目录：

面向对象思想的体现：

第一，继承的思想。Search 类继承于 Sort。Calculator 类继承于 Complex 类。

第二，函数的重载。

第三，面向对象程序设计封装的思想。

第四，类的组合。

第五，构造函数初始化列表的使用

第六，const 的使用，参数的 const 以及返回 const 型。

第七，reference 的使用。

第八，namespace 的使用。

第九，友元函数的使用。

第十，运算符重载。

第十一，c++ 下的强制转换

一、需求分析：

matlab 程序主要实现了四则运算，积分运算，矩阵运算，排序运算，查找运算，和离开等操作。

四则运算应该能实现不同的四则运算，如一般的实数，复数，分数。

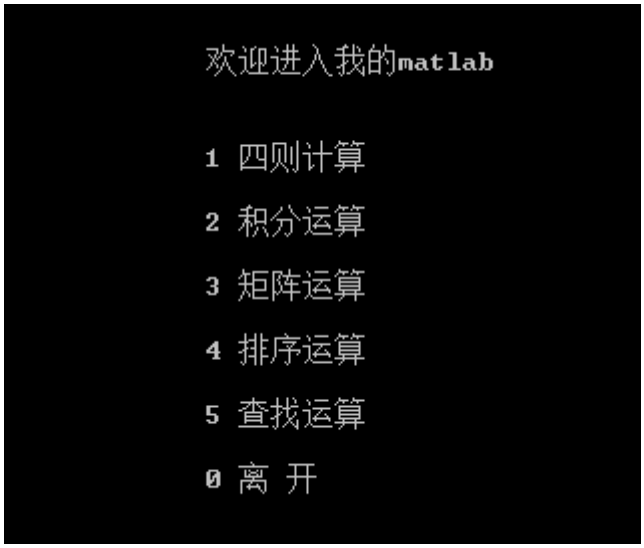
积分运算应有不同的算法逼近，比较不同算法的优缺点。

排序算法也应有不同的算法来实现，比较不同算法的效率。

查找运算可以先排序在查找或者可以直接查找。

二、界面分析：

运行程序可以看到如下界面：



可以看到主要完成五个部分的 matlab 内容，其中四则运算包括了简单的四则运算，复数的四则运算，和分数的四则运算。

三、类的分析：

为了实现上面的功能，我们建立了以下几个类，Calculator 类，Complex 类，Fraction 类，Integrator 类，Matrix 类，Search 类，Sort 类。他们的存在是显然的，每个类完成不同的功能，其中几个类之间又有互相的交流来完成他们共同的任务。对于他们的实现以及为何这么实现我做了如下分析：

1、Calculator 类

①简单四则运算：

主要完成四则运算的功能，刚开始时我们想到了简单的加减乘除四则运算，并运用数据结构中栈的知识写成了一个四则运算的函数，主要完成+，-，*，/，（，和）他们这些符号所组成的运算，各个符号的优先级是显然的。

栈结构的运用：栈是限定只能在表的一端进行插入和删除操作的线性表。允许插入和删除的一端称为栈顶(top)，另一端称为栈底(bottom)。

1）高级语言中的表达式是用人们熟悉的公式形式书写的，编译系统要根据表达式的运算顺序将它翻译成机器指令序列。

2）为解决运算顺序问题，把运算符分成若干等级，称为优先数。

3）为进行表达式的翻译，需设立两个栈，分别存放操作数(NS)和运算符(OS)。

4）首先在 OS 中放入表达式结束符“；”，然后自左至右扫描表达式，根据扫描的每一个符号作如下不同处理：

① 若为操作数，将其压入 NS 栈

② 若为运算符，需看当前 OS 的栈顶元素：

若 OS 栈顶运算符小于当前运算符的优先数，则将当前运算符压入 OS 栈。

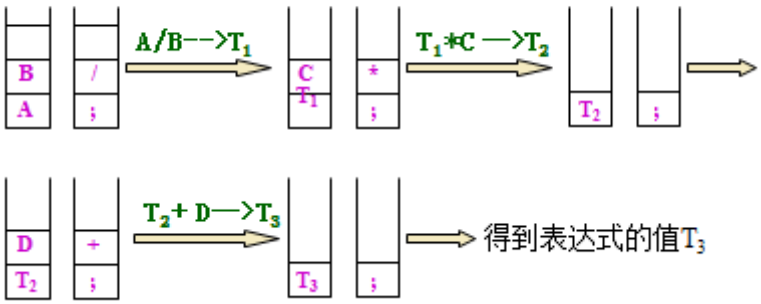
若 OS 栈顶运算符大于等于当前运算符的优先数，则从 NS 栈中弹出两个操作数，设为 x、y，再从 OS 栈中弹出一个运算符，设为θ，由此构成一条机器指令：

y θ x → T，并将结果 T 送入 NS 栈。

若当前运算符为“；”，且 OS 栈顶也为“；”，则表示表达式处理结束，此时 NS 栈顶的元素即为此表达式的值。

例如：

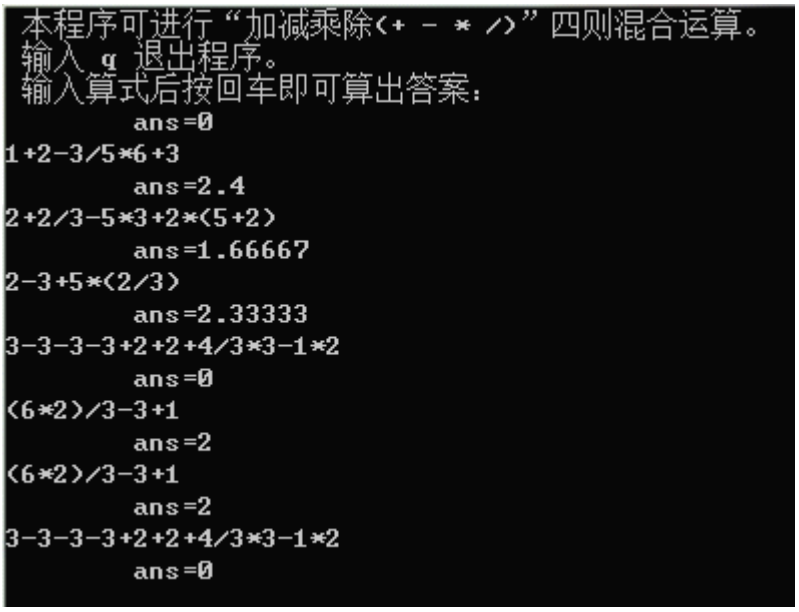
设表达式为：A / B * C + D ； 过程为：



运用这个算法就可以实现我的简单的加减乘除四则运算！

然而到了后期，在经过课堂学习和课后做作业之后，我们觉得四则运算并不一定要局限于简单的整数和浮点数计算，还可以有复数和分数的四则运算。由此引出了我对以下两个类的分析。

简单四则运算运行结果如下：



可以看到运行结果正确。

并且看到visual studio2010非常的智能，只要按上下键就能重新将上一条指令放到缓冲区中，然后按回车进行计算，这与真实matlab中的功能非常相似，也非常有必要。以上结果中最后两条指令（6*2）/3-3+1和3-3-3-3+2+2+4/3*3-1*2就是以这种方法进行的。

②复数的四则运算：

首先，继承这种语法满足了程序的渐增式开发的要求。程序开发是一个渐增的过程，就像我们的学习的过程。而继承通过在原先正确的代码的基础上增加一些新的代码。

由 Calculator 类的头文件中可以看到，它是 Complex 类的一个子类，可以完成复数的四则运算：

```
friend const Calculator operator+(const Calculator &add1, const Calculator &add2);
friend const Calculator operator-(const Calculator &sub1, const Calculator &sub2);
friend const Calculator operator*(const Calculator &c1, const Calculator &c2);
friend const Calculator operator/(const Calculator &c1,const Calculator &c2);
friend ostream& operator<<(ostream& os, const Calculator& com);
friend istream& operator>>(istream& is, Calculator& com);
```

