} );// 错误：哪个add?

// 解决二义性问题的方法有两条
// 方法一：存储函数指针为非函数名字
int (*fp)(int, int) = add;  // 指针所指的add是接受两个int的版本
binops.insert( {"+", fp} ); // 正确：fp指向一个正确的add版本

// 方法二：使用lambda来消除二义性
binops.insert( {"+", [](int i, int j) { return add(i, j); }} );
```

14.9 重载、类型转换与运算符

1. **转换构造函数**和**类型转换运算符**共同定义了**类类型转换（class-type conversions）**，这样的转换有时也被称作**用户定义的类型转换（user-defined conversion）**。

14.9.1 类型转换运算符

1. 类型转换运算符（conversion operator）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。 `operator type() const;`

2. 类型转换函数必须是类的成员函数，它不能声明返回类型，形参列表必须为空。类型转换函数通常应该是 `const` 。
3. 类型转换函数不允许转换成数组或函数类型，但是允许转化成指针（包括数组指针及函数指针）或者引用类型。

```
class SmallInt {
public:
    // 构造函数将算术类型的值转换成SmallInt对象
    SmallInt(int i = 0) : val(i) {}
    // 类型转换运算符将SmallInt对象转换成int
    operator int() const { return val; }

private:
    std::size_t val;
}
```

4. C++11新标准引入了**显示的类型转换运算符（explicit conversion operator）**：

```
class SmallInt {
public:
    explicit operator int() const { return val; }
    // ... ...
}

SmallInt si = 3; // 正确：构造函数不是显式的
si + 3; // 错误：此处需要隐式的类型转换，但类的运算符是显式的
static_cast<int>(si) + 3; // 正确：显式地请求类型转换
```

5. 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。
6. 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。

14.9.2 避免有二义性的类型转换

1. 不要令两个类执行相同的类型转换：如果 `A` 类有一个接受 `B` 类对象的构造函数，则不要在 `B` 类中再定义转换目标是 `A` 类的类型转换运算符。

```
struct B;
struct A {
    A() = default;
    A(const B&);    // 把一个B转换成A
};
struct B {
    operator A() const; // 也是把一个B转换成A
 };

A f(const A&);
B b;

// 二义性错误：含义是f(B::operator A())
// 还是f(A::A(const B&))?
A a = f(b);

// 正确：
A a1 = f(b.operator A());
A a2 = f(A(b));
```

2. 避免转换目标是内置算术类型的类型转换。特别是当你已经定义了一个转换成算术类型的类型转换时，接下来：
   (1) 不要再定义接受算术运算符的重载运算符。如果用户需要使用这样的运算符，则类型转换操作将转换你的类型的对象，然后使用内置的运算符。
   (2) 不要定义转换到多种算术类型的类型转换。让标准类型转换完成向其他算术类型转换的工作。

```
struct A {
    // 两种算术类型的类型转化
    A(int);
    A(double);
    operator int() const;
    operator double() const;
}

void f2(long double);
A a;
// 二义性错误：含义是f(A::operator int())
// 还是f(A::operator double())?
f2(a);

long lg;
// 二义性错误：含义是A::A(int)还是A::A(double)?
A a2(lg);
```

3. 当调用重载的函数时，从多个类型转换种进行选择将变得更加复杂。如果两个或多个类型转换都提供了同一种可行匹配，则这些类型转化一样好。

4. 如果在调用重载函数时我们需要使用构造函数或者强制类型转换来改变实参的类型，则这通常意味着程序的设计存在不足。

```
struct C {
    C(int);
};
struct D {
    D(int);
};
void manip(const C&);
void manip(const D&);
// 二义性错误：含义是manip(C(10))还是manip(D(10))
manip(10);
```

5. 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同一个用户定义的类型转换时才有用。如果所需的用户定义的类型转换不止一个，则该调用具有二义性：

```
struct E {
    E(double);
}
void manip2(const C&);
void manip2(const E&);
// 二义性错误：两个不同的用户定义的类型转换都能用在此处。
// 含义是manip2(C(10)) 还是manip2(E(double(10)))
manip2(10);
```

14.9.3 函数匹配与重载运算符

1. 表达式中运算符的候选函数集既包括成员函数，也包括非成员函数。当我们在表达式中使用**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. 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，则将会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; } // 转换目标为int的类型转换
private:
    std::size_t val;
}

SmallInt s;
// 二义性错误：
// 可以把0转换成SmallInt，然后使用SmallInt的+，再转成int
// 也可以把s转成int，然后对两个int执行内置的加法运算。
int i = s + 0;
```