**实验2：复习2&类与对象初步**

**姓名\_\_\_李飞飞\_班级\_软工2206\_学号\_202105710309\_**

* **请阅读此说明：实验2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（.cpp）打包上传到学习通。**
* **PART I 字符串复习部分**

**1、编写函数，完成和实现拷贝字符串n个字符的功能。(15分)**

**函数原型为：**

**bool myStrNCpy(char \*to ,const char \*from,unsigned startpos,unsigned len);**

**//表示需要从from串中的startpos位置开始拷贝len个字符串内容到to串中，返回拷贝//是否成功的信息(失败情况如位置无效，长度无效等)。**

* **实验要求：**

**❶ （10分）按照给定的函数原型定义并实现函数，并且设计主函数进行函数功能的测试。**

**❷** **（5分）思考：设计的函数原型中const的作用。to形参可以设计为const char\* 吗？from形参可以不设计为const char \*吗？请调整设计验证你的想法。并给出const在形参设计中的作用相关的思考结论。**

* **实验提交：**

**❶ 将完整的源代码和测试截图 粘贴在下面。**

* **测试参考：假设from串内容为：abcdefghijklmn**

**测试时使用可以参考如下可能的调用样本：**

**char to[200],from[200];**

**cin>>from;**

**myStrNCpy (to, from, 0,0);** cout<<to<<endl;

**myStrNCpy (to, from, 3,4);** cout<<to<<endl;

**myStrNCpy (to, from, 3,20);** cout<<to<<endl;

**myStrNCpy (to, "abcdefghijklmn",0,0);** cout<<to<<endl;

**myStrNCpy (to, "abcdefghijklmn", 3,4);** cout<<to<<endl;

**myStrNCpy (to, "abcdefghijklmn", 3,20);** cout<<to<<endl;

* **mystrncpy函数源代码粘贴处：**

#include<iostream>

#include<cstring>

using namespace std;

bool myStrNCpy(char\* to, const  char\* from, unsigned startpos, unsigned len) {

   if (startpos<0 || (startpos + len)>strlen(from)) {

      cout << "copy failed!\n";

      return false;

   }

   for (int i = 0; i < len; i++) \*(to + i) = \*(from + i + startpos);

   \*(to + len) = '\0';

   return true;

}

int main() {

   char to[200], from[200];

   cin >> from;

   myStrNCpy(to, from, 0, 0);  cout << to << endl;

   myStrNCpy(to, from, 3, 4);  cout << to << endl;

   myStrNCpy(to, from, 3, 20);  cout << to << endl;

   myStrNCpy(to, "abcdefghijklmn", 0, 0);    cout << to << endl;

   myStrNCpy(to, "abcdefghijklmn", 3, 4);    cout << to << endl;

   myStrNCpy(to, "abcdefghijklmn", 3, 20);   cout << to << endl;