由这些代码也可以看出我重新定义了Complex类中的一些成员函数， 由面向对象**Name Hiding**的规则可以知道，**在基类中定义了一个函数被重载（可以重载overloading），在派生类中重定义（redefine）或重载（overriding）基类的函数，派生类函数会掩盖所有基类版本，“本地优先”**

因此在派生类中再调用此类函数，以重载的函数为主。

并且，**我们知道派生类会自动继承一些基类的函数**，如我们的程序中未对set()函数进行冲定义或者重载，但是在Calculator.cpp文件中可以看到我们任然用了函数set()，说明编译器自动继承了这些函数。

然而，我们也可以知道编译器不会自动继承一些函数， 主要有一下五类：

A、构造函数（constructor）

B、析构函数（destructor）

构造函数和析构函数是用来处理对象的创建和析构的，它们只知道对在它们的特殊层次的对象做什么。所以，在整个层次中的所有的构造函数和析构函数都必须被调用，也就是说，

构造函数和析构函数不能被继承。

C、operator= 。operator= 也不能被继承，因为它完成类似于构造函数的活动。这就是说，尽管我们知道如何由等号右边的对象初始化左边的对象的所有成员，但这并不意味着这个初始化在继承后仍有意义。

D、overloading operator new

E、friend relationship

在继承过程中，如果我们不亲自创建这些函数，编译器就综合它们。被综合的构造函数使用成员方式的初始化，而被综合的operator= 使
用成员方式的赋值。

因此我们在Calculator类中重定义(redefine)了构造函数，并运用了基类的构造函数和构造函数参数化列表来定义！如下：

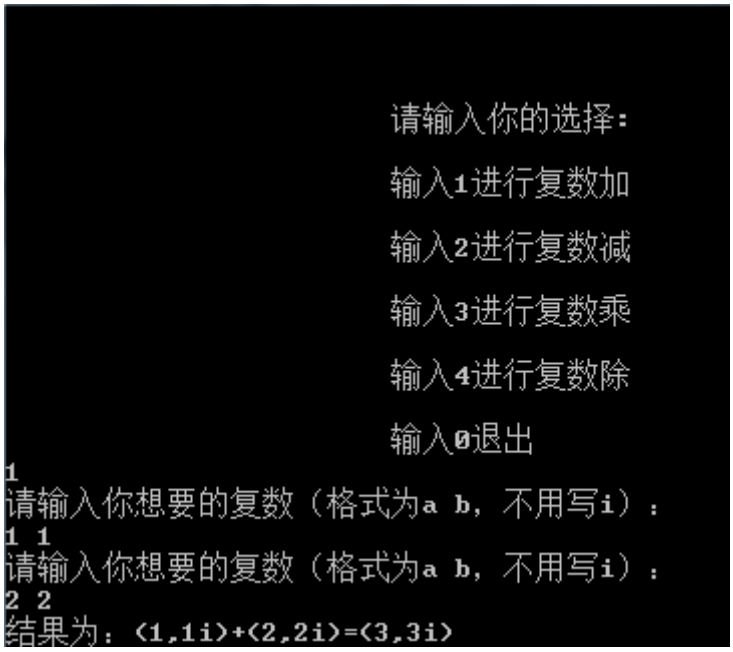
```
Calculator(double rp=0.0,double ip=0.0):Complex(rp,ip)
{ }
```

我们组对类的数据成员申明为protected型的看法：

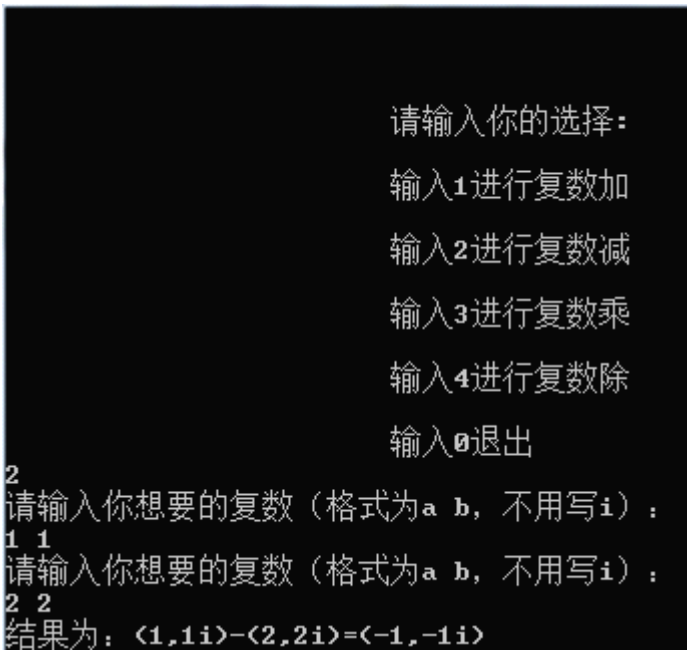
虽然申明为protected型，是数据成员的封装性收到破坏。但是可以看到基类的私有成员被派生类继承（不论是私有继承、公有继承还是保护继承）后变为不可访问的成员，派生类中的一切成员均无法访问它们。因此如果需要在派生类中引用基类的某些成员，应当将基类的这些成员声明为protected，而不要声明为private。

复数运算运行结果：

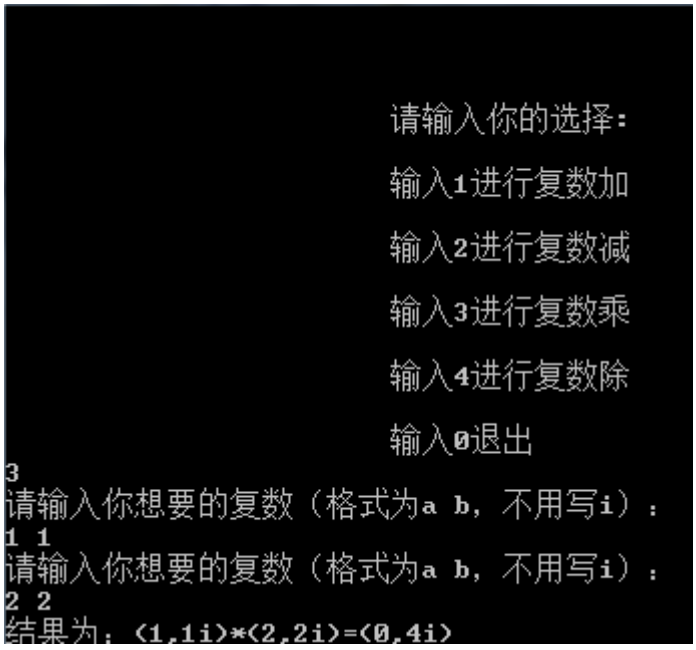
复数加：



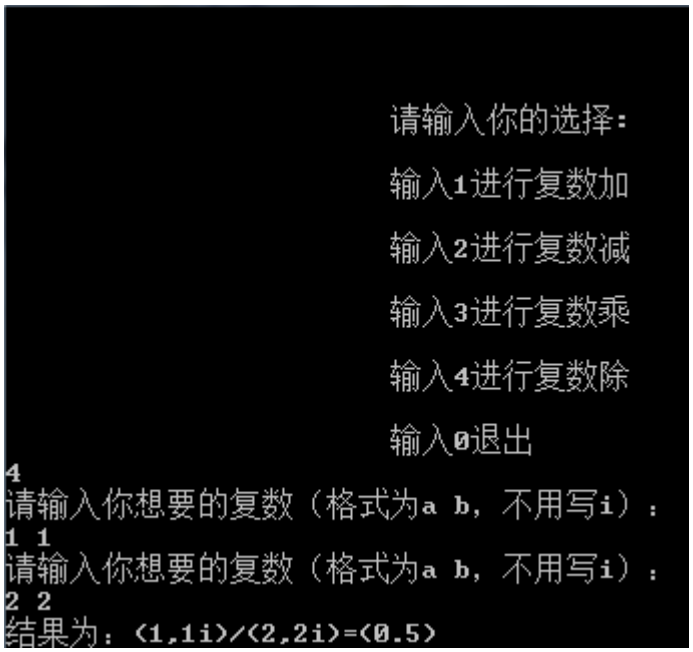
复数减：


请输入你的选择:
输入1进行复数加
输入2进行复数减
输入3进行复数乘
输入4进行复数除
输入0退出
2
请输入你想要的复数（格式为a b，不用写i）：
1 1
请输入你想要的复数（格式为a b，不用写i）：
2 2
结果为：(1,1i)-(2,2i)=(-1,-1i)

