

实验 4：类的运算符重载

姓名__徐佳辉__班级__计科 1902__学号__201906080621__

- 请阅读此说明：实验 4 满分 110 分；其中 10 分为附加，可选做；做完实验后请按要求将代码和截图贴入该文档。然后将此文档、源代码文件（.hpp, .cpp）打包上传到学习通。

1、（总分 15 分）课堂例题巩固。

● 实验要求：

- 1) 运行文件夹“4-1 static”中的两个程序，体会 static 的作用。（5 分）
- 2) 装配并运行课程 ppt 上的代码，并思考：❶ 如果不将 average 函数定义为静态成员函数行不行？程序能否通过编译？需要作什么修改？（5 分）❷ 为什么要用静态成员函数？请分析其理由。（5 分）

● ppt 附图：

第8讲 类与对象的进一步讨论
-其他

❖ 静态成员

- 静态成员例：
include <iostream>
using namespace std;
class Student //定义Student类
{
public:
student(int n,int a,float s):num(n),age(a),score(s){}
//定义构造函数
void total();
static float average(); //声明静态成员函数

第8讲 类与对象的进一步讨论
-其他

❖ 静态成员

- 静态成员例：
private:
int num;
int age;
float score;
static float sum; //静态数据成员
static int count; //静态数据成员
};
void Student::total() //定义非静态成员函数
{ sum+=score; //累加总分
count++; //累计已统计的人数
}

第8讲 类与对象的进一步讨论
-其他

❖ 静态成员

- 静态成员例：
float Student::average() //定义静态成员函数
{ return (sum/count);}
float Student::sum=0; //对静态数据成员初始化
int Student::count=0; //对静态数据成员初始化

int main()
{ Student stud[3]={ //定义对象数组并初始化
Student(1001,18,70),Student(1002,19,78),Student(1005,20,98)};
int n;
cout <<"please input the number of students:";
cin>>n; //输入需要前面多少名学生的平均成绩

第8讲 类与对象的进一步讨论
-其他

❖ 静态成员

- 静态成员例：
for(int i=0;i<n;i++) //调用3次total函数
stud[i].total();
cout<<"the average score of"<<n<<"students is"
<<Student::average()<<endl;
//调用静态成员函数
return 0;
}

第8讲 类与对象的进一步讨论
-其他

❖ 静态成员

- 静态成员例：程序的装配
- Student类型的声明和实现可以放一个文件Student.hpp,也可以分离Student.hpp(声明)+Student.cpp(实现);
- 静态数据成员的初始化+main函数 放一个文件 test.cpp
注意：静态数据成员可以放类的任何地方，属于类不属于对象。静态成员函数一般放public，若放private，则类外程序不能使用。
程序使用的漏洞：静态成员函数不属于对象，因此是否对对象定义均可以使用。
int main(){
cout<<Student::average(); //产生除0错误
return 0; }

第8讲 类与对象的进一步讨论
-其他

❖ 静态成员

- 静态成员例：修改静态成员函数避免除0错误
float Student::average() //定义静态成员函数
{ if(count==0) return 0;
return (sum/count);
}

请思考：如果不将average函数定义为静态成员函数行不行？程序能否通过编译？需要作什么修改？为什么要用静态成员函数？请分析其理由。

2)

■ 答①：

- 类体中的数据成员的声明前加 `static` 关键字，该数据成员就成为了该类的静态数据成员。
- 静态数据成员实际上是类域中的全局变量。
- 静态数据成员被类的所有对象所共享，包括该类派生类的对象。即派生类对象与基类对象共享基类的静态数据成员。
- 静态函数不包含有编译器提供的隐藏的 `this` 指针，所以不可以调用类的非静态成员，它在类没有实例化的时候就存在

■ 答②：

- 不行，程序无法通过编译。
- 修改：将 `Student::average()` 的调用改为 `stud[0].average()`
- 静态成员函数是整个类的函数，而不是对象的函数，在类没有实例化之前，静态成员函数就已经存在。静态成员函数只能调用静态成员变量，刚好 `sum` 和 `count` 是静态成员变量，用静态成员函数计算 `average` 更加方便

2、（总分 15 分）运行文件夹“4-2 friend”中的程序，体会 `friend` 的作用。

思考几种解决 `display` 需要访问 `Date` 私有数据成员的需求：①将数据的访问控制从 `private` 改为 `public`；②将 `display` 设置为 `Date` 的友元函数；③为 `Date` 类设计读取私有数据（如在 `Date` 类的 `public` 内添加 `int getYear() const{return Year;};` 这样的成员函数）。体会不同策略的差异以及对数据和应用带来的影响。

