**实验4：类的运算符重载**

**姓名：陈王子**

**班级：大数据分析2021**

**学号：202103150503**

* **请阅读此说明：实验4满分100分；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。**

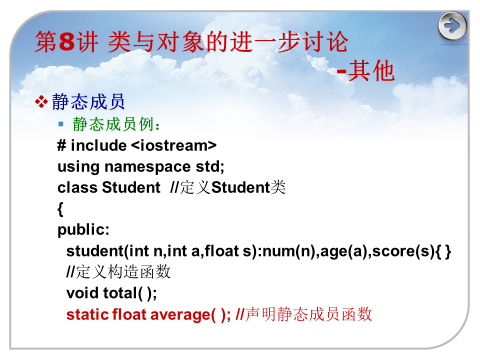
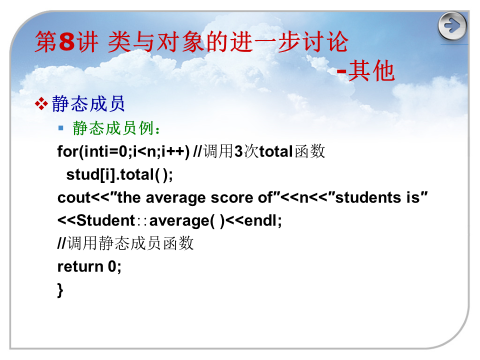
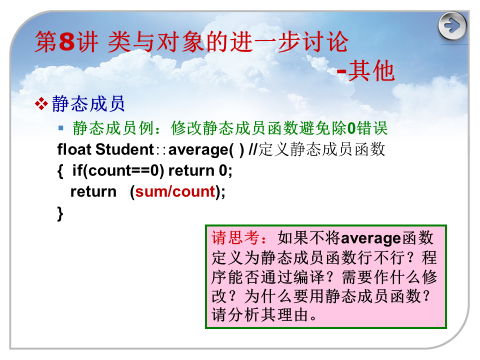
**1、（总分15分）课堂例题巩固。**

* **实验要求：**

**1）（5分）运行文件夹“4-1 static”中的两个程序，体会static的作用。**

**2）（10分）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？ ❷为什么要用静态成员函数？请分析其理由。**

* **ppt附图：**

**** ****  

**2）**

**答❶：1）（5分）运行文件夹“4-1 static”中的两个程序，体会static的作用。**

* **在函数f()的定义中加了static关键字，使得函数f()成为静态函数。静态函数只能在当前文件中被调用，不能被其他文件引用。因此，虽然在主函数之前声明了函数f()，但是由于函数f()是静态的，所以在调用f()时只会调用当前文件中的f()，而不会去寻找其他文件中可能存在的同名函数。**
* **答❷：**

#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

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() {

    sum += score;//累加总分

    count++;//累计人数

}

float Student::average() {

    return (sum / count);

}

float Student::sum = 0;

int Student::count = 0;//静态数据成员初始化

int main() {

    Student stud[3] = {Student(1001,18,70),Student(1002,19,78), Student(1005,20,98)};

    int n;

    cout << "please input the number of students:";

    cin >> n;

    for (int i = 0; i < n; i++) {

        stud[i].total();

    }

    cout << "the average score of " << n << " students is "<<Student::average()<<endl;

}

**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？**

**❷为什么要用静态成员函数？请分析其理由。**

**把 average 函数定义为常规成员函数是不行的，因为 average 函数中使用了静态数据成员 sum 和 count，而常规成员函数只能调用非静态成员。这是因为在常规成员函数中无法访问静态数据成员，因此不能初始化静态数据成员 sum 和 count。所以需要将 average 函数定义为静态成员函数，才能够正确地访问静态数据成员，并实现计算平均数功能。**

**如果不将 average 函数定义为静态成员函数，程序需要做如下修改：**

1. **sum 和 count 需从 Student 类中抽离出来，定义为全局静态变量。因为常规成员函数无法访问静态数据成员。**
2. **average 函数中需要改为调用全局静态变量 sum 和 count。**
3. **main 函数中需要修改为创建 Student 对象数组后，分别调用该数组的每个对象的 input 函数，并在输入完毕后调用全局静态变量 sum 和 count 并计算平均数。**

**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：**

**❶将数据的访问控制从private改为public；**

**❷将display设置为Date的友元函数；**

**❸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1.**

class Date

{

public:

    int month;

    int day;

    int year;

    Date(int m, int d, int y);

};

**方案2.**

class Clock

{

public:

    Clock(int h, int m, int s);

    friend void display(Date&, Clock&);

private:

    int hour;

    int minute;

    int second;

};

void display(Date& d, Clock& c)

{

    cout << d.month << "/" << d.day << "/" << d.year << endl;

    cout << c.hour << ":" << c.minute << ":" << c.second << endl;

}

**方案3.**

class Date

{

public:

    Date(int m, int d, int y);

    int getYear() const { return year; }

    int getMonth() const{ return month; }

    int getDay() const { return day; }

private:

    int month;

    int day;

    int year;

};

**提交：**

#include <iostream>

using namespace std;

class Date

{

public:

    Date(int m, int d, int y);

    int get\_month() const { return month; } //添加读取私有数据的公有接口

    int get\_day() const { return day; }

    int get\_year() const { return year; }

private:

    int month;

    int day;

    int year;

};

class Clock

{

public:

    Clock(int h, int m, int s);

    void display(const Date&);

private:

    int hour;

    int minute;

    int second;

};

Clock::Clock(int h, int m, int s)

{

    hour = h;

    minute = m;

    second = s;

}

void Clock::display(const Date& d)

{

    cout << d.get\_month() << "/" << d.get\_day() << "/" << d.get\_year() << endl;

    cout << hour << ":" << minute << ":" << second << endl;

}

Date::Date(int m, int d, int y)

{

    month = m;

    day = d;

    year = y;

}

int main()

{

    Clock t1(10, 13, 56);

    Date d1(12, 25, 2004);

    t1.display(d1);

    return 0;

}

**3、（20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**

// 构造函数

myVector::myVector(unsigned n, int value)

{

    size = n > CAPACITY ? CAPACITY : n;

    std::fill(data, data + size, value);

}

// 拷贝构造函数

myVector::myVector(const myVector& obj)

{

    std::copy(obj.data, obj.data + obj.size, data);

    size = obj.size;

}

// 赋值操作符重载

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

{

    if (this == &right)

        return \*this;

    std::copy(right.data, right.data + right.size, data);

    size = right.size;

    return \*this;

}

// 下标运算符重载

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

    if (index >= size || index < 0)

        throw "Index out of bounds";

    return data[index];

}

// 调整容量

void myVector::set\_size(unsigned newsize){

    if (newsize > CAPACITY)

        newsize = CAPACITY;

    if (newsize < size)

        size = newsize;

    else

        std::fill(data + size, data + newsize, 0);

    size = newsize;

}

// 获取容量

int myVector::get\_size() const{

    return size;

}

// 返回元素逆置存放的向量

myVector myVector::operator-(){

    myVector ret(\*this);

    for (int i = 0; i < size / 2; ++i)

    {

        swap(ret.data[i], ret.data[size - i - 1]);

    }

    return ret;

}

// 升序排序

void myVector::sort(){

    std::sort(data, data + size);

}

// 从0开始显示向量元素，以逗号分隔每个单元值

void myVector::display() const{

    if (size == 0)

    {

        cout << "[]" << endl;

        return;

    }

    cout << "[" << data[0];

    for (int i = 1; i < size; ++i)

    {

        cout << "," << data[i];

    }

    cout << "]" << endl;

}

// 前置增量重载

myVector myVector::operator++(){

    for (int i = 0; i < size; ++i)

    {

        ++data[i];

    }

    return \*this;

}

// 后置增量重载

myVector myVector::operator++(int){

    myVector ret(\*this);

    ++(\*this);

    return ret;

}

//加号运算符重载

myVector myVector::operator+(const myVector& vec) const {

    // 根据两个向量中元素数量较小的那个来创建新的向量

    int newSize = (size > vec.size) ? vec.size : size;

    myVector result(newSize);

    // 对相应位置上的元素进行相加

    for (int i = 0; i < newSize; ++i) {

        result.data[i] = data[i] + vec.data[i];

    }

    return result;

}

// 减号运算符重载

myVector myVector::operator-(const myVector& vec) const {

    // 根据两个向量中元素数量较小的那个来创建新的向量

    int newSize = (size > vec.size) ? vec.size : size;

    myVector result(newSize);

    // 对相应位置上的元素进行相减

    for (int i = 0; i < newSize; ++i) {

        result.data[i] = data[i] - vec.data[i];

    }

    return result;

}

// 输出流重载

ostream& operator<<(ostream& out, const myVector& vec){

    if (vec.size == 0)

        return out << "[]";

    out << "[" << vec.data[0];

    for (int i = 1; i < vec.size; ++i)

        out << ", " << vec.data[i];

    out << "]";

    return out;