}

* **程序测试截图：**

****

**❷思考答：**

**const表示在此作用域中不允许修改参数，增强了可读性和安全性，一来让人一眼就能看懂，二来防止在工程中不小心修改了原设计不允许改的值。当修改时编译器会直接报错，以此提示程序员**

**to不能设计为,to成了const那to就不能改了这函数不是写了个寂寞吗**

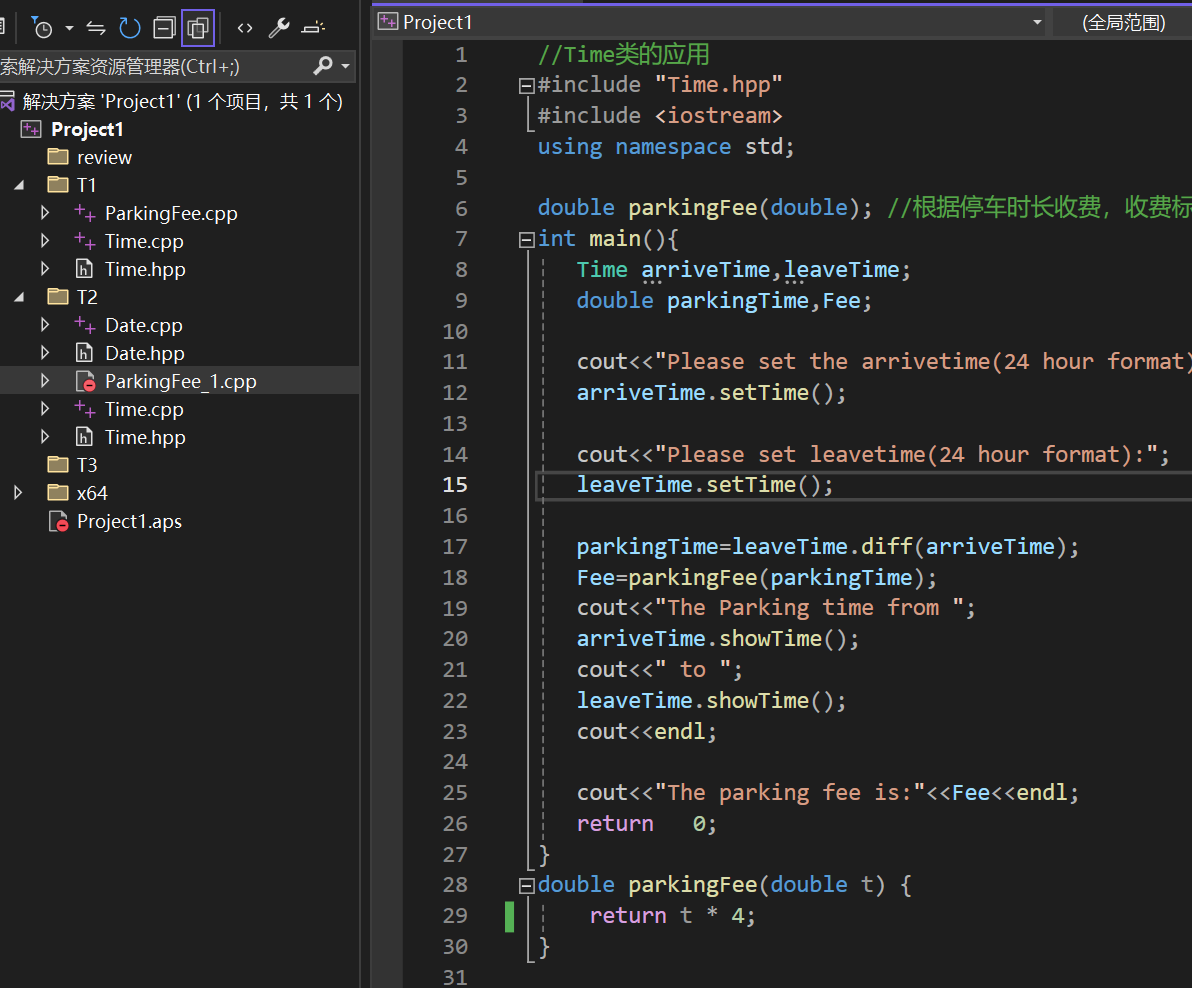
**from写成const如上所诉，就是为了防止from被改写**

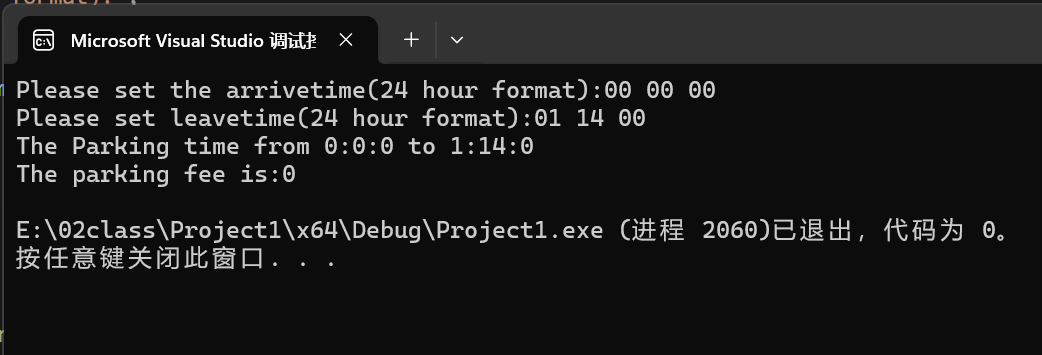
* **PART II 类与对象初步部分**

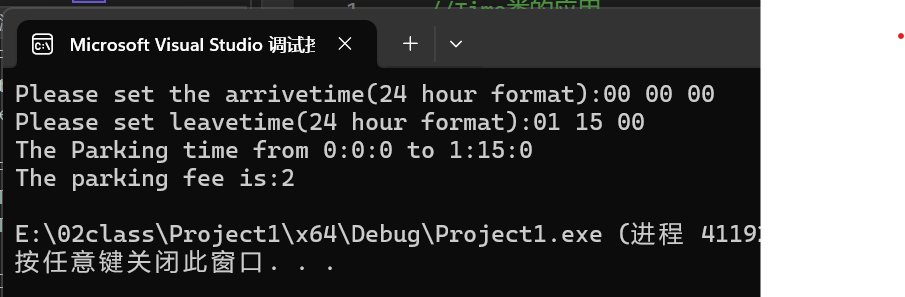
1. **停车场计费系统 (10分)**

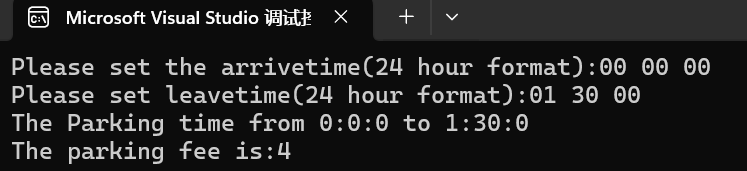
**按照附件2-1给出的文件装配程序：1) 建立工程项目命名名为ParkingFee；2) 导入附件中的三个文件，编译Time.cpp 构建Time类库，编译ParkingFee.cpp完成对应的程序测试。（提示：注意程序在编译中的问题，关注const的约束，将Time.cpp代码适当调整）**

* **程序测试截图（包含工程文件结构截图）：**

****

****

****

****

1. **停车场计费系统 再讨论（35分）**

**以课堂材料给出的停车场系统为基础考虑，假设原有的收费系统进行了补充，可以将收费时段扩展到跨天、月、年收费。**

* + **1小时内免费；**
  + **1小时以上：**
  + **超过15分钟不到30分钟按半小时收费；**
  + **超过30分钟不到1小时按1小时收费；**
  + **超过1小时按每小时4元收费；**
  + **超过8小时，每天（按24小时）按30元封顶收费；停车超过1个月（按30天）的，停车费打9折；超过1年（按365天）的停车费打8折。**
* **实验要求：**

**❶补充Date类，数据成员部分为：年year、月month、日day；成员函数部分可以仿照原来Time类的功能设计。采用补充的新类Date和原来的Time类一起工作完成停车收费程序。主程序可以参考2-2 ParkingFee\_1.cpp（15分）**

**❷考虑另一种程序调整模型：修改Time的设计，改为DateTime，补充数据成员年year、月month、日day，补充或者调整类内成员函数的设计，并实现它。采用你实现的新的DateTime进行停车收费的测试（根据需要调整原有的主函数）。并将完整的源代码和测试截图黏贴在下面。（15分）**

**\*思考：在新的收费规则下，两种程序调整的策略差异导致调整的工作量不同，你更倾向哪一种？给出你的理由（5分）**

**❶**

* **源代码粘贴处：（Date类的设计和实现，测试程序）**
  + - * **Date.hpp**

//Date类声明,即为课程材料中的clock类

class  Date{

public:

    void showDate();

    void setDate();

   double diff(const Date& T); //注意常引用对形参T的约束

private:

