猪脚说第一期

什么是猪脚说

为了改善同学们的上机体验，减轻同学们的压力，集中回答编程中常见问题，继承上一辈助教的优良传统，我们**为大家精心准备了猪脚说。**

猪脚说，就是~~猪脚~~助教们想对大家说的话。每次上机后，我们会及时总结大家**提问相对较多**或**比较重要**的问题，在猪脚说中以详细的篇幅加以阐述，希望同学们或多或少得到一些启发。

猪脚说包括但不限于**共性问题**, **coding 小技巧**, **课外习题。**

格式字符串

提问相关

1. **助教也是人**, 一般情况下没法瞬间回复, 更没办法瞬间完成解答;
2. 很多问题是大部分人都会遇到的, 所以希望同学们养成有问题**先搜索, 再提问**的习惯, 这样获得答案的效率更高;

提问前你要干什么?

1. 明确自己的问题到**代码行**;
2. 先到 **百度** 等搜索引擎搜寻自己的问题;
3. 若百度未能解决你的问题, 请到课程网站的**论坛**上搜寻相关问题;
4. 若课程网站的论坛未能解决你的问题, 请搜索课程微信群的**历史消息**, 看看有没有同学提到过相关问题;
5. 如果以上三步都没法解决你的问题, 那就提问吧!

如何提问更高效?

1. 尽可能**一语中的**地提问
   - [x] "执行**这句**代码时, Dev-C++ 报 SIGSEGV 错误, 截图如下/代码链接如下:" - **精确到行(或尽量精确)+错误信息+代码截图/链接**
   - [ ] "请问助教, 我这里哪错了?" - **请具体描述哪里不符合预期! 不精确到行的提问将相当耗费时间!**
2. 编写易读的代码:
   - **请不要用 `a, b, c, d, e, f` 这样没有意义的名字作为变量/函数的名字**, 建议用英文驼峰命名法, 英文单词不太懂的, 用拼音也问题不大;
   - 没记清楚运算符优先级时, 要么马上查一下, 要么**多用小括号**;
   - 就算 `if/while` 语句后面的执行语句只有一行, 也**请不要省去大括号**;
   - **勤加注释! 勤加注释! 勤加注释!**
   - 整理好你的代码, Dev-C++中使用 `Shift+Ctrl+A` 即可格式化代码, 更美观的代码会让答疑者更愿意帮你阅读代码;
3. 代码片段截图/代码片段链接, 告别微信群刷屏(**特别提出: 请不要抄袭盗用其他同学的代码, 我们有严格的代码查重!**):
   - 代码截图, 完全可以使用**微信自带的截图功能**;
   - 代码链接, 可以使用 **Ubuntu Paste**, 进入网站后使用方法如下:
     - Poster : 随意填写即可, 更推荐你填写学号姓名;
     - Syntax : 选择 `C` ;
     - Expiration : 过期时间, 选择 `A day` 意味着一天后系统将不再提供这段代码的链接;
     - Content : 粘贴**在Dev-C++中格式化后的**代码;
     - 点击 Paste , 随后生成一个网页, 把该网页的网址复制粘贴给别人, 别人就可以通过这个网址看到你的代码了.
4. 如果能够提供**复现错误**的方法, 就更好啦!

在哪里提问?

1. 课程论坛;
2. 微信群;
3. 私聊助教.

问题解决后?

十分鼓励同学们在解决自己的问题后, 把解决的思路写下来, 分享给大家. 可以是上传课程论坛, 可以是写成文档在微信群里分享!

程序调试方法

在周二晚上发布的上机问题汇总提到了debug很重要的一个思想，就是要**把程序按照业务逻辑分段**，然后**打印出每一阶段的结果**以确定程序在哪一步开始出了问题，定位出哪个阶段出问题后，可以**打印出程序运行过程中的某些变量**以确定具体的问题。

