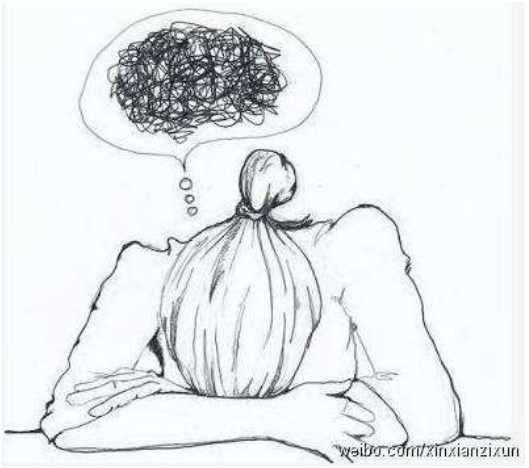
实践1小结

- 规范化问题（无意义标识符、无序缩进...）。
  - 如何写出让同事无法维护的代码
  - 写出一手烂代码的19条准则
- 测试（范例是否覆盖尽可能多的可能性？）
  - **Rectangle类，完备性，例：若初始化为(2,0,0,2)，你的计算结果，他们还好吗？**
  - **Time类，枚举的应用。**
  - **Fraction类，面向对象的思想。**

实践1小结

- 建立“函数也可以是数据类型的成员”的思想，逻辑上完成了类的封装。
- 逐步熟悉类的声明，定义，使用。
  - 数据(属性)，变量成员的访问和设置。
  - 操作(行为)，函数成员的调用。
- 类声明时成员变量不可以赋值，那为什么成员函数中能对变量赋值？
- 旧语法+新语法，乱如麻！



实践1小结

- 语法问题：
  - 关于函数调用，函数的传值调用无法改变实参，向函数传递引用和指针可以改变实参，所以需要谨慎使用，必要时需要const保护。
  - 指针是C的灵魂，学语法的阶段就是盘它的阶段，迟早会搞定这个磨人的小妖精。
  - 具体的：
    - 1.野指针，本次实验主要是使用了未初始化的指针变量！
    - 2.Link错误，本次实验主要是函数的声明和定义的概念！
    - 3.局部变量屏蔽外层同名变量！
    - 4.const成员函数不可（对本对象）调用非const成员函数！

More About Defining Classes for Objects

◎ 如果没有显式地定义，C++为每个类隐式地提供缺省的(无参的)构造函数，拷贝构造函数，析构函数各一个。

◎ 如果显式定义了构造函数，那么建议定义一个无参(或有参带默认值的)构造函数，方便无参地初始化对象，尤其是定义对象数组。

◎ 使用复制构造函数时，将创建一个与调用对象有相同属性的新对象，可以各自独立操作。

◎ 建议所有仅访问(不设置)成员变量的成员函数，声明为只读成员函数，方便const引用对象时使用，即应一致性地使用const限定符。

实践1 小结

- 可维护性问题：
  - 如果你定义的枚举类型初始值从1改为0，程序就需要作大量改动，那么你对枚举类型的力量还一无所知。
  - 所谓类的封装性（information hiding)：类的使用者只能执行类允许公开的数据或操作，避免了外部操作对内部数据的影响，提高程序的可维护性，不能让外面的程序随意修改一个类的成员变量。
  - 一个完备的类的定义是一个随需求逐步完善与扩展的过程，在扩展过程中，原有的成员(变量成员，函数成员)应该尽可能保持可用性(这就对你定义的成员函数的功能有约束要求)。

为什么add函数只传一个参数？

◎在调用某个对象的成员函数时，系统自动产生一个隐藏的指针——this指针。
◎ this指针指向调用其成员函数的对象（即本对象）。
◎类内访问成员时不使用成员访问运算符，也是因为隐式调用了this指针。

```
int Location::GetX() { return X; }
```
若将this指针现形，则等效为：
```
int Location::GetX(Location *this) { return this→X; }
```

因此，下面函数的变量同名屏蔽效应可以通过显式调用this指针避免。
```
void Location::SetX(int X) {this→X=X; }
```

```
Fraction add(const Fraction &) const;
```
若将this指针现形，则相当于：
```
Fraction add(const Fraction *this, const Fraction &);
```

若定义了一个类，含有私有的int型数据成员a。为方便修改属性a，判断下列公有的设置函数的正误：

```
void set_a(int x) {a=x;}

void set_a(int a) {a=a;}

void set_a(int x) {this->a=x;}

void set_a(int a) {this->a=a;}
```

成员变量-数据-属性
成员函数-操作-方法

关于枚举类型，下列写法会出现什么问题？

```
enum month_set{January,February,……};
//若干同学不约而同地给枚举类型定义了相同的名字。

switch (pNow->tm_mon)    {
        case 0:month = January; break;
        case 1:month = February; break;
……};

void showtime(){cout<<month+1<<"月"; ……};
```

如果好心同事修改枚举定义为 January=1

@1班10名，2班13名同学练习了枚举类型的使用

@byD2120109

```
void Fraction::input(){
    while(1){cout<<"输入分子和分母"<<endl;
    cin>>Above>>Below;
    if(Below==0) cout<<"分母不为零"<<endl;
    else{
        reduction();
        return ;}
    }
}
```

