实践2 小结

- 编程思维的变化
  - 面向过程：设计函数，数据作为参数传递给函数。
  - 面向对象：把数据和函数(对数据的操作)结合成封装成类，用类创建对象。
    - 对本对象数据的操作约束在类的成员函数中完成，提高安全性，可维护性(逐步扩展类的功能，如：Fraction V1.0， Fraction V2.0)。
    - 使用类对象时也不需要了解其内部如何实现(如cin， cout)。

实践2 小结

- 运算符重载的意义：
  - 对对象(object)的操作更直观，对程序员友好。
- 调试
  - A). 类定义完全了一步到位，开测，一蹴而就。
  - B). 新增一个函数，测试一个，步步为营。

➢ 调试程序之境界

1.有语法错误，编译器提示显示在出错行。
2.有语法错误，编译器提示显示在出错行后…N…行。
3.语法正确，链接出错了。
4.可以运行了。。。。。。

实践2 小结

- 成员函数代码在物理上(内存中)可以仅存储一份代码，但是逻辑上可认为每一个对象都有一份代码。所有对象共享成员函数，那成员函数怎么识别是哪个对象调用的？
  - 编译器在编译阶段，进行函数的重构，通过将this指针作为函数的第一个参数找到调用该函数的对象。
  - 因为this指针的存在，成员函数引用返回便捷。
- static静态成员变量存储于全局数据区，static静态成员函数逻辑上和物理(内存)上都只有一份代码，是类的成员，但受访问限定符修饰。但不属于具体的对象，静态成员函数没有this指针，只能访问static静态成员变量，不可以定义为只读成员函数。
- 友元函数/类不是类的成员，没有this指针，不受访问限定符修饰。

实践2 小结

- 具体问题
  - 一致性(consistently)需求，const &对象时(引用型参数应该在能被定义为const的情况下，尽量定义为const)，只能使用对象的只读操作(即只读成员函数)。
  - 体会友元函数、成员函数和普通函数在使用上的不同。如重载关系运算符和数学运算符时
    - 成员函数默认第一个操作数是本对象，用参数传入第二个操作数，故只需要一个形参。
    - 用普通函数和友元函数没有this指针，故必须要两个形参。

实践2 小结

- 几大难点疑点
  - 为什么有些运算符建议用友元/普通函数重载？
    - 用成员函数重载后某些运算不支持怎么解决？
  - 类成员函数返回引用，何时？如何？
  - 理解二元运算符函数调用时，对象的自动类型转换。
  - 为什么建议用构造函数完成从基本数据类型→自定义类的转换？

静态变量缺初始化定义的错误不多。因为是编译错误，提交的作业中只有4人缺少定义。

但是下面这种定义是大多数！

有的同学测试结果中已经显示num出现了负数，依旧没有完善类的定义。

```
Complex(double a=0,double b=0)
{Real=a; Image=b;  num++;}
~Complex(){num--;}
```

@by大多数(90%)

Complex(Complex &c){} *@byD2120122*

正确的同学：

D2120105，D2120213,15,16,21

```
Complex(double
{
    num+=1;
    ShowNum();
};
~Complex()
{
    num-=1;
    ShowNum();
};
```

若构造函数为(下列算式可执行的前提是构造函数可完成隐式类型转换)：

```
Complex(double r=0.0, double i=0.0):Real(r),Image(i){}
```

```
Complex operator+(Complex& c2) const;
```

c=c1+c2; √ c=c1+0.5; × c=0.5+c1; ×

```
Complex operator+(const Complex& c2) const;
```

c=c1+c2; √ c=c1+0.5; √ c=0.5+c1; ×

```
friend Complex operator+(const Complex& c1, const Complex& c2);
friend Complex operator+(Complex  c1, Complex c2);
```

c=c1+c2; √ c=c1+0.5; √ c=0.5+c1; √

第一个操作对象可以是非this对象的二元运算符，建议定义为友元函数（或普通函数）包括：算术运算符，关系运算符，流类运算符(必须非成员)

```
c=0.5+0.5;          //运算符+两侧都是实型，是否正确取决于构造函数的定义能否
                    //将求和结果(实型)转换为Complex类，与+的重载无关。
```

Why a Lvalue operator needs to be overloaded as a function operator that returns a reference?