复数乘：


请输入你的选择:
输入1进行复数加
输入2进行复数减
输入3进行复数乘
输入4进行复数除
输入0退出
3
请输入你想要的复数（格式为a b，不用写i）：
1 1
请输入你想要的复数（格式为a b，不用写i）：
2 2
结果为：(1,1i)*(2,2i)=(0,4i)

复数除：



③分数四则运算：

其实分数的四则运算与复数的四则运算类似，但是为了用不同的方法来完成这一个任务，我们主要用了数据成员类的方法来完成分数四则运算，在 Calculator 类的成员数据中定义：

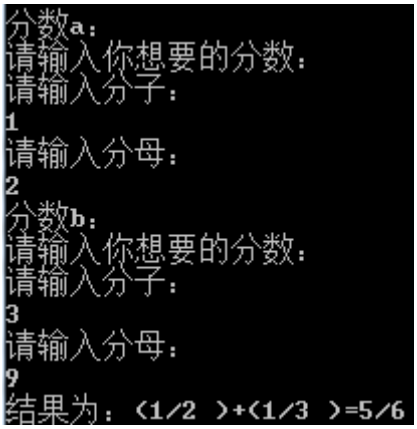
Fraction a1;

Fraction b1;

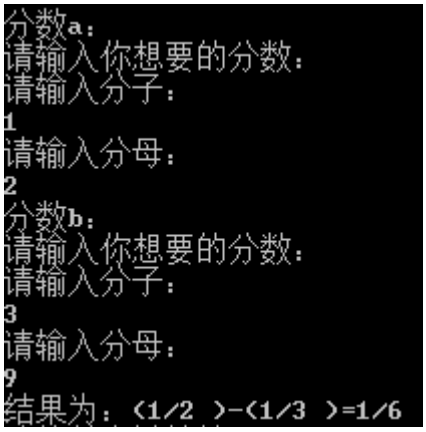
就可以用Fraction类型的两个数据了。Fraction类的具体实现过程见下面分析。

分数四则运算结果：

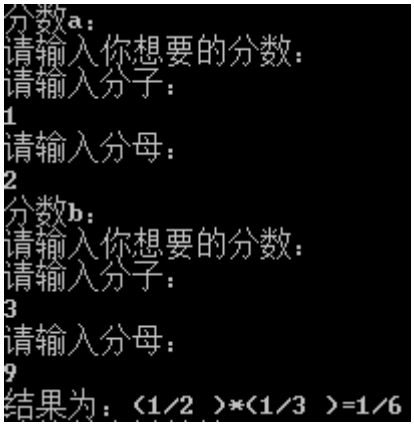
分数加：



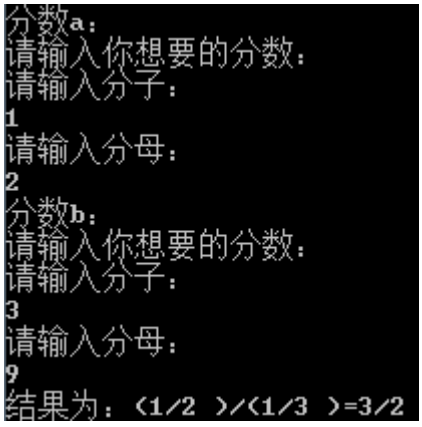
分数减：



分数乘：



分数除：



2、Fraction 类--对返回效率（the return optimization）的讨论

首先，这个类老师上课已经布置过作业了，我也做的很认真，觉得这个类很能体现面向对象的思想，所以修改了一下之后便运用到我们的 project 中了。

这个类主要可以完成分数的设定，分数的约分，分数的加减乘除，分数的大小比较，以及分数的输入输出运算。

函数重载是面向对象中一个重要的思想。我们希望用户自定义的数据类型也与我们原先的运算有相似的规律，因此重载加减乘除运算符是有必要的。

Fraction类的头文件代码中：

```
friend const Fraction operator+(const Fraction& x,const Fraction& y);
```

```
friend const Fraction operator-(const Fraction& x,const Fraction& y);
friend const Fraction operator*(const Fraction& x,const Fraction& y);
friend const Fraction operator/(const Fraction& x,const Fraction& y);
friend const Fraction operator+(const Fraction& x);
friend const Fraction operator-(const Fraction& x);
const bool operator<(const Fraction& x);
const bool operator<=(const Fraction& x);
const bool operator>(const Fraction& x);
const bool operator>=(const Fraction& x);
const bool operator==(const Fraction& x);
const bool operator!=(const Fraction& x);
friend ostream& operator<<(ostream& os, Fraction& com);
friend istream& operator>>(istream& os, Fraction& com);
```

可以看出，我主要运用 friend 函数和类的成员函数来完成重载。

参数列表中大量运用了 reference 和 const 类型，可以提高代码的效率和保证类的数据成员不被修改。具体分析我以重载+来举例：

```
friend const Fraction operator+(const Fraction& x,const Fraction& y);
```

由上面函数原型可以看到返回类型是const Fraction，const可以保证a+b=c这样的代码是不能编译成功的，如果没有这个const则会编译正确，这和我们主观的认识是相悖的，虽然像a+b=c这样的代码页没有什么不对之处，但为了与我们主观认识相符，我们尽量避免了这样的代码。

由于函数声明为friend类型的，因此对于类来中operator+是一个全局函数，a+b则会被翻译为operator+（a，b）。

返回效率的讨论：以重载的+为举例：

```
const Fraction operator+(const Fraction& x,const Fraction& y)        //此处可以完成c=a+4,c=4+a,即完成了类型转换和加法交换律
{
    Fraction c(x.numerator*y.denominator+y.numerator*x.denominator,x.denominator*y.denominator);
    c.reduction();
    return c;
}
```

粗略得看，我这个函数的返回效率不太高，但是这里c.reduction()我主要完成了分数的约分，所以这样的代码是不可避免的。如果没有c.reduction()这个函数，则可以直接写成如下的方式：

```
const Fraction operator+(const Fraction& x,const Fraction& y)        //此处可以完成c=a+4,c=4+a,即完成了类型转换和加法交换律
{
    Return
Fractionc(x.numerator*y.denominator+y.numerator*x.denominator,x.denominator*y.denominator);
}
```

首先看第一种方式：

①首先，c对象被创建，与此同时他的构造函数被调用。

②然后，拷贝构造函数吧c拷贝到返回值外部存储单元里。

③最后，当c在作用域的结尾时调用析构函数。

因此返回效率较低。

第二种方式：编译器直接把这个对象创建在返回值外面的内存单元。因为不是真正的创建一个局部对象，所以仅需要单个的普通构造函数调用（不需要拷贝构造函数），效率比较高！

3、Complex 类

现将 Complex 类头文件中成员函数主要实现的功能列于下：

```
friend const Complex operator+(const Complex &add1, const Complex &add2);
friend const Complex operator-(const Complex &sub1, const Complex &sub2);
friend const Complex operator*(const Complex &c1, const Complex &c2);
friend const Complex operator/(const Complex &c1,const Complex &c2);
friend ostream& operator<<(ostream& os, const Complex& com);
friend istream& operator>>(istream& is, Complex& com);
```

可以看到 Complex 类主要实现过程与 Fraction 类相似，因此，在次我不在赘述！

但是不同的是，可以从 Complex.h 头文件中看到我将成员数据设定为 protected，也即为了能让它的子类---Calculator 类使用这些数据成员。

4、Integrator 类 ---对 c++中类型转换的分析

Integrator 类主要完成了 matlab 运算中的积分运算。积分运算就需要一个函数，一个积分的上下限。因此我们定义了 choice，min，和 max 三个数据成员，choice 主要选择需要积分的函数，min 和 max 确定了积分的上下限。