有输入需求的时候，请给客户一点提示。

VS

@by大多数

```
void Fraction::input() { cin >> Above >> Below;}
```

若通分函数为传参方式 **void makeCommon(Fraction);**

A同学在通分函数里面完成了数学意义上的通分，即两个对象都改为同分母，然后写了下面这个相加函数，请问A同学的代码能得到正确的期望的结果吗？为什么？

```
Fraction Fraction::add(Fraction f2) {
    makeCommon(f2);
    Fraction result;
    result.above = above + f2.above;
    result.below = below;
    result.reduction();
    return result;
}
```

学霸B同学将A同学(左)的代码改为引用传递(右)。代码运行结果正确了，皆大欢喜了吗？

```
void fraction::makeCommond(fraction a)
{
    if(Below==a.Below)
    {
        return;
    }
    else
    {
        int x,y;
        int m,n;
        m=a.Below;
        n=a.Above;
        x=gcd(Below,a.Below);
        y=Below*m/x;
        Below=y;
        Above=Above*y/Below;
        a.Below=y;
        a.Above=n*y/m;
//a传值调用，返回后a被释放，调用处传进来的对象不受影响。
        return;
    }
```

```
    Above=Above/a;
    Below=Below/a;
}

void fraction::makeCommond(fraction &s1)//通分 ，此时
{
    int a,b,c;
    a=Below;
    b=s1.Below;
    if(a==b) return;

    c=Below*s1.Below;
    Above*=s1.Below;
    s1.Above*=Below;
    Below=c;
    s1.Below=c;
//引用调用，函数调用时引用的对象也被通分了。
}

fraction fraction::add(fraction s1)    //加
{
    fraction s;
```

类的封装性——面向对象编程思想的最基本特征。

通分函数要注意跟数学意义的通分有区别。

void makeCommon(const Fraction&); 传入的参数是另一个对象，比如说Fraction f1,f2；当f1.makeCommon(f2)时只完成f1自身的改变，而不应该去改变f2，一个对象不允许去擅自改变另一个对象。如果f2也要完成通分，那么应该再对f2调用一次通分函数，f2.makeCommon(f1)。

分数相等判断的函数实现

```
bool fraction::equal(fraction f2) {
    fraction temp(above,below);
    return ! (temp.sub(f2));
}
return ! (temp.sub(f2)).above;
return ((temp.div(f2)).above==(temp.div(f2)).below)?true:false;
return (temp.div(f2)==1)?true:false;
```

```
bool Fraction::lessThan(const Fraction &f) const {
    Fraction f1(above, below), f2(f.above, f.below);
    f1.makeCommon(f2);
    f2.makeCommon(f1);
    if (f1.above < f2.above)  return true;
    else return false;
}
```

注意，比较大小时候，符号的影响。

比如：2/9　2/(-9)　-2/9 -2/(-9)这四个数与1/4比较，你的代码运行结果如何？还有些同学比较两数的代码，两个分数不等的时候，相等，大于，小于的判断全都得到false。

面向对象的思想与基本概念！

```
void twolines(double x1,double y1,double x2,double y2,double
x3,double y3,double x4,double y4){
        LinearEquation ll(y1-y2,x2-x1,y3-y4,x4-x3,(y1-y2)*x1-
(x1-x2)*y1,(y3-y4)*x3-(x3-x4)*y3);
        if((y1-y2)*(x3-x4)==(y3-y4)*(x1-x2))
       {cout<<"两直线平行"<<endl;}
        else{cout<<ll.getX()<<endl;
        cout<<ll.getY()<<endl;
        }
}
```

利用 LinearEquation类来计算相交点，该计算独立于LinearEquation类或者说该计算是“LinearEquation 类”服务的一个客户。

还可以将点封装成Dot类(结构)

```
bool TwoLines(Dot &d0, Dot &d1, Dot &d2, Dot &d3, Dot &d4);
```

函数返回指针类型

tm *localtime(const time_t *tp);

time_t tp;　　time(&tp); //为tp赋值

~~tm nowTime;~~　　tm* pNow;

~~pNow=&nowTime;~~ ~~//pNow指针落地，指向有意义的地址，生命期在函数域内~~

pNow=localtime(&tp);　cout<<pNow→tm_hour;

更合理地应定义为*void localtime(tm *tm_time, const time_t *time);*

果然在Visual Studio 2010及以后的版本，新增了安全函数，改成localtime_s()，语法格式：

errno_t localtime_s(struct tm _tm,　const time_t *time);*

time_t tp;　　time(&tp); //为tp赋值

tm nowTime;　　tm* pNow;

pNow=&nowTime; //pNow指针指向有意义的地址

localtime_s(pNow, &tp);　cout<<pNow→tm_hour;

多文件结构问题

- 练习阶段，可以把完整的类定义放在一个.h头文件中，包括后面自己完成了模板类的定义后，可以方便自己去使用。
  - 如: **D040001_Time.h**
- 在大型工程中，养成声明与定义分离的习惯
  - 声明性质的内容放在头文件中，即所有成员函数仅在头文件中声明。
  - 定义性质的内容放在源文件中，即在源文件中完成所有成员函数的类外定义。

实践2：运算符重载

- 要求
  - 1.掌握运算符重载为成员函数的方法。
  - 2.理解友元函数和友元类。
  - 3.掌握运算符重载为友元函数的方法。
  - 4.使用静态成员。
- 提交程序
  - 1.习题4.8，改造Complex类。
  - 2.习题4.9，Renminbi类。
  - 3. 实验12，二-5，改造Fraction类。