

第12章 使类的对象像数值 一样工作

刘 卉

huiliu@fudan.edu.cn



前言

□ 内置类型对象的行为

- 复制操作：原对象与复制对象有相同值，但相互独立。
- 提供丰富的运算符，并且在逻辑上相似的类型之间，可以自动转换。e.g. `1+2.3` // 1自动转换为1.0, `int` \Rightarrow `double`

□ 自定义类

- 适当定义复制和赋值操作，类的对象彼此独立。
- 重定义各种运算符，并定义类型转换。

前言

- 本章学习操作符重载，包括类型转换.
- 定义一个string类的简易版Str
 - 基于Vec类
 - 模仿string类，在表达式中使用 '+' 操作和类型转换.
 - 重点讨论：如何为类设计良好的接口.

```
#include "Vec.h"
```

第12章 使类的对象像数值一样工作

```
class Str {
```

```
public:
```

```
    typedef Vec<char>::size_type size_type;
```

```
    Str(){} // 1. default constructor; create an empty Str; call Vec()
```

```
    // 2. create a Str containing n copies of c
```

```
    Str(size_type n, char c): data(n, c){}
```

```
    // 3. create a Str from a null-terminated array of char
```

```
    Str(const char* cp)
```

```
{std::copy(cp, cp + std::strlen(cp), std::back_inserter(data));}
```

```
    // 4. create a Str from the range denoted by iterators b and e
```

```
    template <class In> Str(In i, In j)
```

```
{std::copy(i, j, std::back_inserter(data));}
```

```
private:
```

```
    Vec<char> data;
```

```
};
```

12.1 一个简单的string类

□ 默认构造函数

```
Str(){}
```

- 隐式调用Vec的默认构造函数，创建一个空的Str对象。

相当于 `Str(): data(){}`

- 类如果定义了其它构造函数→必须显式定义默认构造函数。

□ 创建一个新的类对象时，

step1. 系统分配内存空间，以保存该对象；

step2. 根据构造函数的初始化列表，初始化该对象；
包括成员的隐式初始化。

step3. 执行构造函数的函数体。

□ 其它构造函数

- Str(size_type n, char c): data(n, c){}

- 调用Vec类对应的构造函数来构造data.

隐式调用Vec()将data初始化为空

- Str(const char* cp){std::copy(...);}

- 构造函数初始化列表为空→数据成员data被隐式初始化为空的Vec对象.

- template <class In> Str(In i, In j) {...}

- 定义了一组构造函数→可使用不同类型的迭代器来实例化Str对象.

- 没有定义拷贝构造函数，赋值操作符和析构函数
 - 使用默认操作→调用Vec类的相关操作.
 - Str类本身并没有进行内存分配，不需要析构函数，也不需要拷贝构造函数和赋值操作符.



12.2 自动类型转换

- Str对象的行为就像值一样
 - 复制Str对象时，原对象和复制对象彼此独立。
- 隐式类型转换
 - 使用const char*构造一个Str对象

```
Str s("hello"); // constructor Str(const char* cp) called
```

- 赋值操作符需要const Str&参数

```
s = "world"; // assign a new value to s
```

const char* \Rightarrow const Str&?

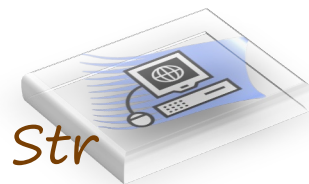
□ 其它类型对象→该类对象

- 通过带单个参数的构造函数定义.

除带默认值的参数之外, 仅有一个参数的构造函数也算.

e.g. 编译器使用构造函数"`Str(const char*)`", 根据给定字符串常量, 创建一个`Str`类的匿名对象, 然后用该对象来初始化或赋值.

```
s = "hello"; // 用Str("hello")匿名对象给s赋值
```





12.3 Str类的操作

□ string类的基本操作

`cin >> s` // use the input operator to read a string

`cout << s` // use the output operator to write a string

`s[i]` // use the index operator to access a character

`s1+s2` // use the addition operator to concatenate two strings

□ 成员函数 or 非成员函数?