在本周的线上答疑过程中，我们发现很多同学没有理解这个想法，有些同学的理解也出了偏差。接下来我们会以具体代码来具体地讲解这个方法，这两份代码是同一个同学两次找我debug时发来的，我们可以从中了解到完整的debug流程，以及容易陷入的误区。

debug流程

这份代码是第一次作业的第二题的部分代码（大家不需要关注他具体是怎么实现的）：

```
#include<stdio.h>
int main()
 {
    // 初始化各种变量
    gets(str);  // 读入数据

    // 解析输入的数据，把数字存入num数组中，把操作符存入cal数组中
    // 比如输入20+3*4
    // 解析后的数组为：
    // num: [20, 0,  3, 0,  4]
    // cal: [\0, +, \0, *, \0]
    cnt=j; //cnt为运算符号的个数

    //先算乘除
    for(k=0; k<=j; k++) {
        if((cal[k]=='+'||cal[k]=='-'))
            continue;
        if(cal[k]=='*'||cal[k]=='/') {
            if(cal[k]=='*') num[k+1]=num[k]*num[k+1];//存结果在后面位置数字
            else num[k+1]=num[k]/num[k+1];
            num[k]=-1;  // 数字参与运算后标记为-1
            cnt--;
        }
    }

    //算加减
    // 此处省略若干代码

    printf("%d\n",ans);
```

```
    return 0;
}
```

这位同学的思路我在注释中大概写出来了，但是大家并不需要关注他具体是怎么实现的，只需要理解他的处理流程就可以。

我先是问了这位同学他自己觉得哪里可能出bug，他回复不知道哪里，可能是算加减的时候。

那么我是如何快速地找到出问题的地方的呢？接下来将会展示我给这份完全“陌生”的代码调试的全流程：

首先，我在明确了他的处理思路之后，可以把这份代码分为四部分，每部分的业务逻辑如下：

1. 读入数据；
2. 对输入的数据进行初步解析，构建`num`数组和`cal`数组；
3. 计算乘除法，更新`num`数组；
4. 计算加减法，结果量用`ans`变量去存储；

如果大家在调试自己的程序时觉得无从下手，往往可能是自己都没有想清楚自己的程序的业务流程。

紧接着，我会在`gets(str)`之后打印出`str`的内容：

```
#include<stdio.h>
int main()
 {
    // 初始化各种变量
    gets(str);  // 读入数据
    printf("%s", str);
    // ......
```

如果输出的内容和输入的内容一样，我们就能确定程序的第一步是没问题的。当然在本题中，读入的逻辑过于简单，似乎没有调试的必要，但是在其他的一些有复杂输入的题目中，**对自己的输入做检查**还是有必要的。

比如，我刚写完这部分内容，就有同学发来了如下代码：

```
int main()
{
   char a[100],b[100],c[100],A[100],B[100],C[100];
   scanf("%s",&a);
   scanf("%s",&b);
}
```

显然读入字符串时不应该加取址符（有的编译器确实能过，但是这样写是有问题的）。

现在第一步确定没有问题了，我会接下来检查他的程序第二步运行结果有没有问题，可以通过打印`num`和`cal`数组来实现：

```
// ......
// 解析输入的数据，把数字存入num数组中，把操作符存入cal数组中
    // 比如输入20+3*4
    // 解析后的数组为：
    // num: [20, 0,  3, 0,  4]
    // cal: [\0, +, \0, *, \0]
    cnt=j; //cnt为运算符号的个数

    for (int i = 0; i <= cnt; i++) {
        printf("%d %c ", num[i], cal[i]);
    }
// ......
```

这样，如果输出的内容和我们预期的相同，就能确定第二步是没有问题的。

同理，在对第三步的结果做检查时，可以把处理后的`num`数组打印出来：

```
// ......
//先算乘除
    for(k=0; k<=j; k++) {
        if((cal[k]=='+'||cal[k]=='-'))
            continue;
        if(cal[k]=='*'||cal[k]=='/') {
            if(cal[k]=='*') num[k+1]=num[k]*num[k+1];//存结果在后面位置数字
            else num[k+1]=num[k]/num[k+1];
            num[k]=-1;  // 数字参与运算后标记为-1
            cnt--;
        }
    }

    for (int i = 0; i <= j; i++) {
        if (num[i] != -1) printf("%d %c ", num[i], cal[i]);
    }
// ......
```

这里需要注意一下我们打印内容的不同——打印的内容应当能体现出这一阶段的处理结果。

这位同学这三步的结果都符合预期，那说明他的问题一定出在第四步。

那么具体该如何确定具体是哪里出了问题呢？我们在前面说过，这位同学的设计是用`ans`去存储计算的结果的——那我们可以在每次改变`ans`的值之后把`ans`打印出来，看看每次值的更新是否符合我们预期，以及与预期的差别在哪里，这样很快就能找到具体的问题所在了。