◎ 如果只需要赋值一次，不需要连续赋值，可以返回void。

◎ 以值传递返回对象，可以实现连续赋值，但是return语句要把对象(即*this)拷贝到保存返回值的外部存储单元之中，将额外调用一次拷贝构造函数和析构函数，虽然功能仍然正确，但是增加了不必要的函数调用开销，降低赋值运算符的效率。

◎ 以引用返回对象，不仅可以实现连续赋值，而且运算符左侧的对象就是操作对象的副本，直接return一个已经存在的对象的引用，不存在额外的函数调用和拷贝过程。

```
void operator=(Complex& c);
```

c0=c1; √ c2=0.5; × c0=c1=c2; ×

```
Complex operator=(const Complex& c);
Complex operator=(Complex c);
```

c0=c1; √ c2=0.5; √ c0=c1=c2; √

```
Complex & operator=(const Complex& c);
```

c0=c1; √ c2=0.5; √ c0=c1=c2; √

//且执行/存储效率更高

左值运算符（运算对象在运算符左边，如=，+=，前置++，[]等）重载时，this对象随运算而改变，要求返回引用——程序效率更高或必须返回引用。

请问非左值运算符(如 +，-，*，/等）重载时，
问题1：是否也可以返回引用？问题2：为什么？

```
Complex  &Complex::operator+(const Complex &c2) const
{return  Complex(Real+c2.Real, Image+c2.Image);}
```

问题3：不限于运算符，如果实在需要返回非this对象的引用，该如何实现？

除了上学期讲过的两种，上次实验课果然现场抓到有同学想到第三种方法了。

```
Renminbi(int y=0,int j=0,int f=0)
:yuan(y),jiao(j),fen(f){}   // int   --> Renminbi
operator float(){return yuan+0.1*jiao+0.01*fen;}
                                    // float <-- Renminbi
```

```
return yuan+jiao/10+fen/100;
```

r=r1+2;

①+，发现第一个参数类型不是基本类型参数。但是发现Renminbi类定义了类型转换函数，于是将r1和2都转换为float类型进行运算。

②=，发现构造函数带默认参数，可以将int型转换为Renminbi型，将第一步运算得到的float型3.23转换为int型的3后，再构造为(3,0,0)，然后调用默认=赋给r。

不理解的同学，自行复习1.6.4.

再多一问：调用几次构造函数？

重载+　　*Renminbi operator+(Renminbi r2);*
然后执行　*r=r1+2;*

- 发现 + 运算符无法匹配合适函数，或者说是有两个函数都可匹配。
- 双向类型转换引起的二义性问题。

下面哪个重载正确？

```
Renminbi operator+(int y)
{yuan+=y;  return Renminbi(*this);}

Renminbi operator+(int y)
{return Renminbi(yuan+y, jiao, fen);}
```

重新定义，仅“自定义类→基本类型”单向转换

```
class Renminbi{
    ……
    Renminbi(int y,int j,int f);
    Renminbi();
    operator float();
    Renminbi operator+(Renminbi r2);
    Renminbi operator+(int y);
};
```

r=r1+2;　　可执行吗？

float f2;　r=r1+f2;　可执行吗？

- 建议只进行基本类型→自定义类型的转换。

```
Renminbi Renminbi::operator+(const Renminbi &c)const{
    Renminbi c1(*this);
    Renminbi c2(c);
    int yuan,jiao,fen;
    yuan=(int)(float(c1)+(float)c2);
    jiao=(int)((float(c1)+(float)c2)*10)%10;
    fen=(int)((float(c1)+(float)c2)*100)%10;
    return Renminbi(yuan,jiao,fen);
}
```

@byD2120220

VS

```
Renminbi& Renminbi::operator+(const Renminbi& r)
{
        yuan += r.yuan;  jiao += r.jiao;
        fen += r.fen;       return *this;
}
```

@by大多数

Fraction类

- 定义好的Fraction类提供给客户使用时，需要保证客户用这个Fraction类进行各种运算时的正确性。
- 类成员函数相互配合，组合使用，配合不当，必然出现运算的错误。
- 提议将符号约束在分子并不是规定，只是一种防止错误的措施，类的封装性可以使程序员在定义类时可以总体规划类的成员函数的设计。

```
Fraction f1(0),   f2=f1.reciprocal();
Fraction f3(2,-9),  f4(1,4);
```

//解决方案一：使符号体现在分子上

```
Above = (Below > 0 ? 1 : -1) * Above;
Below = abs(Below);
```

完备了吗?