索引操作符和size函数

□ public成员函数

调用Vec<char>::operator[]

```
char& operator[](size_type i) { return data[i]; }
```

```
const char& operator[](size_type i) const { return data[i]; }
```

```
size_type size() const { return data.size(); }
```

调用Vec<char>::size()



操作符重载

- 操作符的种类 & 是否成员函数 → 参数个数
 - 作为非成员函数的二元操作符：两个参数
 - 作为成员函数的二元操作符：左操作数就是调用该操作符的对象 → 一个参数(右操作数)。

```
class str{  
public:  
    str& operator+=(const str&);  
};  
str operator+(const str&, const str&);
```

```
str s1, s2;  
s1 = s1+s2; //s1=operator+(s1, s2)  
s1 += s2; //s1.operator+=(s2)
```



12.3.1 输入-输出操作符

- 在str.h中添加输入-输出操作符的声明

```
std::istream& operator>>(std::istream&, Str&);
```

```
std::ostream& operator<<(std::ostream&, const Str&);
```

- 必需是非成员函数

如果定义为成员函数:

- cin>>s等价于cin.operator>>(s)→我们无权定义cin的成员函数;
- s.operator>>(cin)等价于s>>cin, 违背标准库的规则.

□ 输出操作符

```
ostream& operator<<(ostream& os, const Str& s)
{
    for (Str::size_type i = 0; i != s.size(); ++i)
        os << s[i];
    return os;
}
```

12.3.2 友元 (Friends)

```
// this code won't compile quite yet
istream& operator>>(istream& is, Str& s)
{
    s.data.clear(); // obliterate existing value(s). compile error! data is private
    char c;
    while (is.get(c) && isspace(c));
    // read and discard leading whitespace. nothing to do, except testing the condition
    if (is) {
        // if still something to read, do so until next whitespace character
        do s.data.push_back(c); // compile error! data is private
        while (is.get(c) && !isspace(c));
        if (is) // if we read whitespace, then put it back on the stream
            is.unget();
    }
    return is;
}
```

□ 为什么编译不能通过?

- `operator>>`不是类成员，不能访问`s`的`private`数据成员.
- `operator>>`需要对`data`写入，不能通过访问器函数解决.

□ 解决方法：把'`operator>>`'声明为`Str`的友元

- 友元具有与成员相同的访问权限.

```
class Str {  
    friend std::istream& operator>>(std::istream&, Str&);  
    // as before  
};
```


- 友元声明可以出现在类定义中的任何地方
 - 友元函数是类接口的一部分。
 - 好的习惯：把友元声明放在类定义的开始位置。



12.3.3 其他二元操作符

□ operator+

- 成员函数 or 非成员函数？

操作数的类型——`const Str&`: ①连接操作不会改变任何操作数的值；②左右操作数的对称性→非成员函数

- 返回什么类型？

- 为了支持链式相加(e.g. `s1+s2+s3`)→返回`Str`对象(而不是引用).

```
Str operator+(const Str&, const Str&);
```

 注意：如定义`operator+`→也需要定义`operator+=`

□ 先实现operator+=

- 改变了左操作数→定义为类Str的public成员

```
Str& operator+=(const Str& s) {  
    std::copy(s.data.begin(), s.data.end(), std::back_inserter(data));  
    return *this;  
}
```

□ 利用operator+=实现operator+

```
Str operator+(const Str& s, const Str& t)
{
    Str r = s;    // copy constructor called
    r += t;       // operator+= called
    return r;     // copy constructor called
}
```



12.3.4 混合类型表达式

□ 连接操作的表达式中包含字符指针

```
const std::Str    greeting = "Hello, " + name + "!";
```

- 字符指针通过Str(const char*)隐式转换为Str类型，再调用operator+. 与下列操作等价：

```
Str temp1("Hello, "); //Str::Str(const char*)
```

```
Str temp2 = temp1 + name; //operator+(const Str&, const Str&), Str::Str(const Str&)
```

```
Str temp3("!") //Str::Str(const char*)
```

```
Str greeting = temp2 + temp3; //operator+(const Str&,const Str&)
```

- 多个临时对象→开销很大.
- 在商用string库的实现中, 并不依赖自动类型转换来实现混合类型的表达式计算.