    long normalize() const; //注意const在此的必要性

    int year;

    int month;

    int day;

};

* + - * **Date.cpp**

//Date类实现

#include "Date.hpp"

#include <iostream>

using namespace std;

int mp[13] = { 0,31,28,31,30,31,30,31,31,30,31,30,31 };

void Date::showDate(){

    cout<<year<<":"<< month <<":"<< day;

}

void Date::setDate(){

    do{

      cin>> year >> month >> day;

    } while (year < 0 || year>2024 || month < 0 || month>12 || day<0 ||

        day>((mp[month] + (month == 2 && (year % 400 == 0 || (year % 4 == 0 && (year % 100))) ? 0 : 1))));

}

double Date::diff(const Date& T){

    long d = normalize() - T.normalize();

    if (d < 0) return -1; // 如果差值为负数，则返回-1

    return d; // 返回日期差值

}

long Date::normalize()const{

    long totalDays = day; // 将天数加到总天数中

    // 计算年份和月份对应的天数

    for (int i = 1; i < month; ++i) {

        totalDays += mp[i];

    }

    // 如果是闰年并且当前月份大于2月，则2月多加一天

    if (month > 2 && ((year % 400 == 0) || (year % 4 == 0 && year % 100 != 0))) {

        totalDays += 1;

    }

    // 计算年份对应的天数

    totalDays += (year - 1) \* 365 + (year - 1) / 4 - (year - 1) / 100 + (year - 1) / 400;

    return totalDays;

}

* + - * **ParkingFee\_1.cpp**

//Time类的应用

#include "Date.hpp" //+补充Date类的应用

#include "Time.hpp"

#include <iostream>

#include<cmath>

using namespace std;

double parkingFee(double day,double mint); //根据停车时长收费,需调整为根据真实的停车时长（可能包含日期差，时间差)收费

int main(){

   Date arriveDate,leaveDate;