选择什么去打印

通过上面的例子，大家应当能体会到基本的debug步骤了。其实大家把自己的代码发给我们时，面对完全陌生的代码，我当然不可能从头至尾一行一行检查其代码逻辑——我一定是通过上面的这个流程来快速定位错点的。

那么我们该选取什么变量去打印，以及选择在程序的哪里去打印呢？上次教会了他这种调试方法后，他在调试第一次作业第四题时又给我发来了如下代码：

```
#include<stdio.h>
#include<string.h>
char shu1[100],shu2[100];
```

```
int num1[100]={0},num2[100]={0};
int len1,len2,mark,flag1,flag2;  // 各种变量


 int main()
 {
    int ans,len,i=0;
    read();
    // do something else
     return 0;
 }

int read()  //除去前导零，并倒着装进新数组
{
    int i,flag1,flag2;
    scanf("%s",&shu1);
    scanf("%s",&shu2);
    len1=strlen(shu1);
    len2=strlen(shu2);
    // 根据前导零的个数去更新len1和len2
//printf("len1:%d len2:%d\n",len1,len2);
    for(i=0;i<len1;i++){
        num1[i]=shu1[len1-i-1]-'0';//char类型转为int
//printf("i:%d %d ",i,num1[i]);
    }
    for(i=0;i<len2;i++){
        num2[i]=shu2[len2-i-1]-'0';
//printf("i:%d %d ",i,num2[i]);
    }
}
```

我在此放出了他的主函数和一个`read`函数的部分代码，大家还是不需要关注代码的具体细节，大家可以看到他的调试痕迹是非常明显的——似乎在每次有变量的值改变之后他都要把其输出出来，大家如果有兴趣的话可以试着给自己的程序加上这么多`printf`，看看输出的内容自己是否能分辨清楚。

之所以会出现这种情况，就是因为我上面强调的**对程序的业务逻辑分层**工作没有做好——看起来是在不停地`printf`，但其实他还是没有一个明确的调试方向。

在我看来，这个程序的`read()`函数所实现的应当是一个整体的业务逻辑，否则他也不会把他封装成一个函数，所以我发现了这一点后，我会直接在主函数中进行打印：

```
 int main()
 {
    int ans,len,i=0;
    read();

    printf("len1 = %d    len2 = %d\n", len1, len2);  // 打印两数组长度
     // 打印两数组内容
    for (int i = 0; i < len1; i++) {
        printf("%d", num1[i]);
    }
    printf("\n");
    for (int i = 0; i < len2; i++) {
        printf("%d", num2[i]);
    }
```

```
    // do something else
     return 0;
 }
```

他的`read`函数调用结束后会改变两数组的值，以及`len1,len2`两变量的值，我就直接打印出来，发现其实在这里,`len2`的值就已经不对了——接下来去`read`函数中寻找何时改变了`len2`的值就可以——这就非常简单了。

可见，**给程序的业务逻辑分层永远是我们调试程序（其实也应该是写程序）的第一步！！！**

如何打印出有效的内容

在周二的上机问题汇总里，我强调了大家的输出和标准输出一样时可能是打印了不可见字符，可以用下面这份代码去模拟：

```
int main()
{
    char s[1024] = {'\0'};
    scanf("%s", s);  // 读入字符串
    int len = strlen(s);  // 字符串长度
    for (int i = 0; i <= len; i++) {
        printf("%c", s[i]);  // 逐个字符打印
    }
    return 0;
}
```

当然，我这里只是为了复现这个问题，实际写代码的时候肯定不会这么逐个字符去打印字符串的。

大家会发现我们打印出来的结果和输入的看起来完全一样，但是评测就是过不去，怎么办？

可以这么去debug：既然字符本身不可见，那我就在他周围包裹上可见的字符，这样不就能显示出来了吗？比如这样：

```
int main()
{
    char s[1024] = {'\0'};
    scanf("%s", s);  // 读入字符串
    int len = strlen(s);  // 字符串长度
    for (int i = 0; i <= len; i++) {
        // 逐个字符打印，并用可见字符进行包裹，结尾再加上换行符
        printf("-------%c-------\n", s[i]);
    }
    return 0;
}
```

