OOP 之 运算符重载

目录：

运算符重载：

通过自定义运算符，程序员可以自定义类的操作。

运算符的重载有成员函数和友元两种形式。有的运算符可以选择任意一种实现，有的则必须使用友元函数的形式。

这里通过 String 类的编写，讲述常见的几个运算符的重载。

1.赋值运算符=

这个我们在复制控制一节写过，如下：

```
String &String::operator=(const String &s) {

    if (s != *this) {
        delete[] _str;
        _str = NULL;
```

```
        _str = new char[s.size() + 1];
        strcpy(_str, s.c_str());
    }
    return *this;
}
```

再次强调这里的三个要点：

1.要注意判断自身赋值的情况

2.要返回自身的引用

3.赋值运算符必须重载为成员函数的形式

这个运算符还可以接受 char *类型的参数

```
String &String::operator=(const char *str) {
    delete[] _str;
    _str = NULL;
    _str = new char[strlen(str) + 1];
    strcpy(_str, str);
    return *this;
}
```

2.复合赋值运算符+=

跟=类似，也必须重载为成员函数的形式

```
String &String::operator+=(const String &s) {

    char *tmp = new char[size() + s.size() + 1];
    strcpy(tmp, _str);
    strcat(tmp, s._str);

    delete[] _str;
    _str = NULL;

    _str = tmp;
    return *this;
}
String &String::operator+=(const char *s) {
    *this += String(s);
```

```
    return *this;
}
```

注意这里，重载的第二个+=运算符调用了第一个重载版本，这是避免重复代码的体现。

3.输出运算符 <<

输出运算符必须重载为友元函数的形式。因为<<运算符的第一个操作数必须为 IO 流

String 的输出运算符如下：

```
friend std::ostream &operator<<(std::ostream &os, const String &s);
```

这是类内部的友元声明

```
inline std::ostream &operator<<(std::ostream &os, const String &s) {
    os << s._str;
    return os;
}
```

注意输出运算符最后不要换行，把是否换行的权利交给使用类的用户

4.输入运算符>>

这个与输出运算符类似，但是参数不加 const，而且要注意 IO 失败的情况

类内部的声明：

```
friend std::istream &operator>>(std::istream &is, String &s);
```

函数实现：

```
inline std::istream &operator>>(std::istream &is, String &s) {
    char buf[1024];
    is >> buf;
    if (is) {
        s = buf;
    }
    return is;
}
```

5.加法运算符 +

加法运算符既可以重载为成员函数的形式，也可以重载为友元函数的形式。

但是重载为成员函数有个限制，我们知道标准库中的 string 可以实现下面三种加法：

A)string + char *
B)String + string
C)Char * + string

而采用成员函数的形式，限定了第一个操作符必须为 string 类型，所以我们只能实现上面两种运算。

所以最好的办法是采用友元函数重载的形式。

```
friend String operator+(const String &, const String &);
friend String operator+(const String &, const char *);
friend String operator+(const char *, const String &);
```

三个函数的实现如下：

```
inline String operator+(const String &lhs, const String &rhs) {
    String ret(lhs);
    ret += rhs;
    return ret;
}
```

```
inline String operator+(const String &lhs, const char *s) {
    return lhs + String(s);
}

inline String operator+(const char *s, const String &rhs) {
    return String(s) + rhs;
}
```

这里第一个+利用了已经重载的+=，而后面两个+运算符都利用了第一个加号。需要注意的是，加法运算法产生一个新的对象，参数代表的对象不会发生任何改变，所以返回的是一个局部变量

6.比较运算符 < <= >= > == !=

这些运算符都是对称形状的，所以最好采用友元函数的形式

String 的这些运算符如下：

```
friend bool operator==(const String &, const String &);
    friend bool operator!=(const String &, const String &);

    friend bool operator<(const String &, const String &);
    friend bool operator>(const String &, const String &);
    friend bool operator<=(const String &, const String &);
    friend bool operator>=(const String &, const String &);
```

实现如下：

```
inline bool operator==(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) == 0;
}
inline bool operator!=(const String &lhs, const String &rhs) {
    return !(lhs == rhs);
}

inline bool operator<(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) < 0;
}
```

```
inline bool operator>(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) > 0;
}
inline bool operator<=(const String &lhs, const String &rhs) {
    return !(lhs > rhs);
}
inline bool operator>=(const String &lhs, const String &rhs) {
    return !(lhs < rhs);
}
```

7.下标操作符[]