对 c++中类型转换的讨论：

C++中提供了四种类型转换方式，最直观的理解如下：

dynamic_cast:通常在基类和派生类之间转换时使用；

const_cast:主要针对const和volatile的转换.

static_cast:一般的转换，如果你不知道该用哪个，就用这个。

reinterpret_cast:用于进行没有任何关联之间的转换，比如一个字符指针转换为一个整形数。

static_cast

用法：static_cast < T> (V)

该运算符把V转换为T类型，但没有运行时类型检查来保证转换的安全性。它主要有如下几种用法：

用于类层次结构中基类和子类之间指针或引用的转换。进行上行转换（把子类的指针或引用转换成基类表示）是安全的；**进行下行转换（把基类指针或引用转换成子类表示）时，由于没有动态类型检查，所以是不安全的。**

用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。

把空指针转换成目标类型的空指针。

把任何类型的表达式转换成void类型。

注意：static_cast不能转换掉V的const、volitale、或者__unaligned属性。

dynamic_cast

用法：dynamic_cast < T> (V)

该运算符把V转换成T类型的对象。T必须是类的指针、类的引用或者void *；如果T是类指针类型，那么V也必须是一个指针，如果T是一个引用，那么V也必须是一个引用。

dynamic_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。

在类层次间进行上行转换时，dynamic_cast和static_cast的效果是一样的；在进行下行转换时，dynamic_cast具有类型检查的功能，比static_cast更安全。

```
class B{
public:
int m_iNum;
virtual void foo();
};
class D:public B{
public:
char *m_szName[100];
};

void func(B *pb){
D *pd1 = static_cast<D *>(pb);
D *pd2 = dynamic_cast<D *>(pb);
}
```

在上面的代码段中，如果pb指向一个D类型的对象，pd1和pd2是一样的，并且对这两个指针执行D类型的任何操作都是安全的；但是，如果pb指向的是一个 B类型的对象，那么pd1将是一个指向该对象的指针，对它进行D类型的操作将是不安全的（如访问m_szName），而pd2将是一个空指针。另外要注意：B要有虚函数，否则会编译出错；static_cast则没有这个限制。这是由于运行时类型检查需要运行时类型信息，而这个信息存储在类的虚函数表（关于虚函数表的概念，详细可见<Inside c++ object model>）中，只有定义了虚函数的类才有虚函数表，没有定义虚函数的类是没有虚函数表的。

另外，dynamic_cast还支持交叉转换（cross cast）。如下代码所示。

```
class A{
public:
int m_iNum;
virtual void f(){}
};

class B:public A{
};

class D:public A{
};

void foo(){
B *pb = new B;
pb->m_iNum = 100;
D *pd1 = static_cast<D *>(pb); //copile error
D *pd2 = dynamic_cast<D *>(pb); //pd2 is NULL
delete pb;
}
```

在函数foo中，使用static_cast进行转换是不被允许的，将在编译时出错；而使用 dynamic_cast的转换则是允许的，结果是空指针。

reinpreter_cast

用法：reinpreter_cast<T> (V)

T必须是一个指针、引用、算术类型、函数指针或者成员指针。它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针（先把一个指针转换成一个整数，在把该整数转换成原类型的指针，还可以得到原先的指针值）。

const_cast

用法：const_cast<T> (V)

该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外，T和V的类型是一样的。T 必须是指针、引用或指向成员的指针的类型。

其中：volatile修饰符告诉编译程序不要对该变量所参与的操作进行优化.在两种特殊的情况下需要使用volatile修饰符:

①第一种情况涉及到内存映射硬件(memory-mapped hardware,如图形适配器,这类设备对计算机来说就好象是内存的一部分一样)。

②第二种情况涉及到共享内存(shared memory,既被两个以上同时运行的程序所使用的内存)。

常量指针被转化成非常量指针，并且仍然指向原来的对象；常量引用被转换成非常量引用，并且仍然指向原来的对象；常量对象被转换成非常量对象。

举如下一例：

```
class B{
public:
int m_iNum;
}
void foo(){
const B b1;
b1.m_iNum = 100; //comile error
B b2 = const_cast<B>(b1);
b2. m_iNum = 200; //fine
}
```

上面的代码编译时会报错，因为b1是一个常量对象，不能对它进行改变；使用const_cast把它转换成一个常量对象，就可以对它的数据成员任意改变。注意： b1和b2是两个不同的对象。

再如：

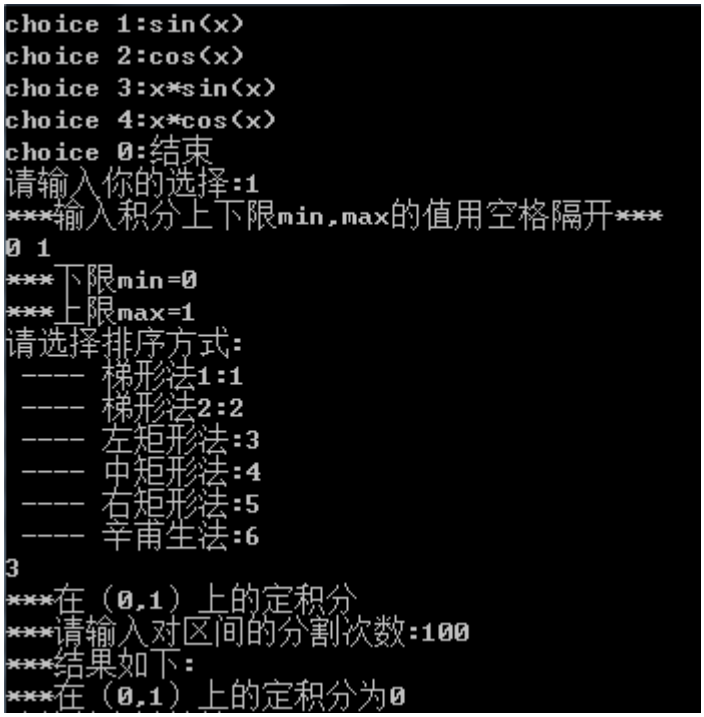
```
class A
{
public:
void f();
int i;
};
extern const volatile int* cvip;
extern int* ip;
void use_of_const_cast( )
{
```

```
const A a1;
const_cast<A&>(a1).f( ); //  去掉a1的const
ip = const_cast<int*> (cvip); //  去掉  const  和  volatile
}
```

在integrator类中列举了一些求积分的算法函数，其中每个函数中有一个相同点：就是需要类型转换的语句：h=(static_cast<double>(max)-static_cast<double>(min))/n;