● 实验要求：

1) 尝试三种方案。（5 分）

2) 并提交（10 分）：改写 `Date` 类，为其添加读取私有数据的公有接口。并将这些接口应用到 `display` 函数中。

■ 改写后的 `Date` 类以及改写后的 `display` 函数：

■ 方案一：

```
■ class Date { //声明 Date 类

■     public:

■         Date(int, int, int);

■         int month;

■         int day;

■         int year;

■     };

```

■ 方案二：

```

■ class Date { //声明 Date 类

■     public:

■         Date(int, int, int);

■         friend void Clock::display(const Date &); //声明 clock 中的 display 函数为友元成员函数

■     private:

■         int month;

■         int day;

■         int year;

■ };

```

■ 方案三:

```

■ class Date { //声明 Date 类

■     public:

■         Date(int, int, int);

■         int getmonth() const {

■             return month;

■         }

■         int getday() const {

■             return day;

■         }

■         int getyear() const {

■             return year;

■         }

■     private:

■         int month;

■         int day;

■         int year;

■ };

```

```

■ void Clock::display(const Date &d) { //display 的作用是输出年、月、日和时、分、秒

■     cout << d.getmonth() << "/" << d.getday() << "/" << d.getyear() << endl; //引用 Date 类对象中的私有数据

■     cout << hour << ":" << minute << ":" << second << endl; //引用本类对象中的私有数据

■ }

```

■

3、(30 分) 在 C++ 的标准模板库里定义了很多好用的扩展类型，现在我们也来试试吧。我们先来学习做 **vector 类型**。根据 4-3 myVector 文件夹中的 myVector.hpp 的类声明实现该类并通过 myVectorTest.cpp 的测试。

■ 源代码粘贴处: myVector.cpp 的源代码

```

■ #include "myVector.hpp"

■ #include <iostream>

■ using namespace std;

■

■ //用指定值 value 初始化 n 个单元 ,n<=CAPACITY

■ myVector::myVector(unsigned n, int value) {

■     size = n;

■     for (int i = 0; i < n; i++) {

■         data[i] = value;

■     }

■ }

■

■ //拷贝构造

■ myVector::myVector(const myVector &obj) {

■     size = obj.size;

■     for (int i = 0; i < obj.size; i++) {

■         data[i] = obj.data[i];

```

```

■     }

■ }

■

■ //冒泡排序

■ void myVector::sort() {

■     for (int i = 0; i < size - 1; i++) {

■         for (int j = 0; j < size - 1 - i; ++j) {

■             if (data[j] > data[j + 1]) {

■                 int temp = data[j + 1];

■                 data[j + 1] = data[j];

■                 data[j] = temp;

■             }

■         }

■     }

■ }

■ }

■

■ //赋值重载

■ myVector &myVector::operator=(const myVector &right) {

■     this->size = right.size;

■     for (int i = 0; i < right.size; i++) {

■         this->data[i] = right.data[i];

■     }

■     return *this;

■ }

■

■ //下标运算

■ int &myVector::operator[](unsigned index) {

■     if (index < 0 || index > size - 1) {

```

```

■         cout << "下标越界！" << endl;

■         exit(1);

■     }

■     return data[index];

■ }

■

■ //返回元素逆置存放的向量，即将原向量元素首尾交换的结果，注意原向量保持不变。

■ myVector myVector::operator-() {

■     myVector res;

■     for (int i = 0; i < size; i++) {

■         res.data[i] = data[size - i - 1];

■     }

■     return res;

■ }

■

■ //调整容量

■ void myVector::set_size(unsigned newsize) {

■     size = newsize;

■ }

■

■ //获取容量

■ int myVector::get_size() const {

■     return size;

■ }

■

■ //从 0 开始显示向量元素

■ void myVector::display() const {

■     for (int i = 0; i < size; i++) {

