Solution4

191220008 陈南疃

概念题

1、简述Demeter法则的基本思想；过度使用Demeter法则会带来什么问题。

基本思想：只与直接的朋友通信。模块间的耦合度反映在对象类之间的关联性上。要降低模块间的耦合度，可以对类中成员函数能访问的对象的集合作一定的限制，并尽量使该集合为最小。

过度使用的问题：过度使用迪米特法则会使系统产生大量的中介类，从而增加系统的复杂性，使模块之间的通信效率降低。所以，在采用迪米特法则时需要反复权衡，确保高内聚和低耦合的同时，保证系统的结构清晰。

2、简述为什么要对操作符重载进行重载；操作符重载会带来什么问题。

操作符重载原因：把已经定义的、有一定功能的操作符进行重新定义，来完成更为细致具体的运算等功能。可以将概括性的抽象操作符具体化，便于外部调用而无需知晓内部具体运算过程。C++允许对已有的操作符进行重载，使得它们能对自定义类型（类）的对象进行操作。

带来的问题：使用重载操作符，可以令程序更自然、更直观，而滥用操作符重载会使得类难以理解，也有可能降低程序效率。

3、简述操作符重载的两种形式；这两种形式有什么区别。

两种形式：

①通过成员函数的形式来实现运算符重载：使重载运算符成为该类的成员函数。这允许运算符函数访问类的私有成员。它也允许函数使用隐式的this指针形参来访问调用对象。

②利用友元函数进行重载的方式：使重载的成员函数成为独立分开的函数。当以这种方式重载时，运算符函数必须声明为类的友元才能访问类的私有成员。它没有隐式的this指针。

区别：

①格式：

类成员函数实现操作符重载的格式

```
class 类名
{
 public:
     返回类型 operator 操作符(形参表);
 };
 //类外定义格式
 返回类型 类名::operator 操作符(形参表)
 {
     //函数体
 }
```

友元函数重载操作符的格式

```
class 类名
{
    friend 返回类型 operator 操作符(形参表);
};
//类外定义格式:
返回类型 operator操作符(参数表)
{
    //函数体
}
```

②参数:

当重载成为成员函数时，双目运算符仅有一个参数。对单目运算符，重载为成员函数时，总是隐含了一个参数，该参数是 this 指针。 This 指针指向调用该成员函数对象的指针。

当重载友元函数时，将没有隐含的参数 this 指针。这样，对双目运算符，友元函数有 2 个参数，对单目运算符，友元函数有一个参数。

③针对不同运算符:

1)对于单目运算符:

为了尽可能到保证类封装,我们尽可能的保证不去添加新的友元函数(毕竟这些函数不是自己人但可以访问私有成员)。所以对于单目运算符我们都是重载为成员函数的。

2)对于双目运算符:

a.当重载为成员函数时，会隐含一个this指针；当重载为友元函数时，将不存在隐含的this指针，需要在参数列表中显示地添加操作数。
b.当重载为成员函数时，只允许右参数的隐式转换；当重载为友元函数时，能够接受左参数和右参数的隐式转换。

3)C++规定，赋值运算符“=”、下标运算符“[ ]”、函数调用运算符"( )"、成员运算符“->”必须作为成员函数重载。
4)流插入“<<”和流提取运算符“>>”、类型转换运算符不能定义为类的成员函数，只能作为友元函数。
5)一般将单目运算符和复合运算符（+=、-=、/=、*=、&=、!=、^=、%=、>>=、<<=）重载为成员函数。