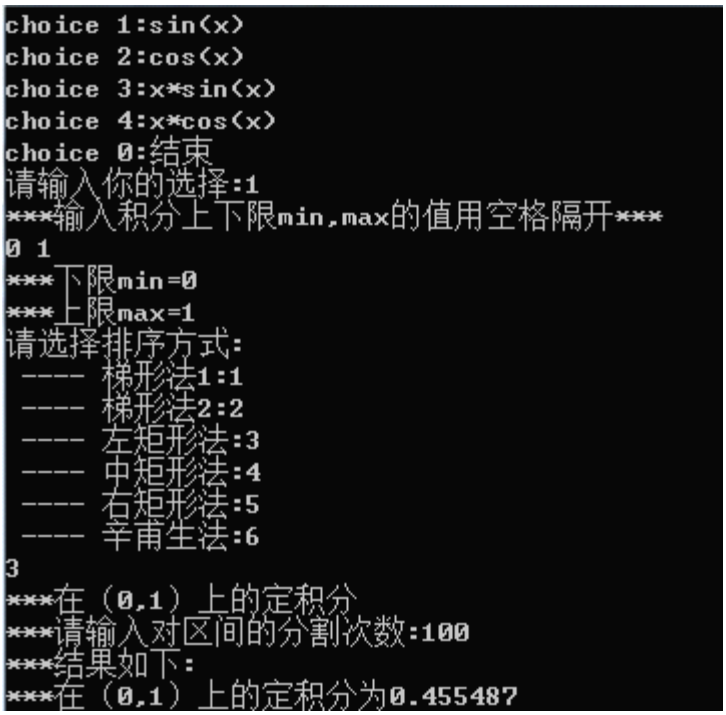
此时输出结果我以左矩形法来举例：

首先，当语句定义为h=(max-min)/n时，查看结果如下：

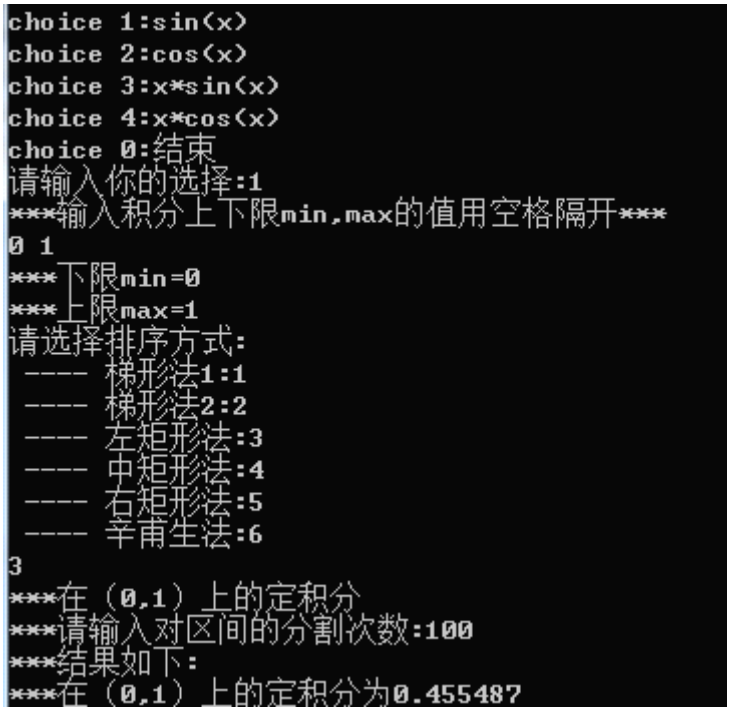


由上图可以看到当语句为h=(max-min)/n时，结果输出错误！可以看到进行下行转换时的危险性!!

当语句为h=((double)max-(double)min)/n时结果如下：



当语句为：h=(static_cast<double>(max)-static_cast<double>(min))/n;



由上可以看到语句h=((double)max-(double)min)/n与

语句h=(static_cast<double>(max)-static_cast<double>(min))/n的效果相同。

但是static_cast的好处由上面的分析也是很显然的。

关于积分函数的算法：

利用复合梯形公式实利用复合梯形公式实利用复合梯形公式实利用复合梯形公式实现定积分的计算现定积分的计算现定积分的计算现定积分的计算 假设被积函数为 f(x)，积分区间为[a，b]，把区间[a，b]等分成 n 个小区间，各个区间的长度为 h，即 h=(b-a)/n，称之为“步长”。根据定积分的定义及几何意义，定积分就是求函数 f(x)在区间[a，b]中图线下包围的面积。将积分区间 n 等分，各子区间的面积近似等于梯形的面积，面积的计算运用梯形公式求

解，再累加各区间的面积，所得的和近似等于被积函数的积分值，n 越大，所得结果越精确。以上就是利用复合梯形公式实现定积分的计算的算法思想。

复合梯形公式：

$$T_n=\frac{h}{2}\left(f(a)+2\sum_{i=1}^{n-1}f(x_i)+f(b)\right)$$ [2]

具体算法如下：

算法一

1：输入积分区间的端点值 a 和 b；

2：输入区间的等分个数 n（要求 n 尽可能大，以保证程序运行结果有较高的精确度）；

3：计算步长 h= (b−a)/n；

4：对累加和赋初值 T = (fa−f b)/2；

5：计算累加和 T = ∑f (xi)

6：算出积分值 Tn=T×h；

7：输出积分近似值 Tn。

利用 Smpson 公式实现定积分的计算实现定积分的计算实现定积分的计算实现定积分的计算假设被积函数为()fx，积分区间为[],ab，把区间[],ab 等分成 n 个小区间，各个区间的长度为 h。在复合梯形公式的基础上，构造出一种加速计算积分的方法。作为一种外推算法， 它在不增加计算量的前提下提高了误差的精度。

具体算法如下：

算法二

1：输入积分上限 b 和下限 a；

2：输入区间的等分个数 n（要求 n 尽可能大，以保证程序运行结果有较高的精确度）；

3：利用辛甫生公式：$S[n]=(4\times T[2n]-T[n])/3$，实现对定积分的求解（其中 T[2n]，T[n] 均为梯形公式计算所得的结果，由此可见辛甫生公式是以梯形公式为基础的）；

4：算出积分值 Sn；

5：输出积分近似值 Sn.

以上两种算法我们小组都已经实现，下面我列出一种我们小组研究过，但是这个算法比较复杂，我们小组最终未实现！

利用 Guass 公式实现定积分计算

Guass 型求积公式是构造高精度差值积分的最好方法之一。他是通过让节点和积分系数待定让函数 f(x)以此取 i=0,1,2....n 次多项式使其尽可能多的能够精确成立来求出积分节点和积分系数。高斯积分的代数精度是 2n-1，而且是最高的。通常运用的是-1---+1 的积分节点和积分系数，其他积分域是通过变换 x=(b-a)t/2　　+(a+b)/2 变换到-1 到 1 之间积分。