}

// 输入流重载

istream& operator>>(istream& in, myVector& vec){

    string str;

    int val = 0; // 实际读取到的整数值

    int i = 0;   // 当前向量元素的索引

    // 从输入流中读取一行字符串

    getline(in, str);

    // 解析向量字符串

    for (auto it = str.begin(); it != str.end(); ++it)

    {

        // 如果可以读取到一个整数值

        if (isdigit(\*it))

        {

            val = val \* 10 + (\*it - '0'); // 多位数情况下，对读取到的字符进行拼凑

        }

        // 如果遇到分隔符或者字符串结束符，则将当前读取到的整数值添加到向量中

        if (!isdigit(\*it) || it == str.end() - 1)

        {

            vec.data[i++] = val;

            val = 0; // 重置 val 为 0

            if (i >= CAPACITY)

            {

                // 如果向量已经满了，设置向量大小并返回输入流

                vec.size = i;

                return in;

            }

        }

    }

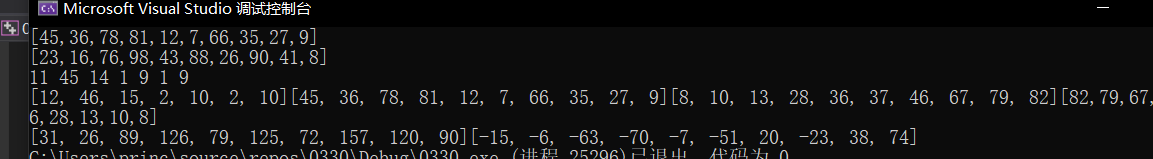
    // 设置向量大小并返回输入流

    vec.size = i;

    return in;

}

* **程序测试截图：**



**4、（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。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**

#include <iostream>

#include <cstring>

#include <limits>

using namespace std;

class myString {

public:

    // 默认构造函数

    myString();

    // 带参构造函数：使用 char\* s 初始化字符串

    myString(const char\* s);

    // 带参构造函数：使用 char\* s、开始位置 start 和字符串长度 len 初始化字符串

    myString(const char\* s, int start, int len);

    // 带参构造函数：使用字符 c 重复 count 次构造新字符串

    myString(int count, char c);

    // 拷贝构造函数

    myString(const myString& other);

    // 赋值运算符重载

    myString& operator=(const myString& other);

    // 析构函数

    ~myString();

    // 成员函数

    void display() const; // 显示字符串

    void input(); // 输入字符串

    int len() const; // 求字符串长度

    // 运算符重载

    char& operator[](int index); // 下标重载运算

    friend bool operator==(const myString& a, const myString& b); // 字符串等于比较

    friend bool operator>(const myString& a, const myString& b); // 字符串大于比较

    friend myString operator+(const myString& a, const myString& b); // 字符串拼接

    myString& operator+=(const myString& other); // 字符串拼接赋值运算

    // 输入输出流重载

    friend ostream& operator<<(ostream& out, const myString& s);

    friend istream& operator>>(istream& in, myString& s);

private:

    char\* str;

    int length;

};

// 默认构造函数实现

myString::myString() {

    str = nullptr;

    length = 0;

}

// 带参构造函数：使用 char\* s 初始化字符串

myString::myString(const char\* s) {

    length = strlen(s);

    str = new char[length + 1];

    strcpy\_s(str, length + 1, s);

}

// 带参构造函数：使用 char\* s、开始位置 start 和字符串长度 len 初始化字符串

myString::myString(const char\* s, int start, int len) {

    length = len;

    str = new char[length + 1];

    strncpy\_s(str, length + 1, s + start, length);

    str[length] = '\0';

}

// 带参构造函数：使用字符 c 重复 count 次构造新字符串

myString::myString(int count, char c) {

    length = count;

    str = new char[length + 1];

    for (int i = 0; i < length; ++i) {

        str[i] = c;

    }

    str[length] = '\0';

}

// 拷贝构造函数实现

myString::myString(const myString& other) {

    length = other.length;

    str = new char[length + 1];

    strcpy\_s(str, length + 1, other.str);

}

// 赋值运算符重载实现

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

    if (this != &other) {

        delete[] str;

        length = other.length;

        str = new char[length + 1];

        strcpy\_s(str, length + 1, other.str);

    }

    return \*this;

}

// 析构函数实现

myString::~myString() {

    delete[] str;

}

// 成员函数实现

void myString::display() const {

    if (str != nullptr && strlen(str) > 0) {

        cout << str;