   Time arriveTime, leaveTime;

   double parkingTime, Fee;

   double parkingDate;

   cout << "Please set the arriveDate:";

   arriveDate.setDate();

   cout << "Please set the arrivetime(24 hour format):";

   arriveTime.setTime();

   cout << "Please set the leaveDate:";

   leaveDate.setDate();

   cout << "Please set leavetime(24 hour format):";

   leaveTime.setTime();

   parkingTime = leaveTime.diff(arriveTime);

   //补充计算停车的日期差

   parkingDate = leaveDate.diff(arriveDate);

   Fee=parkingFee(parkingDate,parkingTime); //更新原有的收费计算

   cout << "The Parking time from ";

   arriveDate.showDate();

   cout << " ";

   arriveTime.showTime();

   cout << " to ";

   leaveDate.showDate();

   cout << " ";

   leaveTime.showTime();

   cout << endl;

   cout << "The parking fee is:" << Fee << endl;

   return   0;

}

double parkingFee(double day, double mint) {

   double ans = 0;

   //计算分钟贡献

   ans += min(30.0, 4 \* mint);

   //计算日期贡献

   ans += day \* 30;

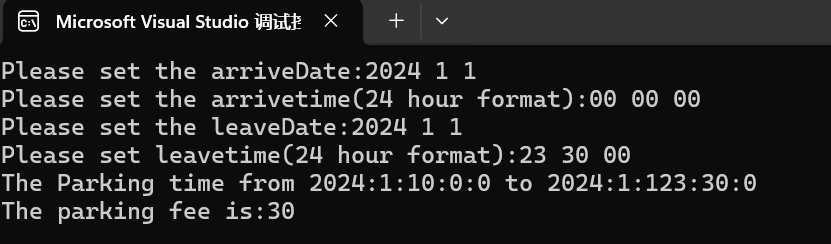
   if (day > 365)return ans \* 0.8;

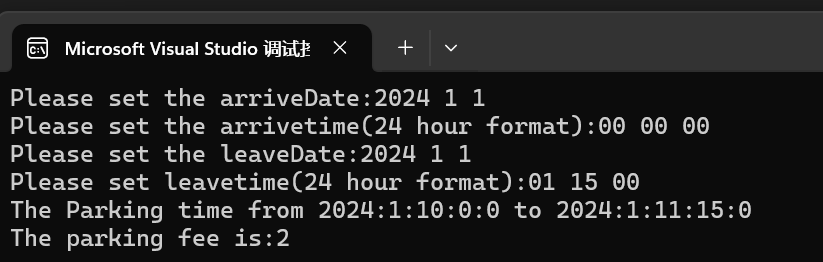
   if (day > 30)return ans \* 0.9;

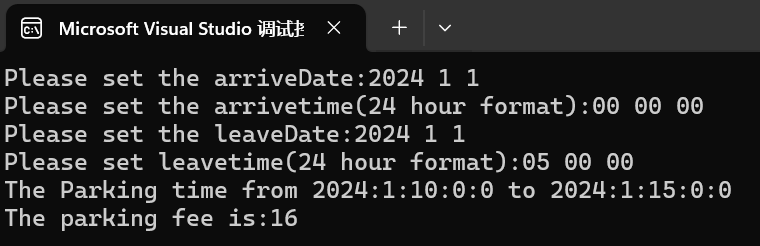
   return ans;