算法三

1：输入积分上限 b 和下限 a；

2：利用 Guass 公式，求定积分

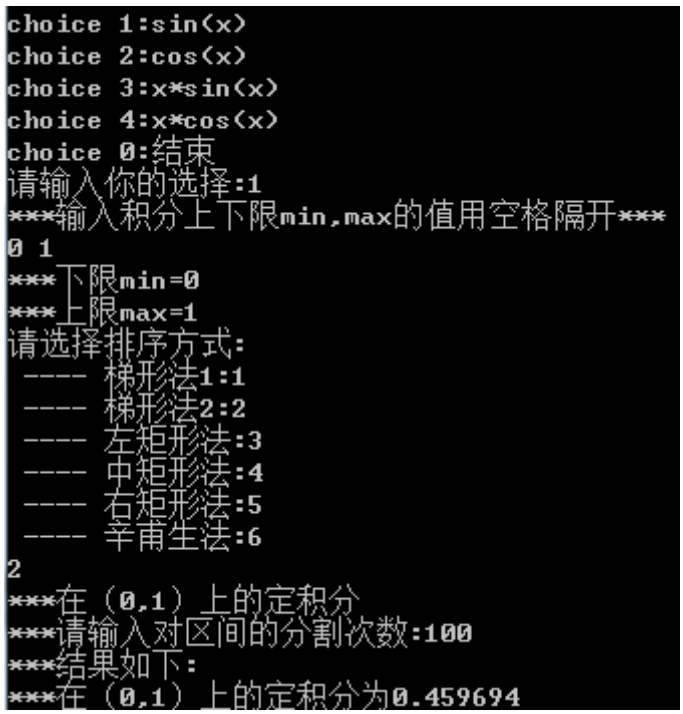
3：算出积分值 Sn；

4：输出积分近似值 Sn。

积分运算运行结果：

梯形法 1 运行结果：

梯形法 2 运行结果：



左矩形法运行结果：

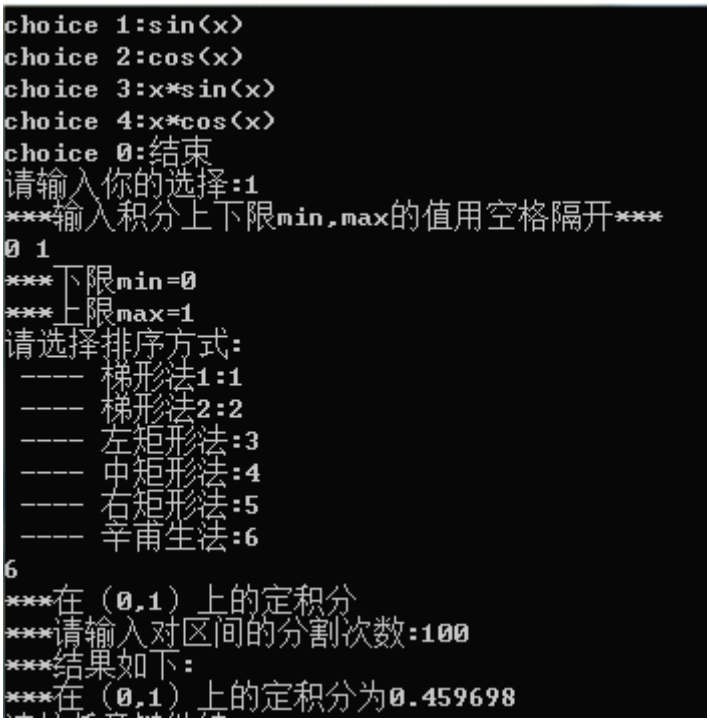
```
choice 1:sin(x)
choice 2:cos(x)
choice 3:x*sin(x)
choice 4:x*cos(x)
choice 0:结束
请输入你的选择:1
***输入积分上下限min,max的值用空格隔开***
0 1
***下限min=0
***上限max=1
请选择排序方式:
 ---- 梯形法1:1
 ---- 梯形法2:2
 ---- 左矩形法:3
 ---- 中矩形法:4
 ---- 右矩形法:5
 ---- 辛甫生法:6
3
***在（0,1）上的定积分
***请输入对区间的分割次数:100
***结果如下:
***在（0,1）上的定积分为0.455487
```

中矩形法运行结果：

```
choice 1:sin(x)
choice 2:cos(x)
choice 3:x*sin(x)
choice 4:x*cos(x)
choice 0:结束
请输入你的选择:1
***输入积分上下限min,max的值用空格隔开***
0 1
***下限min=0
***上限max=1
请选择排序方式:
 ---- 梯形法1:1
 ---- 梯形法2:2
 ---- 左矩形法:3
 ---- 中矩形法:4
 ---- 右矩形法:5
 ---- 辛甫生法:6
4
***在（0,1）上的定积分
***请输入对区间的分割次数:100
***结果如下:
***在（0,1）上的定积分为0.459694
```

右矩形法运行结果：

辛甫生算法运行结果：



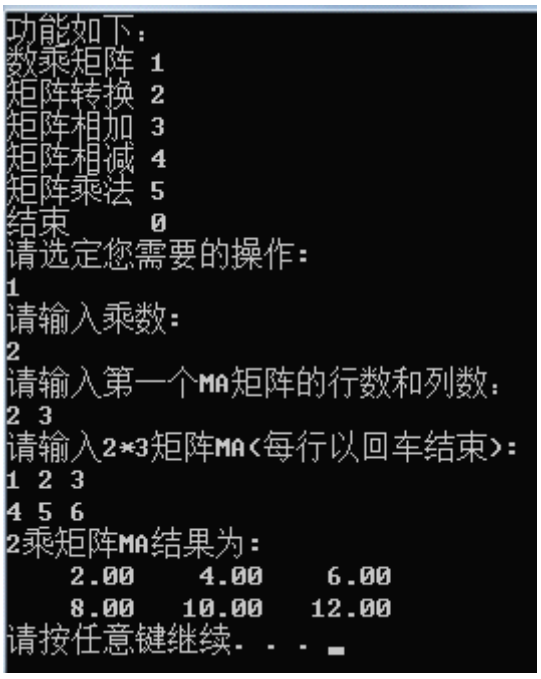
可以看到不同的算法运行结果不同，理论上说每种算法都是对定积分的一种近似模拟，所以都存在误差。其中梯形法 1 和梯形法 2 本质上是一个原理。

5、Matrix 类

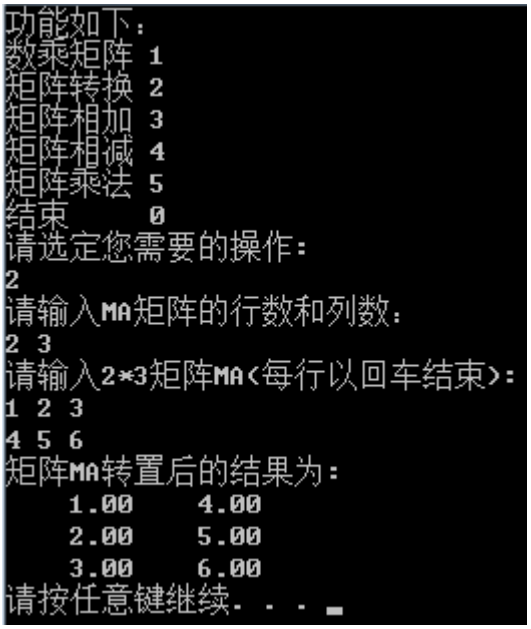
Matrix 类主要完成了 matlab 运算中的矩阵运算。

结果如下：

数乘矩阵：



矩阵转换：



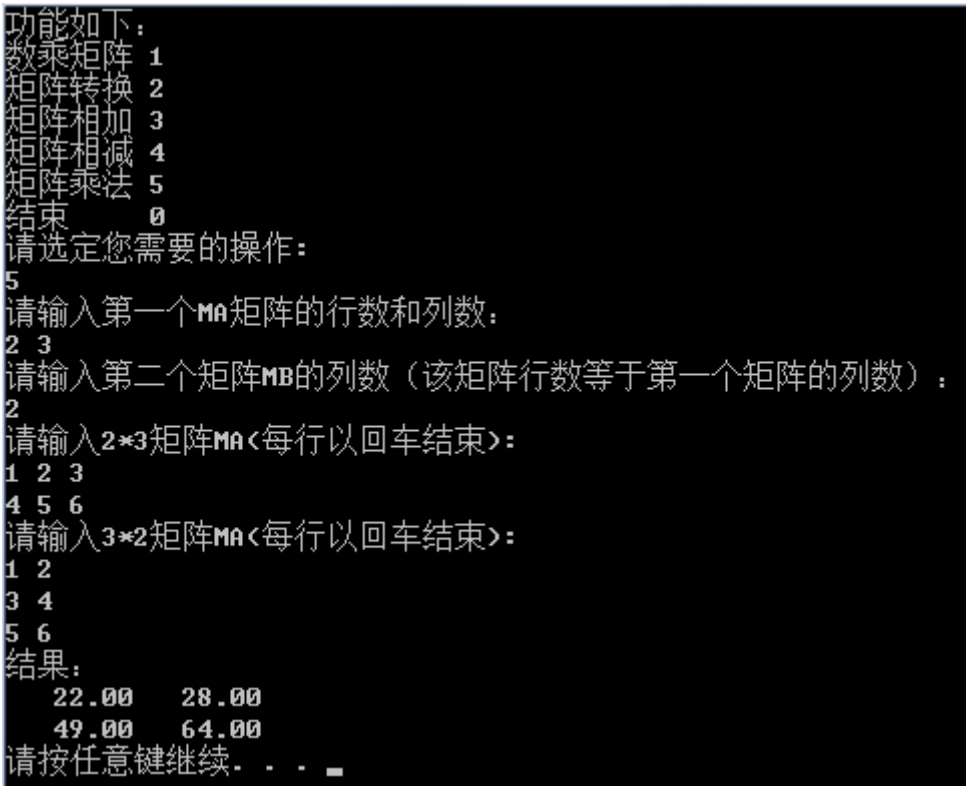
矩阵相加：

```
功能如下:
数乘矩阵 1
矩阵转换 2
矩阵相加 3
矩阵相减 4
矩阵乘法 5
结束     0
请选定您需要的操作:
3
请输入MA和MB矩阵的行数和列数:
2 3
请输入2*3矩阵MA(每行以回车结束):
1 2 3
4 5 6
请输入2*3矩阵MB(每行以回车结束):
7 8 9
1 2 3
结果:
    8.00   10.00   12.00
    5.00    7.00    9.00
请按任意键继续. . .
```

矩阵想减：

```
功能如下:
数乘矩阵 1
矩阵转换 2
矩阵相加 3
矩阵相减 4
矩阵乘法 5
结束     0
请选定您需要的操作:
4
请输入MA和MB矩阵的行数和列数:
2 3
请输入2*3矩阵MA(每行以回车结束):
1 2 3
4 5 6
请输入2*3矩阵MB(每行以回车结束):
7 8 9
1 2 3
结果:
   -6.00   -6.00   -6.00
    3.00    3.00    3.00
```