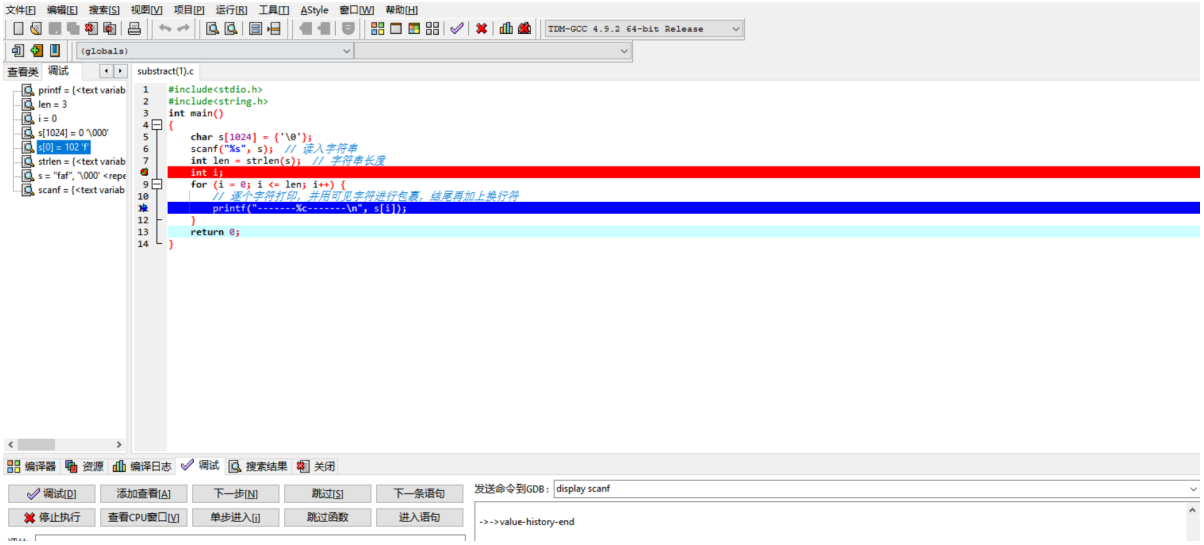
当然，针对具体的问题，我们会有不同的的打印策略，这个经验就需要大家在做题时候不断地总结经验，只要按着本章节所述的方法去调试，相信几周后大家的代码调试能力就会有巨大的进步。**如果大家平时debug时觉得自己的调试过程很出色，欢迎发到群里和大家分享！**

关于断点

这周在答疑过程中，很多同学对debug过程使用**断点**表现出了茫然感。这让助教同学也感觉有点懵，原来大伙还不知道断点是什么，正好借此机会给大伙详细讲讲断点是什么，以及怎么样在debug调试过程中使用断点。

断点调试介绍

断点调试是指在程序的某一行设置一个断点，调试时，**程序运行到这一行就会停住**，然后你可以一步一步往下调试，调试过程中可以看各个变量当前的值，出错的话，调试到出错的代码行即显示错误，停下。进行分析从而找到这个Bug。比如，下图是DEVC++的断点调试过程的截图。



大家可以看到，当我们点击程序左侧行数的时候，会出现一个红色的小圆圈，当我们在进行程序调试过程（而非编译或者运行过程）时，程序会停在设置断点的位置，通过下一步执行下一步语句，到达如图蓝色的位置，或者通过跳过直接执行到下一个断点的位置（或者执行不到就直接结束了）。在这个过程中，可以通过左边的监视来查看运行到这一步代码时的**各个变量的值**（当然这在DEV中需要打开这个功能）。大家可以参考这里来进行dev调试的使用。其他IDE的断点调试功能就更智能了，这里无需赘述，大家一看就应该会用。

断点debug的本质就是把上述的**printf的过程在代码执行的过程中可视化了**。

其他IDE介绍

为了高效开发，其实IDE工具的使用也是需要进行升级优化的。DEVC++对于初学者来说，优点是使用简单，基本上即下即用。但是对于开发的效率来说，由于不能提词补全等等的缺点，导致其开发使用效率并不是特别高。为了更好的完成作业以及进行未来可能的开发，这边推荐大家一些较为好用的IDE。