}

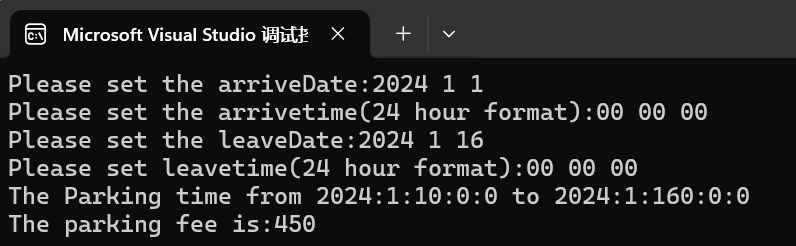
* **程序测试截图：**
  + - * **同一天：**

****

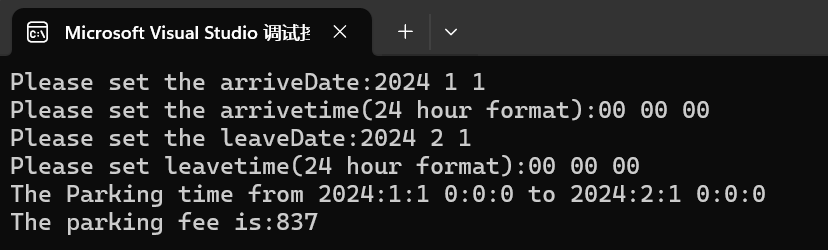
****

****

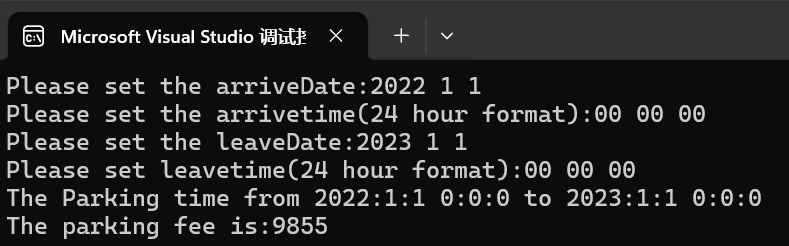
* + - * **同一月**

****

* + - * **同一年九折**

****

* + - * **不同年 八折**



**❷**

* **源代码粘贴处：（DateTime类的设计和实现，测试程序）**
  + **DataTime.h**

#pragma once

//DateTime类声明,即为课程材料中的clock类

class  DateTime {

public:

    void showDateTime();

    void setDateTime();

    double diff(const DateTime& T); //注意常引用对形参T的约束

private:

    long normalize() const; //注意const在此的必要性

    int year;

    int month;

    int day;

    int hour;

    int minute;

    int second;

};

* + **DataTime.cpp**

//DateTime类实现

#include "DateTime.h"

#include <iostream>

using namespace std;

int mp1[13] = { 0,31,28,31,30,31,30,31,31,30,31,30,31 };

void DateTime::showDateTime() {

    cout << year << ":" << month << ":" << day;

    cout << " ";

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

}

void DateTime::setDateTime() {

    do {

        cin >> year >> month >> day;

    } while (year < 0 || year>2024 || month < 0 || month>12 || day<0 ||

        day>((mp1[month] + (month == 2 && (year % 400 == 0 || (year % 4 == 0 && (year % 100))) ? 0 : 1))));

    do {

        cin >> hour >> minute >> second;

    } while (hour < 0 || hour>24 || minute < 0 || minute>59 || second < 0 || second>59);

}

double DateTime::diff(const DateTime& T) {

    //免费一小时

    long d = normalize() - T.normalize();

    if (d % (24 \* 60) > 60)d -= 60;

    if (d < 0) return 0;

    int h = d / 60, m = d % 60;

    if (m < 15) return h;

    if (m >= 15 && m < 30) return h + 0.5;

    if (m >= 30 && m < 60) return h + 1;

}

long DateTime::normalize()const {

    long totalDays = day; // 将天数加到总天数中

    // 计算年份和月份对应的天数

    for (int i = 1; i < month; ++i) {

        totalDays += mp1[i];

    }

    // 如果是闰年并且当前月份大于2月，则2月多加一天

    if (month > 2 && ((year % 400 == 0) || (year % 4 == 0 && year % 100 != 0))) {

        totalDays += 1;

    }

    // 计算年份对应的天数

    totalDays += (year - 1) \* 365 + (year - 1) / 4 - (year - 1) / 100 + (year - 1) / 400;

    return totalDays\*24\*60+hour\*60+minute;

}

* + **ParkingFee\_2.cpp**

#include "DateTime.h"

#include<cmath>

#include<iostream>

using namespace std;

double parkingFee(double hour) {

   //获取天数

   int day = ((int)(floor(hour))) / (24);

   hour -= day \* 24;

   double ans = 0;

   //计算分钟贡献

   ans += min(30.0, 4 \* hour);

   //计算日期贡献

   ans += day \* 30;

   if (day > 365)return ans \* 0.8;

   if (day > 30)return ans \* 0.9;

   return ans;

}

int main() {

   DateTime a, b;

   cout << "请输入开始时间：";

   a.setDateTime();

   cout << "请输入离开时间：";

   b.setDateTime();

   double mints = b.diff(a);

   cout << "起止时间：";

   a.showDateTime();

   cout<< "到";

   b.showDateTime();

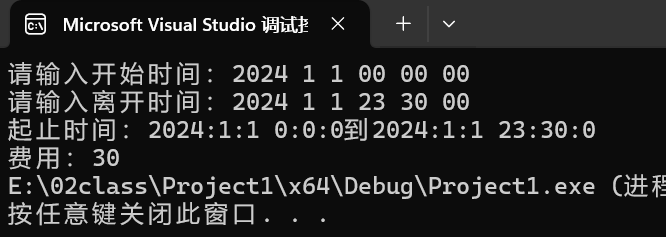
   cout<< endl;