```

```

■         cout << data[i] << ' ';
■     }
■     cout << endl;
■ }
■
■
■ //表示求复杂可以求 left 和 right 的并集, 并集元素个数不超过 CAPACITY
■ myVector operator+(const myVector &left, const myVector &right) {
■     myVector res(left);
■     int newsize = left.size;
■     for (int i = 0; i < right.size; i++) {
■         bool pd = 1;
■         for (int j = 0; j < left.size; j++) {
■             if (right.data[i] == left.data[j]) {
■                 pd = 0;
■                 break;
■             }
■         }
■         if (pd == 1) {
■             res.data[newsize] = right.data[i];
■             newsize++;
■         }
■     }
■     res.set_size(newsize);
■     res.sort();
■     return res;
■ }
■
■ //表示求 left 和 right 的差集

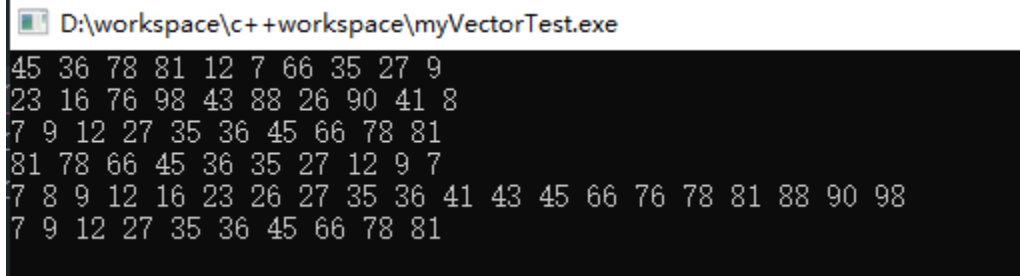
```

```

■ myVector operator-(const myVector &left, const myVector &right) {
■
■     myVector res;
■
■     int newsize = 0;
■
■     for (int i = 0; i < left.size; i++) {
■
■         bool pd = 1;
■
■         for (int j = 0; j < right.size; j++) {
■
■             if (left.data[i] == right.data[j]) {
■
■                 pd = 0;
■
■                 break;
■
■             }
■
■         }
■
■         if (pd == 1) {
■
■             res.data[newsize] = left.data[i];
■
■             newsize++;
■
■         }
■
■     }
■
■     res.set_size(newsize);
■
■     res.sort();
■
■     return res;
■
■ }

```

■ 程序测试截图：



```

D:\workspace\c++\workspace\myVectorTest.exe
45 36 78 81 12 7 66 35 27 9
23 16 76 98 43 88 26 90 41 8
7 9 12 27 35 36 45 66 78 81
81 78 66 45 36 35 27 12 9 7
7 8 9 12 16 23 26 27 35 36 41 43 45 66 76 78 81 88 90 98
7 9 12 27 35 36 45 66 78 81

```

4、（40 分+附加 5 分）在 C++ 的标准模板库里定义了很多好用的扩展类型，现在我们也来试试吧。然后我们来学习做 **string** 类型。根据 4-4 myString 文件夹中的 myStringTest.cpp 的测试需求将 myString.hpp 的类声明补充完整，并实现 myString 类，通过 myStringTest.cpp 的测试。

■ 源代码粘贴处：myString.hpp 的源代码，myString.cpp 的源代码


```
//myString.cpp

#include "myString.hpp"

#include <iostream>

#include <string.h>

using namespace std;


//构造函数

myString::myString(const char *s) {

    if (s == NULL) {

        size = 0;

        str = new char[size + 1];

        strcpy(str, "");

    } else {

        size = strlen(s);

        str = new char[size + 1];

        strcpy(str, s);

    }

}


myString::myString(const myString &a, int start, int len) {

    str = new char[len + 1];

    memcpy(str, a.str + start, len);

    str[len + 1] = '\0';

}


myString::myString(int n, char s) {

    size = n;

    str = new char[n + 1];
```

```
    for (int i = 0; i < n; i++) {  
  
        str[i] = s;  
  
    }  
  
    str[n + 1] = '\\0';  
  
}
```

//拷贝构造函数

```
myString::myString(const myString &obj) {  
  
    size = obj.size;  
  
    str = new char[size + 1];  
  
    strcpy(str, obj.str);  
  
}
```

//析构函数

```
myString::~myString() {  
  
    if (str != NULL) {  
  
        delete[] str;  
  
        str = NULL;  
  
        size = 0;  
  
    }  
  
}
```

//显示字符串

```
void myString::display() const {  
  
    cout << str << endl;  
  
}
```