Visual Studio Code 是一个轻量级功能强大的源代码编辑器，支持语法高亮、代码自动补全（又称 IntelliSense）、代码重构、查看定义功能，并且内置了命令行工具和 Git 版本控制系统。适用于 Windows、macOS 和 Linux。它内置了对 JavaScript、TypeScript 和 Node.js 的支持，并为其他语言和运行时（如 C++、C#、Java、Python、PHP、Go、.NET）提供了丰富的扩展生态系统。

Clion是一款专门开发C以及C++所设计的跨平台的IDE。它是以IntelliJ为基础设计的，包含了许多智能功能来提高开发人员的生产力。这种强大的IDE帮助开发人员在Linux、OS X和Windows上来开发C/C++,同时它还能使用智能编辑器来提高代码质量、自动代码重构并且深度整合Cmake编译系统，从而提高开发人员的工作效率。

前者的优势在于轻量化，但是配置过程可能较为痛苦。后者的优势在于是专用的C/C++IDE功能十分强大，但是可能内存占用比较多，大家可以根据需求选择，百度相关内容。

23级程序设计基础练习题解

1. 判断可逆素数

思路分析

本题在实现思路上没有任何难点，我们只需要实现这两个功能：

1. 给定一个整数，判断其是否为素数；
2. 给定一个整数，返回其颠倒之后的数。

具体实现过程

根据上面的思路分析，很容易确定程序主题框架：

```
#include <stdio.h>
#include <math.h>

int isPrime(int num);  // 判断是否为素数
int reverseInt(int num);  // 翻转整数

int main()
{
    int n;
    scanf("%d", &n);
    n = n > 0 ? n : -n;  // 正数化

    if (isPrime(n) && isPrime(reverseInt(n))) printf("yes");
    else printf("no");

    return 0;
}
```

其中`isPrime`函数用于判断输入的整数是否为素数，是则返回1，否则返回0；`reverseInt`用于将输入的整数颠倒顺序后输出。

值得注意的是，为了方便两函数的实现，我们在主函数中将输入的整数正数化——在判断素数的问题上正负号无关紧要。（一些简单的`if-else`逻辑可以简化为三目运算符）

接下来实现`isPrime`函数。判断素数有很多种算法，这里就采用最朴素的想法（ds这门课的**常规作业**不考虑效率）：从2开始遍历到$sqrt(n)$，如果都没有整除n，则其就是素数（一个整数的因子不会大于其平方根）：

```
int isPrime(int num) {
    for (int i = 2; i < sqrt(num); i++) {
        if (num % i == 0) return 0;
    }
    return 1;  // 全都没有整除，返回1
}
```

对于`reverse`函数，只要清楚`n % 10`表示n的个位数，`n /= 10`相当于把n的个位数字去除掉，就问题不大。

```c
int reverseInt(int n) {
    int res = 0;  // 记录翻转的结果
    while (n) {
        int tmp = n % 10;  // tmp表示当前数的最后一位
        n /= 10;
        res = res * 10 + tmp;
    }
    return res;
}
```

参考代码

```c
#include <stdio.h>
#include <math.h>

int isPrime(int num);  // 判断是否为素数
int reverseInt(int num);  // 翻转整数

int main()
{
    int n;
    scanf("%d", &n);
    n = n > 0 ? n : -n;  // 正数化

    if (isPrime(n) && isPrime(reverseInt(n))) printf("yes");
    else printf("no");

    return 0;
}


int isPrime(int num) {
    for (int i = 2; i < sqrt(num); i++) {
        if (num % i == 0) return 0;
    }
    return 1;  // 全都没有整除，返回1
}


int reverseInt(int n) {
    int res = 0;  // 记录翻转的结果
    while (n) {
        int tmp = n % 10;  // tmp表示当前数的最后一位
        n /= 10;
        res = res * 10 + tmp;
    }
    return res;
}
```

2. 矩形相交

思路分析

本题的题面已经给我们提示了要去计算x轴和y轴上的交集。如果两轴上都有交集，显然计算乘积，否则直接输出0即可。

具体实现过程

首先在主函数中读入数据，并按照题面提示计算两轴上的交集：

```
// 主函数中
    int Ax1, Ay1, Ax2, Ay2, Bx1, By1, Bx2, By2;
    scanf("%d%d%d%d%d%d%d%d", &Ax1, &Ay1, &Ax2, &Ay2, &Bx1, &By1, &Bx2, &By2);

    int delta_x = min(max(Ax1, Ax2), max(Bx1, Bx2)) - max(min(Ax1, Ax2), min(Bx1,
Bx2));  // x轴上交集
    int delta_y = min(max(Ay1, Ay2), max(By1, By2)) - max(min(Ay1, Ay2), min(By1,
By2));  // y轴上交集
```