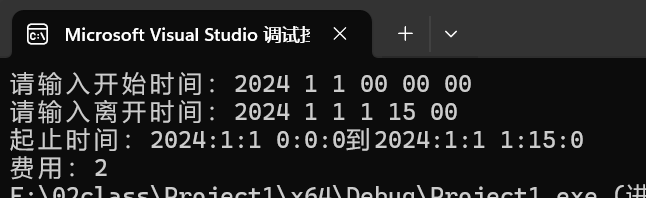
   cout << "费用：" << parkingFee(mints);

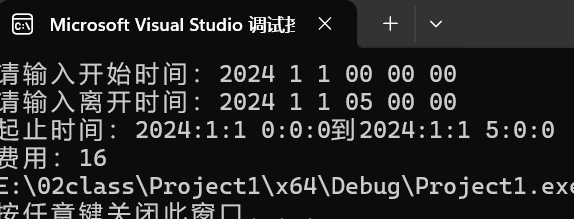
   return 0;

}

* **程序测试截图：**
  + - * **同一天：**

****

****

****

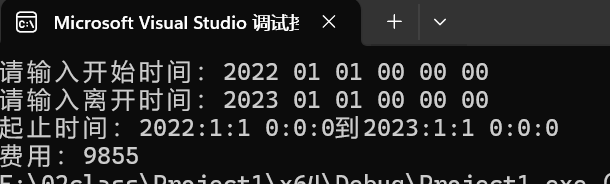
* + - * **同一月**

****

* + - * **同一年九折**

****

* + - * **不同年 八折**

****

**\*思考答：**

我更喜欢写在一起的DateTime类，将同类信息封装后更便于扩展和使用，可以提供更简介的api供使用者调用，比如要新增功能：获取时间戳。拆分开就完全无法提供此api,只能由使用者手动操作，非常不方便~ 而且分开写有大量逻辑重复的代码，白白增加了很多工作量，程序员都懒得很，要不是为了作业我才不愿意多写这么多废代码

**3、位置类position：（40分）**

**设计并实现一个平面坐标系内的位置类position。包含的基本数据成员有：横坐标，纵坐标；包含的基本成员函数有：设置位置set；读取位置show；判断第几象限quadrant；计算到原点的距离distance；计算到其他点的距离distance；计算经过原点到这个位置的直线的斜率slope；计算经过这个位置到其他点的直线的斜率slope；按坐标轴平移位置move。其他成员函数功能可以自行补充。**

* **实验要求：**

**❶ 按照描述完成position类的基本的设计和实现。将数据成员设计为私有(private)成员；将成员函数设计为公有(public)成员。并通过以下测试程序。（20分）**

**#include “position.hpp”**

**#include <iostream>**

**using namespace std;**

**int main() {**

**position a, b, c,d,e;**

**a.set(5, 15);**

**a.show();//输出(5,15)**

**cout<<a. quadrant()<<endl; //输出a的象限**

**b.set(-4.5, 6.7);**

**b.show();**

**cout<<b. quadrant()<<endl; //输出b的象限**

**c.set(-10, -100);**

**c.show();**

**cout<<c. quadrant()<<endl; //输出c的象限**

**d.set(20.5, 5.5);**

**d.show();**

**cout<<d. quadrant()<<endl; //输出b的象限**

**e.set();//默认为原点**

**e.show();**

**cout<<a.distance(b)<<endl;**

**cout<<c.distance()<<endl;//默认求与原点的距离**

**cout<<a.slope()<<endl;//与原点构成直线的斜率**

**cout<<a.slope(d)<<endl;//与d构成直线的斜率**

**a.move(3);//沿x轴平移**

**a.show();**

**b.move(-4, 5);**

**b.show();**

**c.move(0, 6);//沿y轴平移**

**c.show();**

**return 0;**

**}**

* **源代码粘贴处：（position类的设计和实现）**
  + **Position.hpp**

#pragma once

#include<cstring>

#include<iostream>

#include<cmath>

using namespace std;

class position{

public:

    position();

    ~position();

    void set(double x=0,double y=0);

    void show()const;

    //判断第几象限

    string quadrant()const;

    double distance()const;

    double distance(const position &other)const;

    double slope()const;

    double slope(const position& other)const;

    void move(double distX,double distY=0);

private:

    double x;

    double y;

};

position::position(){

    x = 0, y = 0;

}

position::~position(){}

inline void position::set(double x, double y){

    this->x = x; this->y = y;

}

inline void position::show()const {

    cout << "横坐标：" << x << " 纵坐标：" << y<<endl;