下标运算符需要处理越界的情况，而且最后重载 const 和非 const 两个版本，返回值为引用，而不是局部变量（为什么？）

先看一个例子：

```
#include <vector>
#include <iostream>

using namespace std;

class Foo
{
public:
    int &operator[](std::size_t index)
    {
        return _data[index];
    }
    const int &operator[](std::size_t index) const
    {
        return _data[index];
    }

    void init()
    {
        for(size_t ix = 0; ix != 100; ++ix)
        {
            _data.push_back(ix);
        }
    }
```

```
private:
    vector<int> _data;
};

int main(int argc, char **argv) {

    Foo foo;
    foo.init();

    cout << foo[23] << endl;
}
```

String 类相应的实现如下：

```
char &String::operator[](std::size_t index) {
    return _str[index];
}
const char &String::operator[](std::size_t index) const {
    return _str[index];
}
```

通过实现 String 类总结运算符重载的原则：

A)= += 必须写成成员函数，而且返回值必须为自身引用

B)[]运算符必须为成员函数，而且有 const 和非 const 两个版本，分别可作为右值和左值

C)+ == != < <= > >=这些算术和关系运算符，最好定义为 friend 的形式

D)>> <<必须为 friend 形式，而且输入操作要处理错误的情况

E)有些操作要配合使用，例如重载了+，就最好重载+=，重载了==，就去重载!=

F)重载运算符应尽可能使用其他运算符的操作。比如!=使用==，+使

用了+=的操作

String 类的完整代码如下：

头文件 h

```
#ifndef STRING_H_
#define STRING_H_

#include <string.h>
#include <cstddef>
#include <iostream>

class String {
public:
    String();
    String(const char *);
    String(const String&);
    ~String();
    String &operator=(const String &);
    String &operator=(const char *);

    String &operator+=(const String &);
    String &operator+=(const char *);

    char &operator[](std::size_t index);
    const char &operator[](std::size_t index) const;

    std::size_t size() const;
    const char* c_str() const;
    void debug();

    friend String operator+(const String &, const String &);
    friend String operator+(const String &, const char *);
    friend String operator+(const char *, const String &);

    friend bool operator==(const String &, const String &);
    friend bool operator!=(const String &, const String &);

    friend bool operator<(const String &, const String &);
    friend bool operator>(const String &, const String &);
    friend bool operator<=(const String &, const String &);
```

```
    friend bool operator>=(const String &, const String &);

    friend std::ostream &operator<<(std::ostream &os, const String &s);
    friend std::istream &operator>>(std::istream &is, String &s);

private:
    char *_str;
};

inline String operator+(const String &lhs, const String &rhs) {
    String ret(lhs);
    ret += rhs;
    return ret;
}

inline String operator+(const String &lhs, const char *s) {
    return lhs + String(s);
}

inline String operator+(const char *s, const String &rhs) {
    return String(s) + rhs;
}
inline bool operator==(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) == 0;
}
inline bool operator!=(const String &lhs, const String &rhs) {
    return !(lhs == rhs);
}

inline bool operator<(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) < 0;
}
inline bool operator>(const String &lhs, const String &rhs) {
    return strcmp(lhs._str, rhs._str) > 0;
}
inline bool operator<=(const String &lhs, const String &rhs) {
    return !(lhs > rhs);
}
inline bool operator>=(const String &lhs, const String &rhs) {
    return !(lhs < rhs);
}

inline std::ostream &operator<<(std::ostream &os, const String &s) {
    os << s._str;
```

```
    return os;
}

inline std::istream &operator>>(std::istream &is, String &s) {
    char buf[1024];
    is >> buf;
    if (is) {
        s = buf;
    }
    return is;
}

#endif /* STRING_H_ */
```

CPP 文件如下：

```
#include <iostream>
#include "String.h"

String::String() :
        _str(new char[1])         //""
{
    *_str = '\0';    // ""
}
String::String(const char *s) {
    _str = new char[strlen(s) + 1];
    strcpy(_str, s);
}
String::String(const String &s) {
    _str = new char[s.size() + 1];
    strcpy(_str, s.c_str());
}
String::~String() {
    delete[] _str;
    _str = NULL;
}
String &String::operator=(const String &s) {

    if (s != *this) {
        delete[] _str;
        _str = NULL;
        _str = new char[s.size() + 1];
        strcpy(_str, s.c_str());
```

```
    }
    return *this;
}

String &String::operator=(const char *str) {
    delete[] _str;
    _str = NULL;
    _str = new char[strlen(str) + 1];
    strcpy(_str, str);
    return *this;
}

String &String::operator+=(const String &s) {

    char *tmp = new char[size() + s.size() + 1];
    strcpy(tmp, _str);
    strcat(tmp, s._str);

    delete[] _str;
    _str = NULL;

    _str = tmp;
    return *this;
}
String &String::operator+=(const char *s) {
    *this += String(s);
    return *this;
}

char &String::operator[](std::size_t index) {
    return _str[index];
}
const char &String::operator[](std::size_t index) const {
    return _str[index];
}

std::size_t String::size() const {
    return strlen(_str);
}
const char* String::c_str() const {
    return _str;
}
void String::debug() {
    std::cout << _str << std::endl;
```