矩阵乘法：



其中矩阵乘法的算法比较复杂，具体如下：

```
void Matrix::Multiplication()
    {
        int i,j,k;
        cout<<"请输入第一个MA矩阵的行数和列数："<<endl;
        cin>>R1>>C1;
        cout<<"请输入第二个矩阵MB的列数（该矩阵行数等于第一个矩阵的列数）："<<endl;
        cin>>C2;
         R2=C1;
        double **a=NULL;
        a=(double**)malloc(R1*sizeof(double*));
        for(i=0;i<R1;i++)
            a[i]=(double*)malloc(C1*sizeof(double));
         cout<<"请输入"<<R1<<"*"<<C1<<"矩阵MA(每行以回车结束?):"<<endl;
        for(i=0;i<R1;i++)
            for(j=0;j<C1;j++)
                cin>>a[i][j];
        double **b=NULL;
        b=(double**)malloc(C1*sizeof(double *));
        for(i=0;i<C1;i++)
        b[i]=(double*)malloc(C2*sizeof(double));
         cout<<"请输入"<<R2<<"*"<<C2<<"矩阵MA(每行以回车结束?):"<<endl;
        for(i=0;i<C1;i++)
            for(j=0;j<C2;j++)
                cin>>b[i][j];
```

```
        double **MG=NULL;
        MG=(double**)malloc(R1*sizeof(double*));
        for(i=0;i<R1;i++)
            MG[i]=(double*)malloc(C2*sizeof(double));
        for(i=0;i<R1;i++)
            for(j=0;j<C2;j++)
                MG[i][j]=0;
        for(i=0;i<R1;i++)
            for(j=0;j<C2;j++)
                for(k=0;k<C1;k++)
                    MG[i][j]+=a[i][k]*b[k][j];
        cout<<"结果："<<endl;
        for(i=0;i<R1;i++)
        {
            for(j=0;j<C2;j++)
            cout<<setw(8)<<setiosflags(ios::fixed)<<setiosflags(ios::right)<<setprecision(2)<<MG[i][j];
            cout<<endl;
        }
        for(i=0;i<R1;i++)
        {
            free(a[i]);
            a[i]=NULL;
            free(MG[i]);
            MG[i]=NULL;
        }
        free(a);
        a=NULL;
        free(MG);
        MG=NULL;
        for(i=0;i<C1;i++)
        {
            free(b[i]);
            b[i]=NULL;
        }
        free(b);
        b=NULL;
    }
```

6、Sort 类

Sort 类主要完成了 matlab 运算中的排序运算。

首先根据不同的数据长度等情况，写不同的排序算法非常有必要，这对排序的效率等很有好处，本程序需要使用者自己确定自己的排序的方法。

由 Sort 类的头文件可以看到我们定义了 protected 类型的数据成员以实现数据的分装，

但是为了继承之后派生类对基类数据的使用，所以定义成了 protected 类型了。

关于排序运算的算法：

选择排序：

选择排序是不断在待排序序列（无序区）中按记录关键字递增（或递减）次序选择记录，放入有序区中，逐渐扩大有序区，直到整个记录区为有序区为止。

A、 简单选择排序：

1）过程：在当前无序序列中选择一个关键字最小的记录，并将它和最前端的记录交换。重复上述过程，使记录区的前端逐渐形成一个由小到大的有序区。

2）该算法由两层循环构成，外层循环表示共需进行 n-1 趟排序，内层循环表示每进行一趟排序需要进行的记录关键字间的比较。

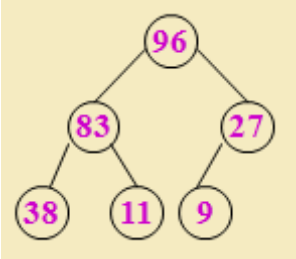
B、堆排序：

1）堆：设有序列{ $k_1,k_2,\ldots,k_n$ }，若满足：

$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \quad 或 \quad \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \quad (\ i=1,2,\cdots,\lfloor \frac{n}{2} \rfloor)$$

则称该序列构成的完全二叉树是一个堆。

例如，有序列：{ 96，83，27，38，11，9 }
构成的完全二叉树如下图所示,它是一个堆。



2）堆排序的过程由两部分组成：

① 将现有的序列构成一个堆

② 输出堆顶元素，重新调整元素，构成新 的堆，直到堆空为止。

3）堆的构造：

① 由所给序列生成对应的完全二叉树；

② 设完全二叉树 r[1..n]中结点 r[k]的左右子树均为堆，为构成以 r[k]为根结点的堆，需进行调整。方法是将 r[k]的值与其左右子树的根结点进行比较，若不满足堆的条件，则将 r[k]与其左右子树中根结点大的交换，继续进行比较，直到所有子树均满足堆为止。

③ 对于一个无序序列 r[0..n-1]构成的完全 二叉树，只要从它最后一个非叶子结点: [(n-2)/2]开始直到根结点为止，逐步进行上述调整，即可将该完全二叉树调整为堆。

4）堆排序：由以下两个步骤进行：

① 由给定的无序序列构造堆；

② 将堆顶元素与堆中最后一个元素交换，然后将最后一个元素从堆中删除，将余下元素构成的完全二叉树重新调整为堆，反复进行，直到堆空为止。

插入排序：

插入排序是将当前无序区中最前端的记录插入到有序区中，逐渐扩大有序区，直到所有记录都插入到有序区中为止。

C、线性插入排序：

过程：在有序区中进行顺序查找以确定插入的位置，然后移动记录腾出空间，以便相应关键字的记录插入。

对半插入排序：

与线性插入排序相比，除了采用的是对分查找插入的位置外，其它类似

D、希尔排序：

希尔排序属于插入排序，基本思想是：设线性表有 n 个结点，首先取一个整数 h < n 作为间隔，将线性表分为 h 个子表，所有距离为 h 的结点放在同一个子表中，在每一个子表中分别进行插入排序。然后缩小间隔 h，例如取 h =⌈h/2⌉，重复上述的子表划分和排序工作。直到最后取 h = 1，将所有结点放在同一个序列中排序为止。

开始时 *h* 的值较大，子序列中的对象较少，排序速度较快；随着排序进展，*h* 值逐渐变小，子序列中对象个数逐渐变多，由于前面工作的基础，大多数对象已基本有序，所以排序速度仍然很快。

h 的取法有多种。最初 shell 提出取 *h* = [*n*/2]，*h* = [*h*/2]，直到 *h* = 1。后来 knuth 提出取 *h* =[*h*/3] +1。还有人提出都取奇数为好，也有人提出各 *h* 互质为好。

希尔排序的增量序列的确定是一个很复杂的数学问题，共同的原则是

(1) 序列中各值没有除 1 之外的公因子；

(2) 最后一个增量必须为 1

交换排序：

交换排序是根据序列中两个结点关键字的比较结果，来对换在序列中的位置。

算法的特点是将关键字较大的结点向序列的尾部移动，关键字较小的结点向序列的前部移动。

E、冒泡排序

1）过程：先将第 1 个记录和第 2 个记录进行比较，若不满足则交换二个记录，然后比较第 2 个和第 3 个，依此类推，直至对第 n-1 个和第 n 个记录比较并处理完。上述过程称为一趟冒泡排序。其结果是将最大的记录被交换到最后一个位置。重复以上过程，但第 2 趟只需对前 n-1 个记录进行排序，第 3 趟只需对前 n-2 个记录进行排序。如此下去，直至没有记录需要交换为止，这样最多进行 n-1 趟排序。

用一个变量 f 记录最后一次交换的元素下标，在一趟排序结束后，f 指出表的后部从 f+1～n-1 的元素已排好序以后冒泡只需从 0～f 中即可。

冒泡排序的优化：

冒泡排序程序如下：

```
void Sort::Bsort(int r[], int n)
{
    count=0;
    int   j, k, f=n-1;
    while(f>=0){
       k=f-1;   f=-1;
       for(j=0;j<=k;j++){
          count++;
          if(r[j]>r[j+1])
          {
              swap(&r[j],&r[j+1]);
              f=j;
```

```
        }
      }
    }
}
```

