# Solution6

**191220008 陈南疃**

## 概念题

### 1、在C++中，protected类成员访问控制的作用是什么?

在C++中，除了public和private，还提供了另外一种类成员访问控制：protected，

- 用protected说明的成员不能通过对象使用，但可以在派生类中使用。
- protected访问控制缓解了封装与继承的矛盾

### 2、请简述派生类对象的初始化和析构顺序，并简述理由，为什么需要按照这个顺序?

派生类对象的初始化由基类和派生类共同完成：

- 从基类继承的数据成员由基类的构造函数初始化；
- 派生类的数据成员由派生类的构造函数初始化。

当创建派生类的对象时：

- 先执行基类的构造函数，再执行派生类构造函数（不然可能会使用到基类中未定义的成员）。
- 默认情况下，调用基类的默认构造函数，如果要调用基类的非默认构造函数，则必须在派生类构造函数的成员初始化表中指出。

包含成员对象的对象消亡时：

- 先调用本身类的析构函数，执行完后会自动去调用基类的析构函数（不然可能使用到已经基类中已经析构的成员）。

## 编程题

### 1、下面的设计有什么问题？如何解决?

```
class Rectangle { //矩形类
public:
    Rectangle(double w, double h): width(w), height(h) {}
    void set_width(double w) { width = w; }
    void set_height(double h) { height = h; }
    double get_width() const { return width; }
    double get_height() const { return height; }
    double area() const { return width*height; }
    void print() const { cout << width << " " << height << endl; }
private:
    double width; //宽
```

```
    double height; //高
};
class Square: public Rectangle {//正方形类
public:
    Square(double s): Rectangle(s,s) {}
    void set_side(double s) { //设置边长。
        set_width(s);
        set_height(s);
    }
    double get_side() const { //获取边长。
        return get_width();
    }
}
```

**提示：**从用户安全的角度考虑设计。该题是否适用public继承？ 用public继承会带来什么样的问题？

不适用public继承。因为在矩形基类中的public成员函数，会针对单个边进行操作，适合一般的矩形操作，而在正方形派生类中，默认长和宽是相等的，若用public继承，则可能在调用基类成员函数的时候会破坏正方形的性质，不安全。

因此，可以采用private继承，对基类中的成员函数进行私有化，在派生类中利用这些函数重新定义相关操作的函数。

## 2、在作业二中，我们定义了时间类Time，现在我们利用时间类 Time，定义一个带时区的时间类 ExtTime。除了构造函数和时间调整函数外， ExtTime 的其它功能与 Time 类似。

```
enum TimeZone { W12 = -12, W11, W10, W9, W8, W7, W6, W5, W4, W3, W2, W1, GMT,
E1, E2, E3, E4, E5, E6, E7, E8, E9, E10, E11, E12 };

class Time
{
public:
    Time();
    Time(int h, int m, int s);
    void set(int h, int m, int s);
    void increment();
    void display();
    bool equal(Time& other_time);
    bool less_than(Time& other_time);
protected:
    int hour, min, sec;
};

Time::Time()
{
    hour = 0;
    min = 0;
    sec = 0;
}

Time::Time(int h, int m, int s)
{
```

```
    hour = h;
    min = m;
    sec = s;
}

void Time::set(int h, int m, int s)
{
    hour = h;
    min = m;
    sec = s;
}

void Time::increment()
{
    if (sec == 59)
    {
        if (min == 59)
        {
            if (hour == 23)
                hour = min = sec = 0;
            else
            {
                hour++;
                min = sec = 0;
            }
        }
        else
        {
            min++;
            sec = 0;
        }
    }
    else
        sec++;
}

void Time::display()
{
    cout << hour << ":" << min << ":" << sec << endl;
}

bool Time::equal(Time& other_time)
{
    return other_time.hour == hour && other_time.min == min && other_time.sec ==
sec;
}

bool Time::less_than(Time& other_time)
{
    int sub_hour, sub_min, sub_sec;
    sub_hour = other_time.hour - hour;
    sub_min = other_time.min - min;
    sub_sec = other_time.sec - sec;
    if (sub_hour > 0 || sub_hour == 0 && sub_min > 0 || sub_hour == 0 && sub_min
== 0 && sub_sec > 0)
        return true;
    else
        return false;
```

```
}

class ExtTime :public Time
{
private:
    TimeZone timezone;
public:
    ExtTime(); //设置为GMT时间
    ExtTime(int h, int m, int s, TimeZone t); //构造函数
    void set(int h, int m, int s, TimeZone t); //调整时间
    void display();//展示时间，首先打印时区，再打印时间
    bool equal(const ExtTime& other_time); //判断是否相等
    bool less_than(const ExtTime& other_time);//判断是否小于
};

ExtTime::ExtTime() //设置为GMT时间
{
    hour = 0;
    min = 0;
    sec = 0;
    timezone = GMT;
}

ExtTime::ExtTime(int h, int m, int s, TimeZone t) //构造函数
{
    hour = h;
    min = m;
    sec = s;
    timezone = t;
}

void ExtTime::set(int h, int m, int s, TimeZone t) //调整时间
{
    Time::set(h, m, s);
    timezone = t;
}

void ExtTime::display() //展示时间，首先打印时区，再打印时间
{
    if (timezone == 0)
        cout << "GMT ";
    else
    {
        if (timezone < 0)
            cout << "W";
        else
            cout << "E";
        cout << abs(timezone) << " ";
    }
    Time::display();
}

bool ExtTime::equal(const ExtTime& other_time) //判断是否相等
{
    Time time1((hour + GMT - timezone) % 24, min, sec);
    Time time2((other_time.hour + GMT - other_time.timezone) % 24,
other_time.min, other_time.sec);
    return time1.equal(time2);
```

```
}

bool ExtTime::less_than(const ExtTime& other_time) //判断是否小于
{
    Time time1((hour + GMT - timezone) % 24, min, sec);
    Time time2((other_time.hour + GMT - other_time.timezone) % 24,
other_time.min, other_time.sec);
    return time1.less_than(time2);
}
```

我认为该题适合用public继承。

按常理来说，对于ExtTime的对象来说，并不适合调用基类Time中的set函数，equal函数和less_than函数等，因为还需要根据时区进行判断才能得到正确结果，从这个角度来说，是不太适合public继承的。但若仅仅想进行数值上的比较时，则可以调用基类Time中的这些函数，以实现新的功能。此外，基类Time中的increment函数仍可以复用。

因此，当基类中的成员函数不会对派生类的性质产生影响或进行错误的操作时，则可以使用public继承。所以protected继承和private继承则是可以防止这种情况的发生。同时，使派生类中的成员对外界不可访问。