}

下面通过一个简易的智能指针的编写来展示如何重载成员操作运算符

代码如下：

```
#ifndef SMARTPTR_H_
#define SMARTPTR_H_

#include <string>
#include <iostream>

class Student {
public:
	Student(int id, const std::string &name) :
			_id(id), _name(name) {
		std::cout << "Create Student" << std::endl;
	}
	~Student() {
		std::cout << "Destroy Student" << std::endl;
	}

	void print() {
		std::cout << _id << " : " << _name << std::endl;
	}

private:
	int _id;
	std::string _name;
};

class SmartPtr {
public:
	SmartPtr();
	SmartPtr(Student *ptr);
```

```
    ~SmartPtr();

    void reset_ptr(Student *ptr);
    const Student *get_ptr() const;

    Student *operator->();
    const Student *operator->() const;

    Student &operator*();
    const Student &operator*() const;

private:
    Student *_ptr;

private:
    //prevent copy
    SmartPtr(const SmartPtr &);
    SmartPtr &operator=(const SmartPtr &);
};

#endif /* SMARTPTR_H_ */
```

CPP 文件如下：

```
#include "SmartPtr.h"

SmartPtr::SmartPtr() :
        _ptr(NULL) {

}
SmartPtr::SmartPtr(Student *ptr) :
        _ptr(ptr) {

}
SmartPtr::~SmartPtr() {
    delete _ptr;
}

void SmartPtr::reset_ptr(Student *ptr) {
    if (ptr != _ptr) {
        delete _ptr;
        _ptr = ptr;
    }
}
```

```
const Student *SmartPtr::get_ptr() const {
	return _ptr;
}

Student *SmartPtr::operator->() {
	return _ptr;
}
const Student *SmartPtr::operator->() const {
	return _ptr;
}

Student &SmartPtr::operator*() {
	return *_ptr;
}
const Student &SmartPtr::operator*() const {
	return *_ptr;
}
```

自增和自减运算符

```
#ifndef INTEGER_H_
#define INTEGER_H_

#include <iostream>

class Integer {
public:
	Integer(int num);
	~Integer();

	friend std::ostream &operator<<(std::ostream &os, const Integer &obj);
	Integer &operator++();
	Integer operator++(int);

private:
	int _num;
};

inline std::ostream &operator<<(std::ostream &os, const Integer &obj) {
	os << obj._num;
```

```
    return os;
}

#endif /* INTEGER_H_ */
```

CPP 文件

```
#include "Integer.h"

Integer::Integer(int num) :
        _num(num) {

}
Integer::~Integer() {

}

Integer &Integer::operator++() {
    ++_num;
    return *this;
}

Integer Integer::operator++(int) {
    Integer ret(*this);
    ++(*this);
    return ret;
}
```

练习 Complex 类的编写：

实现一个复数类，可以进行简单的加减乘除操作，可以打印以及输出。

函数对象

假设我们有一个单词的容器，现在我们需要去统计里面长度在 6 个字符以上的单词的个数。

我们定义这样的一个函数：

```
 bool GT6(const string &s){
    return s.size() >= 6;
}
```

然后采用标准库的算法来统计单词数目如下：

```
vector<string>::size_type wc = std::count_if(words.begin(), words.end(), GT6);
```

这样做当然是对的，但是我们发现一个严重的问题，我们写的 GT6 仅能统计长度在 6 以上的单词，如果我需要统计 10 和 8，那么我还需要再次编写函数。

一种理想的情况是我们可以把比较的数目也当做参数传入。解决办法就是函数对象。

```
class GT {
public:
    GT(size_t val = 0) :
            bound_(val) {
    }
    bool operator() (const string &s){
        return s.size() >= bound_;
    }
private:
    std::string::size_type bound_;
};
```

此时我们就可以这样来调用：

```
vector<string>::size_type wc = std::count_if(words.begin(), words.end(),
            GT(6));
```

这样便很大程度上提高了灵活性。