    }

    else {

        cout << "empty string";

    }

}

void myString::input() {

    const int MAX\_LEN = 1000;

    char buf[MAX\_LEN];

    cin.getline(buf, MAX\_LEN);

    int new\_length = strlen(buf);

    if (str != nullptr && new\_length <= length) {

        strcpy\_s(str, length + 1, buf);

        length = new\_length;

    }

    else {

        delete[] str;

        length = new\_length;

        str = new char[length + 1];

        strcpy\_s(str, length + 1, buf);

    }

}

int myString::len() const {

    return length;

}

// 运算符重载实现

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

    if (index < 0 || index >= length) {

        cout << "Index error!" << endl;

        exit(-1);

    }

    return str[index];

}

bool operator==(const myString& a, const myString& b) {

    return strcmp(a.str, b.str) == 0;

}

bool operator>(const myString& a, const myString& b) {

    return strcmp(a.str, b.str) > 0;

}

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

    myString res;

    res.length = a.length + b.length;

    res.str = new char[res.length + 1];

    strcpy\_s(res.str, res.length + 1, a.str);

    strcat\_s(res.str, res.length + 1, b.str);

    return res;

}

myString& myString::operator+=(const myString& other) {

    char\* temp = new char[length + 1];

    strcpy\_s(temp, length + 1, str);

    delete[] str;

    length += other.length;

    str = new char[length + 1];

    strcpy\_s(str, length + 1, temp);

    strcat\_s(str, length + 1, other.str);

    delete[] temp;

    return \*this;

}

// 输入输出流重载函数实现

ostream& operator<<(ostream& out, const myString& s) {

    out << s.str;

    return out;

}

istream& operator>>(istream& in, myString& s) {

    const int MAX\_LEN = 1000;

    char buf[MAX\_LEN];

    in.getline(buf, MAX\_LEN);

    s.length = strlen(buf);

    delete[] s.str; // 防止内存泄漏

    s.str = new char[s.length + 1];

    strcpy\_s(s.str, s.length + 1, buf);

    return in;

}

int main() {

    char str[30] = "Sue likes hot weather.";

    myString a, //空串

        b("I love ZJUT"),  //I love ZJUT

        c(str, 10, 11), //hot weather

        d(3, 'a'), //aaa

        e(b);//I love ZJUT

    cout << "a:"; a.display(); cout << endl;

    cout << "b:"; b.display(); cout << endl;

    cout << "c:"; c.display(); cout << endl;

    cout << "d:"; d.display(); cout << endl;

    cout << "e:"; e.display(); cout << endl;

    if (b == str) cout << "equal";

    else cout << "not equal";

    cout << endl;

    if (b == e) cout << "equal";

    else cout << "not equal";

    cout << endl;

    a = c;

    a.display();

    a[2] = 'x';

    a.display();

    if (d > e) d.display();

    else e.display();

    //输出字符串b的内容

    for (int i = 0; i < b.len(); i++)

        cout << b[i];

    cout << endl;

    b = "Steve is happy today.";

    cout << "b:"; b.display(); cout << endl;

    a.input(); //输入：Kate wasn't hungry.

    a.display();  //输出：Kate wasn't hungry.

    a = e + " and " + c;

    a.display(); //输出： I love ZJUT and hot weather

    cout << endl;

    cin >> d;

    cout << d;

    return 0;

}

* **程序测试截图：**

****

**（20分）思考：**

**（5分）-1）为什么myVector不需要重写类的可缺省部分，而myString需要？**

**myVector和myString都是动态数组类，但是它们的数据类型不同。对于myVector，其元素可以是任意类型，因此不需要重写类的可缺省部分。而myString是一个字符串类，涉及到字符串的输入输出、拼接、字符比较等多种操作，因此需要考虑各种情况下如何处理。**

**（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。**

**将关系比较运算符重载为成员函数以便实现链式调用，而字符串拼接作为一个常规操作，为方便使用可以将其设计为类外的普通函数。这样的设计是合理的，因为关系比较运算符（==，>）被重载为成员函数，可以获得类成员的访问权限；字符串拼接函数作为类外函数，与类中的其他函数的访问权限相同，也能够方便地通过命名空间调用。如果希望更改这样的设计，可以将字符串拼接函数也重载为成员函数，实现链式调用，或者将关系比较运算符重载为友元函数，以获得两个对象的私有成员变量。**

**（10分）-3）请为myString类设计输入输出流重载，并在主函数中测试它们。**

**（已经在上文程序中实现并测试）**