在上述算法中，k 表示每遍扫描时需要进行比较的项目数；当 f>=0 时，表示在扫描过程中曾经发生过交换，还需进行扫描比较。F 的数值表示上一遍扫描过程中最后一次发生交换的位置。由这个算法不难分析，在最坏情况下，冒泡排序需要进行 n-1 遍扫描，共需要 n（n-1）/2 次比较和元素的交换。但这个工作量并不是必须的，一般情况下要小于这个工作量。也就是说 f 的用处就是进一步缩小比较的次数。因此本算法中需要用到 while 的循环，而不能只用 for 循环。

F、快速排序

1）基本思想：快速排序是对冒泡排序的一种改进。任取线性表的某个元素作为基准，将整个对象序列划分为左右两个子序列：

--- 左侧子序列中所有元素都小于或等于基准

--- 右侧子序列中所有元素都大于基准

--- 基准元素则排在这两个子序列中间(这也是该元素最终应安放的位置)。

然后分别对这两个子序列重复施行上述方法，直到所有的对象都排在相应位置上为止

2）分割过程：

① 选取表中一个元素 r[k]（一般选取第一个元素），令 x=r[k]，称为控制关键字，用控制关键字和无序区中其余元素关键字进行比较。

② 设置两个指示器 i,j，分别表示线性表第一个和最后一个元素位置。

③ 将 j 逐渐减小，逐次比较 r[j]与 x，直到出现一个 r[j]<x，然后将 r[j]移到 r[i]位置，将 i 逐渐增大，逐次比较 r[i]与 x，直到出现一个 r[i]>x，然后将 r[i]移到 r[j] 位置。

如此反复进行，直到 i=j 为止，最后将 x 移到 r[j]位置，完成一趟排序。此时以 x 为界分割成两个子区。

本程序亮点：在 Sort 类中设置了一个 count 的整型数据，主要是记录了各种排序算法中时间复杂度，即比较的次数，由此可以看到不同的算法在不同的情况下的效率。

排序运算运行结果：

选择排序：

堆排序：

```
Putin your array length:10
Putin your numbers:10 1 9 2 8 3 7 4 6 5
请选择排序方式:
 ---- 选择排序:1
 ---- 堆排序  :2
 ---- 插入排序:3
 ---- 希尔排序:4
 ---- 冒泡排序:5
 ---- 快速排序:6

2
排序时间长度：42
The result after sort is:1 2 3 4 5 6 7 8 9 10
结束请按0，继续请按1或其他数字!
```

插入排序：

```
Putin your array length:10
Putin your numbers:10 1 9 2 8 3 7 4 6 5
请选择排序方式:
 ---- 选择排序:1
 ---- 堆排序  :2
 ---- 插入排序:3
 ---- 希尔排序:4
 ---- 冒泡排序:5
 ---- 快速排序:6
3
排序时间长度：25
The result after sort is:1 2 3 4 5 6 7 8 9 10
结束请按0，继续请按1或其他数字!
```

希尔排序：

```
Putin your array length:10
Putin your numbers:10 1 9 2 8 3 7 4 6 5
请选择排序方式:
 ---- 选择排序:1
 ---- 堆排序  :2
 ---- 插入排序:3
 ---- 希尔排序:4
 ---- 冒泡排序:5
 ---- 快速排序:6
4
The result after sort is:1 2 3 4 5 6 7 8 9 10
```

冒泡排序：

```
Putin your array length:10
Putin your numbers:10 1 9 2 8 3 7 4 6 5
请选择排序方式:
 ---- 选择排序:1
 ---- 堆排序  :2
 ---- 插入排序:3
 ---- 希尔排序:4
 ---- 冒泡排序:5
 ---- 快速排序:6
5
The result after sort is:1 2 3 4 5 6 7 8 9 10
```

快速排序：

```
Putin your array length:10
Putin your numbers:10 1 9 2 8 3 7 4 6 5
请选择排序方式:
 ---- 选择排序:1
 ---- 堆排序  :2
 ---- 插入排序:3
 ---- 希尔排序:4
 ---- 冒泡排序:5
 ---- 快速排序:6
6
The result after sort is:1 2 3 4 5 6 7 8 9 10
```

7、Search 类----代码重用性的分析

Search 类主要完成 matlab 中的查找运算。

首先根据不同的数据长度等情况，写不同的查找算法非常有必要，这对查找的效率等很有好处，本程序需要使用者自己确定自己的查找方法。

关于查找的算法分析：

①线性查找

线性查找又称顺序查找，是一种最简单的查找方法。

基本思想:从第一个记录开始，逐个比较记录的关键字，直到和给定的 K 值相等，则查找成功；若比较结果与文件中 n 个记录的关键字都不等，则查找失败。

性能分析:等概率情况下，$P_i=1/n$，$C_i=i$，所以：

$$ASL=\sum_{i=1}^{n}P_iC_i=\frac{1}{n}\sum_{i=1}^{n}i=\frac{n+1}{2}$$

②对分查找

又称折半查找，是对有序表进行的一种查找。

基本思想:先找到“中间记录”，比较其关键字，如果关键字与给定值 K 相等，则查找成功；如果关键字 x 小于给定值 K，则说明被查找记录必在前半区间内；反之则在后半区间内。这样把搜索区间缩小了一半，继续进行查找。

性能分析

等概率下的平均查找长度为：

$$ASL=\sum_{i=0}^{n-1}P_i\cdot C_i=\frac{1}{n}\sum_{i=1}^{h}C_i=\frac{1}{n}(1*1+2*2^1+$$
$$+3*2^2+\cdots+(h-1)\times 2^{h-2}+h\times 2^{h-1})$$

可以用归纳法证明:

$$1\times 1+2\times 2^1+3\times 2^2+\cdots+(h-1)\times 2^{h-2}+h\times 2^{h-1}=$$
$$=(h-1)\times 2^h+1$$
$$ASL=\frac{1}{n}((h-1)\times 2^h+1)=\frac{1}{n}((n+1)\log_2(n+1)-n)$$
$$=\frac{n+1}{n}\log_2(n+1)-1\approx\log_2(n+1)-1$$

对分查找的优点是平均查找长度小，缺点是要求表中的元素按关键字排序。在 n 很小时(n<30) 对分查找不比顺序查找优越。

对分查找适用于一经建立就很少改动，而又经常需要查找的线性表。

代码重用性分析：

c＋＋中代码重用主要有一下三种方式：

①类的成员函数公用；

②委托：如成员对象；

③继承：

继承的优点之一是它支持渐增式开发，它允许我们在已开发的代码中引进新代码，而不会给源代码带来错误，即使产生了错误，这个错误也只是与新代码有关。也就是说当我们继承已存在的功能类并对其增加数据成员和成员函数（并重定义已存在的成员函数）时，已存在类的代码并不会被改变，更不会产生错误。

如果出现错误，我们就会知道它肯定在我们的新代码中。相对于修改已存在代码体的做法来说，这些新代码很短也很容易读。

认识到程序开发是一个渐增过程，就像人的学习过程一样，这是很重要的。我们做尽可能多的分析，但当开始一个项目时，我们仍不可能知道所有的答案。如果开始把项目作为一个有机的、可进化的生物来“培养”，而不是完全一次性的构造它，我们就会获得更大的成功和更直接的反馈。

有本类可以看到 search 类继承了 sort 类，并且 sort 中代码长度很长，而 search 类主要添加了搜索的功能，如果重新再编写排序（sort）的功能，代码量的开销很大，而运用继承，实现了代码的重用。由此例可以看到继承带来的好处。

四、组内成员分工及互评：

卢俊（姓名） 3 1 0 0 1 0 2 8 8 4（学号） 3（分数）

陈之（姓名） 3 1 0 0 1 0 1 3 2 9（学号） 2（分数）

组内分工：

卢俊：主要完成代码编写，实验报告编写，担任组长。

陈之：主要完成代码测试，算法指导。