12.3.5 定义二元操作符

在二元操作符的设计中, 理解类型转换的意义很重要

- 如果类支持类型转换→把二元操作符定义为非成员函数→保持操作数之间的对称性.
 - 1) 操作符是类的成员→左操作数不能是自动类型转换的结果→左右操作数不对称. e.g. `Str greeting = "Hi"+name`不能运行
 - 2) 操作符是非成员函数→操作数可以是任何类型, 只要能转换成参数类型→左右操作数对称. e.g. `Str greeting = "Hi"+name`可以运行

□ 复合赋值运算符

- 左操作数必须限定为本类对象，不能是自动类型转换的结果。
- 与赋值操作符相同，所有复合赋值操作符都必须定义为类的成员。

```
Str s1;
```

```
.....
```

```
s1 = s1 + "Test";    // s1 = "Test" + s1    ✓
```

```
s1 += "Test";       // "Test" += s1    ✗
```




12.4 某些类型转换是危险的

□ 避免自动类型转换的方法

- 在带一个参数的构造函数前添加explicit→只能在显式构造对象时，使用该构造函数。

e.g. 在Vec定义中，

```
explicit Vec(size_type n, const T& t=T()){create(n, t);}
```

如果没有explicit声明，则

```
Vec<string> p = frame(42); //用户希望对数字42加框
```

会将42隐式转换为Vec(42)，其结果是对42个空行加框。



何时使用explicit声明?

构造函数如果用来定义对象的内容→自动类型转换.

构造函数如果用来定义对象的结构→explicit



12.5 类型转换操作符

- 该类对象→其它类型
 - 必须被定义为类的成员.
 - 定义形式
operator 目标类型

[例] 求一组Student_info对象的平均成绩.

解决方法: 把Student_info对象转换为double值.

```
class Student_info {  
public:  
    operator double() const { return grade(); }  
};  
double sum = 0;  
for (size_t i = 0; i != students.size(); ++i)  
    sum += students[i]; // students[i] is automatically converted to double  
cout << "Average grade: " << sum/students.size() << endl;
```

```
if (cin >> x) {  
    .....  
}
```

标准库定义了 `istream::operator void*`，而 `void*` 可以转换为 `bool` 值：

`cin → void* → bool`



12.6 类型转换和内存管理

- 需要提供从Str到字符数组(' \0'结束)的转换吗?

```
class Str {  
public:  
    // plausible, but problematic conversion operations  
    operator char*(); // added  
    operator const char*() const; // added  
    // as before  
private:  
    Vec<char> data;  
};
```

```
Str s;  
// ...  
ifstream in(s);  
//wishful thinking: converts and  
//then open the stream named s
```

❑ 充满了内存管理的缺陷

```
operator char*();  
operator const char*() const;
```

■ 不能直接返回data

- ❑ 类型不匹配：data是Vec<char>型，需要返回char型数组。
- ❑ 即使类型匹配，返回data会破坏Str类的封装性。

■ 返回data的副本——分配新的空间

- ❑ 需要用户管理这块空间。
- ❑ 类型转换隐式发生时，用户没有指针可销毁——内存泄漏

```
Str s;
```

```
ifstream is(s); //implicit conversion—how can we free the array?
```

□ 标准string库采用的方法

- 通过显式方法(成员函数)，得到字符串的副本.

- 1) `c_str()`: 把当前string对象的内容复制到以'\0'结束的字符数组中，该数组属于string类.
- 2) `data()`: 与`c_str()`类似，返回的数组不以'\0'结束.
- 3) `copy(char*, int)`: 把整数个字符复制到char*指向的空间，该空间由用户来分配和释放.



小结

□ 类型转换

- 其它类型→本类型：通过非explicit的构造函数(仅带一个参数)定义.
- 本类型→其它类型：通过类型转换操作符定义，operator type-name(), 必须是成员函数.
- 如果两个类都定义了到彼此的类型转换，会引起二义性.

```
class A{
    A(B b){...} //B→A
    operator B(){...} //A→B
};

A a; B b;
a+b; //a→B()还是b→A()?
```

□ 友元声明

- 一般放在类定义的开始.
- 类的友元可以访问其私有成员.

```
template<class T>
class Thing {
    friend std::istream& operator>>(std::istream&, Thing&);
    // ...
};
```

- 类也可以作为友元.

□ 模板成员函数

- 类本身可以是模板/非模板.
- 包含模板成员函数的类, 相当于同时拥有很多同名的成员函数.
- 声明和定义与普通模板函数相同.