}

string position::quadrant()const {

    if (x == 0 && y == 0)return "原点";

    if (x == 0)return "y轴上";

    if (y == 0)return "x轴上";

    if (x > 0 && y > 0) return "一象限";

    if (x < 0 && y < 0) return "三象限";

    if (x > 0 && y < 0) return "四象限";

    if (x < 0 && y > 0) return "二象限";

}

inline double position::distance()const {

    return sqrt(x \* x + y \* y);

}

inline double position::distance(const position& other)const {

    return sqrt((x-other.x)\* (x - other.x) + (y - other.y) \* (y - other.y));

}

inline double position::slope()const {

    return y/x;

}

inline double position::slope(const position& other)const {

    return (other.y-y)/(other.x-x);

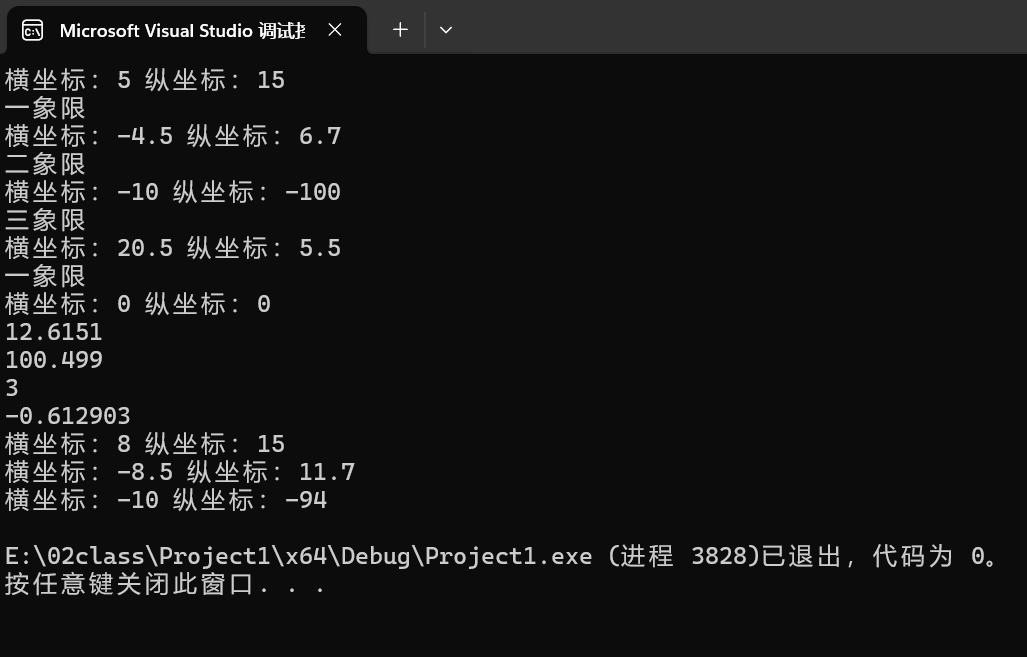
}

inline void position::move(double distX, double distY){

    x += distX; y += distY;

}

* **程序测试截图：**

****

**❷思考：3组同色 调用接口的设计你选择使用默认参数还是重载？你认为哪种是在当前应用情形下更好的设计？请说明设计的理由。（10分）**

**答：**

**默认参数更好，使用函数重载让开发人员能很明确用的是哪个函数，不会有歧义，而且很容易扩展，可以在不修改原有代码的情况下增加更多功能。但是这个作业的需求太简单了，而且这代码交了作业就不用再扩展什么东西了，用默认参数实现更方便，所以在这种情况下默认参数更好。**

**若测试代码中：**

**cout<<a.distance(b)<<endl;**

**cout<<c.distance()<<endl;//默认求与原点的距离**

**使用需求增加为：**

**cout<<distance(a,b)<<endl;**

**cout<<distance(c)<<endl;//默认求与原点的距离**

**那么程序需要做什么样的调整来满足这种需求的变更？**

**请给出你设计实现的方案。（10分）**

**答：描述设计方案并给出相关的实现代码：**

* **源代码粘贴处：** 
  + **Position.hpp（只贴了新增的两个类外定义的函数，其它都一样就不放了）**

double distance(const position& other) {

    return other.distance();

}

double distance(const position& a, const position& b) {

    return a.distance(b);

}

* + **测试函数**

#include "position.hpp"