6)一般将双目运算符重载为友元函数。

编程题

1、定义一个日期类Date，其实例为 year 年 month 月 day 日；定义一个时间类Time，其实例为 hour 时 minute 分 second 秒；定义一个日期时间类Datetime， 其实例为某日/某时。

```
#include <iostream>
#include <cmath>

using namespace std;

class Date
{
    friend class Datetime;
private:
    int _year, _month, _day;
public:
    Date() { _year = _month = _day = 0; }
    Date(int year, int month, int day);
};

class Time
{
    friend class Datetime;
private:
    int _hour, _minute, _second;
public:
    Time() { _hour = _minute = _second = 0; }
    Time(int hour, int minute, int second);
};

class Datetime
{
private:
    Date _date;
    Time _time;
public:
    Datetime(const Date& date, const Time& time);
    Datetime operator + (long) const; // 日期时间加秒数
    Datetime operator - (long) const; // 日期时间减秒数
    long operator - (const Datetime&) const; // 日期时间减日期时间
    Datetime& operator ++ (); // 日期时间加一秒
    Datetime& operator -- (); // 日期时间减一秒
    bool operator < (const Datetime&) const; // 比较日期时间大小
    bool operator == (const Datetime&) const; // 比较日期时间是否相等
    bool is_leapyear(int) const; // 判断是否为闰年
    int days_of_month(int, int) const; // 得到该月的天数
};

Date::Date(int year, int month, int day)
{
    _year = year;
    _month = month;
    _day = day;
}
```

```cpp
Time::Time(int hour, int minute, int second)
{
    _hour = hour;
    _minute = minute;
    _second = second;
}

Datetime::Datetime(const Date& date, const Time& time)
{
    _date = date;
    _time = time;
}

bool Datetime::is_leapyear(int year) const // 判断是否为闰年
{
    return (year % 4 == 0 && (year % 100 != 0)) || (year % 400 == 0);
}

int Datetime::days_of_month(int month, int year) const // 得到该月的天数
{
    switch(month)
    {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        return 31; break;
    case 4:
    case 6:
    case 9:
    case 11:
        return 30; break;
    case 2:
        return 28 + is_leapyear(year); break;
    }
}

Datetime Datetime:: operator + (long duration) const // 日期时间加秒数
{
    Datetime datetime(_date, _time);
    int day = duration / 86400;
    int hour = (duration % 86400) / 3600;
    int minute = (duration % 3600) / 60;
    int second = duration % 60;
    datetime._time._second += second;
    if (datetime._time._second > 60)
    {
        int add_minute = datetime._time._second / 60;
        datetime._time._second -= add_minute * 60;
        datetime._time._minute += add_minute;
    }
    datetime._time._minute += minute;
    if (datetime._time._minute > 60)
    {
```

```
        int add_hour = datetime._time._minute / 60;
        datetime._time._minute -= add_hour * 60;
        datetime._time._hour += add_hour;
    }
    datetime._time._hour += hour;
    if (datetime._time._hour > 24)
    {
        int add_day = datetime._time._hour / 24;
        datetime._time._hour -= add_day * 24;
        datetime._date._day += add_day;
    }
    datetime._date._day += day;
    while (datetime._date._day >
days_of_month(datetime._date._month,datetime._date._year))
    {
        datetime._date._day -= days_of_month(datetime._date._month,
datetime._date._year);
        datetime._date._month++;
        if (datetime._date._month > 12)
        {
            datetime._date._month -= 12;
            datetime._date._year++;
        }
    }
    return datetime;
}

Datetime Datetime:: operator - (long duration) const // 日期时间减秒数
{
    Datetime datetime(_date, _time);
    int day = duration / 86400;
    int hour = (duration % 86400) / 3600;
    int minute = (duration % 3600) / 60;
    int second = duration % 60;
    datetime._time._second -= second;

    if (datetime._time._second < 0)
    {
        int minus_minute = abs(datetime._time._second / 60 - 1);
        datetime._time._second += minus_minute * 60;
        datetime._time._minute -= minus_minute;
    }
    datetime._time._minute -= minute;
    if (datetime._time._minute < 0)
    {
        int minus_hour = abs(datetime._time._minute / 60 - 1);
        datetime._time._minute += minus_hour * 60;
        datetime._time._hour -= minus_hour;
    }
    datetime._time._hour -= hour;
    if (datetime._time._hour < 0)
    {
        int minus_day = abs(datetime._time._hour / 24 - 1);
        datetime._time._hour += minus_day * 24;
        datetime._date._day -= minus_day;
    }
    datetime._date._day -= day;
    while (datetime._date._day < 0)
```

```
    {
        datetime._date._month--;
        if (datetime._date._month <= 0)
        {
            datetime._date._month += 12;
            datetime._date._year--;
        }
        datetime._date._day += days_of_month(datetime._date._month,
datetime._date._year);
    }
    return datetime;
}

long Datetime:: operator - (const Datetime& datetime) const // 日期时间减日期时间
{
    long duration = 0, duration1 = 0, duration2 = 0;
    for (int year1 = 0; year1 < _date._year; year1++)
    {
        if (is_leapyear(year1))
            duration1 += 366 * 86400;
        else
            duration1 += 365 * 86400;
    }
    for (int month1 = 1; month1 < _date._month; month1++)
    {
        duration1 += days_of_month(month1, _date._year) * 86400;
    }
    duration1 += _date._day * 86400 + _time._hour * 3600 + _time._minute * 60 +
_time._second;
    for (int year2 = 0; year2 < datetime._date._year; year2++)
    {
        if (is_leapyear(year2))
            duration2 += 366 * 86400;
        else
            duration2 += 365 * 86400;
    }
    for (int month2 = 1; month2 < datetime._date._month; month2++)
    {
        duration2 += days_of_month(month2, datetime._date._year) * 86400;
    }
    duration2 += datetime._date._day * 86400 + datetime._time._hour * 3600 +
datetime._time._minute * 60 + datetime._time._second;
    duration = duration1 - duration2;
    return duration;
}

Datetime& Datetime:: operator ++ () // 日期时间加一秒
{
    *this = *this + 1;
    return *this;
}

Datetime& Datetime:: operator -- () // 日期时间减一秒
{
    *this = *this - 1;
    return *this;
}
```

```cpp
bool Datetime:: operator < (const Datetime& datetime) const // 比较日期时间大小
{
    if (_date._year < datetime._date._year)
        return true;
    else if (_date._year > datetime._date._year)
        return false;
    if (_date._month < datetime._date._month)
        return true;
    else if (_date._month > datetime._date._month)
        return false;
    if (_date._day < datetime._date._day)
        return true;
    else if (_date._day > datetime._date._day)
        return false;
    if (_time._hour < datetime._time._hour)
        return true;
    else if (_time._hour > datetime._time._hour)
        return false;
    if (_time._minute < datetime._time._minute)
        return true;
    else if (_time._minute > datetime._time._minute)
        return false;
    if (_time._second < datetime._time._second)
        return true;
    else if (_time._second > datetime._time._second)
        return false;
    return false;
}

bool Datetime:: operator == (const Datetime& datetime) const // 比较日期时间是否相等
{
    return _date._year == datetime._date._year && _date._month ==
datetime._date._month && _date._day == datetime._date._day && _time._hour ==
datetime._time._hour && _time._minute == datetime._time._minute && _time._second
== datetime._time._second;
}
```