//输入字符串

```
void myString::input() {

    char s[100];

    gets(s);

    size = strlen(s);

    strcpy(str, s);

}

//求字符串长

int myString::len() const {

    return size;

}

//补充下标重载运算

char &myString::operator[](int index) {

    return str[index];

}

//补充赋值重载运算

myString myString::operator=(const myString &obj) {

    if (str != NULL) {

        delete[] str;

        str = NULL;

        size = 0;

    }

    size = obj.size;

    str = new char[size + 1];

    strcpy(str, obj.str);

    return *this;

}
```

```
}
```

```
myString myString::operator=(const char *s) {
```

```
    if (str != NULL) {
```

```
        delete[] str;
```

```
        str = NULL;
```

```
        size = 0;
```

```
    }
```

```
    size = strlen(s);
```

```
    str = new char[size + 1];
```

```
    strcpy(str, s);
```

```
    return *this;
```

```
}
```

```
//相等运算符重载
```

```
int operator==(const myString &a, const myString &b) {
```

```
    if (a.size != b.size)
```

```
        return 0;
```

```
    for (int i = 0; i < a.size; i++) {
```

```
        if (a.str[i] != b.str[i])
```

```
            return 0;
```

```
    }
```

```
    return 1;
```

```
}
```

```
int operator==(const myString &a, const char *s) {
```

```
    if (a.size != strlen(s))
```

```
        return 0;
```

```
    for (int i = 0; i < a.size; i++) {
```

```
        if (a.str[i] != s[i])

            return 0;

    }

    return 1;

}
```

//>大于运算符重载

```
int operator>(const myString &a, const myString &b) {

    if (strcmp(a.str, b.str) > 0)

        return 1;

    return 0;

}
```

//+运算符重载

```
myString operator+(const myString &a, const myString &b) {

    myString c;

    c.size = a.size + b.size;

    c.str = new char[c.size + 1];

    strcpy(c.str, a.str);

    strcat(c.str, b.str);

    return c;

}
```

```
myString operator+(const myString &a, const char *b) {

    myString c;

    c.size = a.size + strlen(b);

    c.str = new char[c.size + 1];

    strcpy(c.str, a.str);
```

```

        strcat(c.str, b);

        return c;
    }

myString operator+(const char *a, const myString &b) {

    myString c;

    c.size = strlen(a) + b.size;

    c.str = new char[c.size + 1];

    strcpy(c.str, a);

    strcat(c.str, b.str);

    return c;
}

```

```

//myString.hpp

#include <iostream>

using namespace std;

class myString {

public:

    //根据测试程序写构造函数原型

    myString(const char *s = NULL);

    myString(const myString &a, int start, int len);

    myString(int n, char s);

    myString(const myString &obj);

    void display() const; //显示字符串

    void input();         //输入字符串

    int len() const;      //求字符串长

    //补充下标重载运算

    char &operator[](int index);

```

```

//相等运算符重载

friend int operator==(const myString &a, const myString &b);

friend int operator==(const myString &a, const char *s);


//>大于运算符重载

friend int operator>(const myString &a, const myString &b);


//+运算符重载

friend myString operator+(const myString &a, const myString &b);

friend myString operator+(const myString &a, const char *b);

friend myString operator+(const char *a, const myString &b);


//补充赋值重载运算

myString operator=(const myString &obj);

myString operator=(const char *s);


//补充析构函数

~myString();


private:

char *str;

int size;

};

```

■ 程序测试截图：

D:\workspace\devc++\实验四\string\项目1.exe

```
a:
b:I love ZJUT
c:hot weather
d:aaa
e:I love ZJUT
not equal
equal
hot weather
aaa
I love ZJUT
b:Steve is happy today.
Kate wasn't hungry.
Kate wasn't hungry.
I love ZJUT and hot weather
```

***拓展思考：（附加 10 分）5 分-1）**为什么 `myVector` 不需要重写类的可缺省部分，而 `myString` 需要？

5 分-2）在 `myString` 的设计中，我们将关系比较（`==`，`>`）写在类内作为类的成员，而将`+`写在类外作为普通函数，请问这样的设计合理吗？说说你的判断结论和理由？如果不合理的话，更合适的设计应该是什么模样？请描述你的设计方案。

(2) 合理，因为关系比较符一般都是两个 `myString` 比较，左值保证是 `myString` 类型例如：

`myString A, B;`

`A == B → A.operator==(B)`

而`+`运算符可能会出现`"and" + A`的情况，如果是类内重载，左值无法转换成 `myString` 类型，所以需要依靠类外重载实现，`myString operator+(const char *a, const myString &b)`。