#include <iostream>

using namespace std;

int main() {

    position a, b, c, d, e;

    a.set(5, 15);

    a.show();//输出(5,15)

    cout << a.quadrant() << endl; //输出a的象限

    b.set(-4.5, 6.7);

    b.show();

    cout << b.quadrant() << endl; //输出b的象限

    c.set(-10, -100);

    c.show();

    cout << c.quadrant() << endl; //输出c的象限

    d.set(20.5, 5.5);

    d.show();

    cout << d.quadrant() << endl; //输出b的象限

    e.set();//默认为原点

    e.show();

    cout << distance(a,b) << endl;

    cout << distance(c) << endl;//默认求与原点的距离

    cout << a.slope() << endl;//与原点构成直线的斜率

    cout << a.slope(d) << endl;//与d构成直线的斜率

    a.move(3);//沿x轴平移

    a.show();

    b.move(-4, 5);

    b.show();

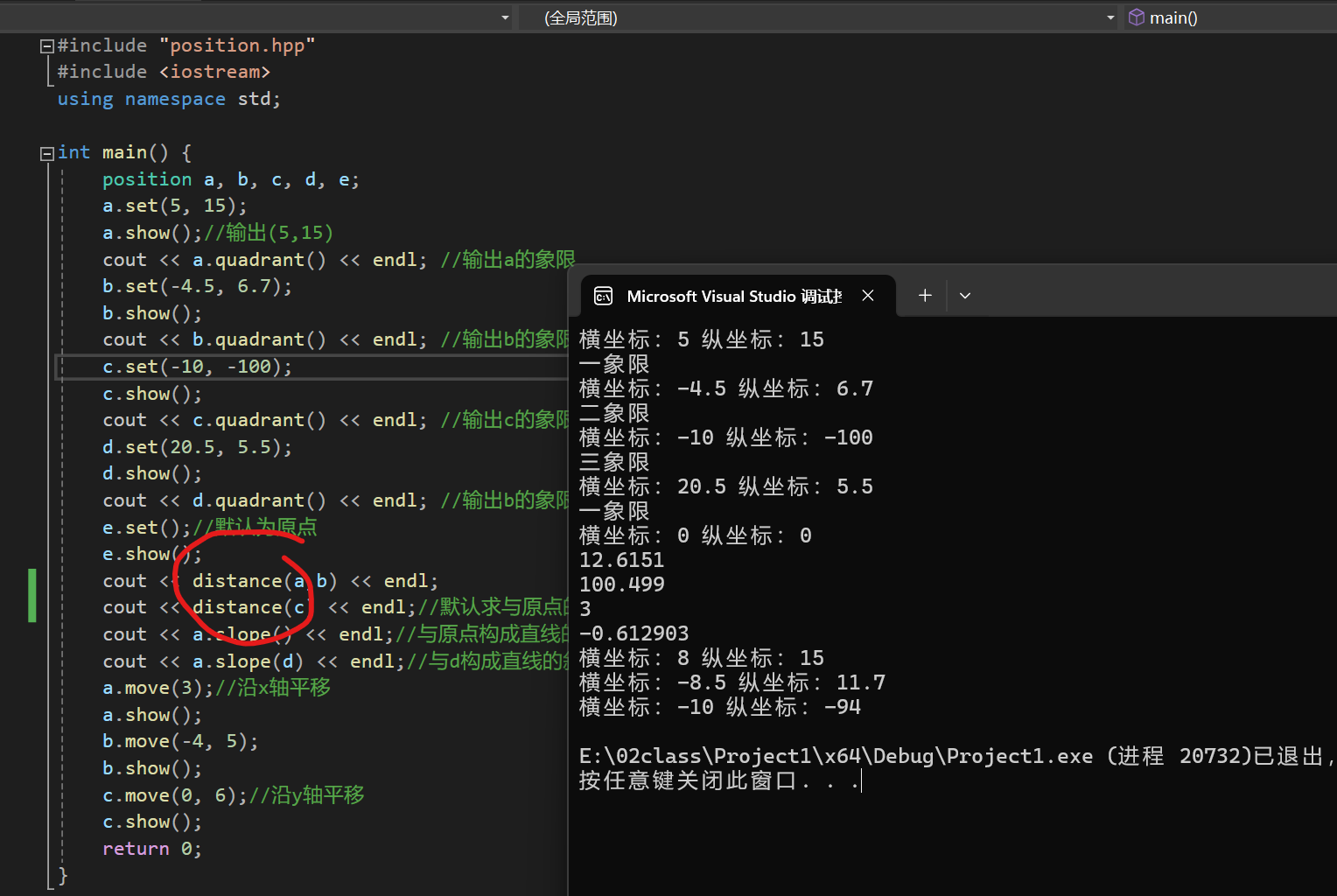
    c.move(0, 6);//沿y轴平移

    c.show();

    return 0;

}

* **程序测试截图：（增加新使用形式后的测试结果）**

****