//解决方案二：定义标准化成员函数

```
void normalize() {
    if(0==Above*Below)  //学习异常处理前，简化约定
        Above=0,                    Below=1;
    else if(above*below<0)
        Above= - abs(Above),  Below=abs(Below);
    else
        Above=abs(Above),     Below=abs(Below);
}
```

注意，防止出现除零，尤其注意求倒数函数

鉴于向基本类型的转换容易引起二义性，采用函数调用方式进行类型转换。

```
int Fraction::toInt() const { return Above / Below; }

double Fraction::toDouble() const { return 1.0 * Above / Below; }

string Fraction::toString() const {
    char ssA[10], ssB[10], ssD[2]="/";   //要求测试数据在9位数以内整型
    string ss;      itoa(Above,ssA,10);  // 包含<cstdlib>
    if(Below>1) {     //要求各种具有改变私有成员功能的函数能保证符号显示
                      //在分子上，且防止出现分母为0
       itoa(Below,ssB,10);
        ss=string(ssA)+string(ssD);  ss+=string(ssB);
       //string类重载了+, +=，相对c风格字符串处理更便捷，代价是程序运行效率
    } else ss=ssA;   //分母为1时仅显示分子
    return ss; }
```

实践3：类与对象的应用

- 要求
  - 1.熟练掌握类与对象的声明、定义、应用。
  - 2.掌握流操作运算符的重载。
  - 3.掌握对(文本)文件流对象的稳健操作。
- 提交程序
  - 1.利用实验10.1的集合类，求两个集合的交集。(必做)
  - 2.习题5.11，5.12，完善给出的字符串类MyString，并完成常用运算符的重载。(必做!!!好好完善MyString类，这个类将为你的后续作业提供服务，类定义的如何直接影响后续作业的完成。)
  - 3.设计类Loan，并在利率固定的假设条件下，用该类给出两种常用的还贷方式下的还款方案。（选做）

编程要求（还贷计算器）

- 设计Loan类，
  - 数据(属性)包括：贷款额(万元),还款期限(年)，年利率(%)…
  - 操作(行为)包括：分别计算下面两种还贷方式下的月还款额，月还本金数，月还利息数，并计算总还款额，每月还款后本金余额等
  - 等额本金（本金保持相同，利息逐月递减，月还款数递减）
  - 等额本息（本金逐月递增，利息逐月递减，月还款数不变）

例	贷款额(万元)	年限	年利率%	月利率%	总还款(万元)
等额本息	50	10	7	0.583333	69.67
等额本金	50	10	7	0.583333	67.65

编程要求（还贷计算器）

两种算法的共同点：

贷款月利率 = 年利率/12

每月还款额 = 每月还款本金+每月还款利息

每月还款利息 = 贷款本金余额*贷款月利率

等额本金算法：

每月还款本金=贷款总额/贷款月数 。

贷款本金余额=贷款总额-已还款月数*每月还款本金

等额本金输出示例:

贷款额6w元，还款期限5年，贷款年利率**，

还款方式：等额本金

每月还款额 | 每月还款本金 | 每月还款利息 | 贷款本金余额

==========

1000+利息　1000　60000*月利率　59000

。。。。。。

等额本息算法：

总还款额=每期还款额*还款月数。

$$每月还款 = 本金 \times 月利率 \times \frac{(1+月利率)^{总还款月数}}{(1+月利率)^{总还款月数}-1}$$

等额本息输出示例:

贷款额20w元，还款期限20年，贷款年利率4.2%，

还款方式：等额本息

每月还款额 | 每月还款本金 | 每月还款利息 | 贷款本金余额

==========

1233.14　533.14　700　199466.86

。。。。。。

编程要求（还贷计算器）

- 以贷款总额50万元，10年，年利率5.39%为例，比较两种方案还款总额的差值。
- 以贷款总额5万元，1年，年利率7%为例，输出分期还款的计划，输出范例见cpp文件。
  - 编写程序，输出到txt文件（注意：存储时各数字的分隔，否则读取时无法实现）
  - 编写程序，读取文件数据，输出到console窗口。（注意文件的命名，保证你们所有人的程序在一个文件夹下也能找到各自的文件）

注意阅读编程要求

支付宝
计算器
范例→