其中几个函数可以直接自己写出来（这种非常简单的函数，一行就写完了，不建议用`math.h`中声明好的函数，原因的话，大家可以想想自己是否清楚`asb`和`fabs`函数的区别）：

```
int min(int a, int b) { return a < b ? a : b; }
int max(int a, int b) { return a > b ? a : b; }
int abs(int n) { return n > 0 ? n : 0 - n; }
```

然后进行判断输出：

```
// 主函数中
    if (delta_x < 0 || delta_y < 0) printf("%d", 0);
    else printf("%d", delta_x * delta_y);
```

参考代码

```
#include <stdio.h>

int min(int a, int b) { return a < b ? a : b; }
int max(int a, int b) { return a > b ? a : b; }
int abs(int n) { return n > 0 ? n : 0 - n; }


int main()
{
    int Ax1, Ay1, Ax2, Ay2, Bx1, By1, Bx2, By2;
    scanf("%d%d%d%d%d%d%d%d", &Ax1, &Ay1, &Ax2, &Ay2, &Bx1, &By1, &Bx2, &By2);

    int delta_x = min(max(Ax1, Ax2), max(Bx1, Bx2)) - max(min(Ax1, Ax2), min(Bx1,
Bx2));  // x轴上交集
    int delta_y = min(max(Ay1, Ay2), max(By1, By2)) - max(min(Ay1, Ay2), min(By1,
By2));  // y轴上交集

    if (delta_x < 0 || delta_y < 0) printf("%d", 0);
```

```
    else printf("%d", delta_x * delta_y);

    return 0;
}
```

3. 求差集

思路分析

本题要求最后的输出顺序和A的输入顺序一致，就不用考虑先排序再双指针等想法了，直接暴力求解就可以，我们要做的有这些事情：

1. 读入A的数，并存入数组；
2. 逐个读取B，并在A中寻找是否有相同的数字，如果有，在A中进行标记以区分；
3. 按照顺序打印出A中未被标记的数。

具体实现过程

首先初始化集合，并把A读入集合：

```
// 主函数中
    int set[1005] = {0};
    int tmp;
    int n = 0;

    for (n = 0; ; n++) {
        scanf("%d", &tmp);
        if (tmp == -1) break;
        set[n] = tmp;
    }
```

然后读入集合B的值，并判断集合中是否已经有该值，如果有，我们将该位置标记为-1（因为题目说读入的都是自然数，不会有-1）：

```
// 主函数中
    while (1) {
        scanf("%d", &tmp);
        if (tmp == -1) break;
        for (int i = 0; i < n; i++) {
            if (set[i] == tmp) set[i] = -1;  // 如果集合中已经存在该数，标记为-1
        }
    }
```

接下来对`set`输出即可，注意要判断其值是否被标记过（== -1）：

```
// 主函数中
    for (int i = 0; i < n; i++) {
        if (set[i] != -1) printf("%d ", set[i]);
    }
```

参考代码

```c
#include <stdio.h>

int main()
{
    int set[1005] = {0};  // 存放结果的集合
    int tmp;
    int n = 0;

    for (n = 0; ; n++) {
        scanf("%d", &tmp);
        if (tmp == -1) break;
        set[n] = tmp;
    }

    while (1) {
        scanf("%d", &tmp);
        if (tmp == -1) break;
        for (int i = 0; i < n; i++) {
            if (set[i] == tmp) set[i] = -1;  // 如果集合中已经存在该数，标记为-1
        }
    }

    for (int i = 0; i < n; i++) {
        if (set[i] != -1) printf("%d ", set[i]);
    }

    return 0;
}
```

4. 矩阵运算

思路分析

本题在思路上并没有什么障碍，我们只需做如下步骤：

1. 定义一个二维数组，表示矩阵，并把第一个矩阵的值读到数组里；
2. 不断地读入操作符与矩阵，直到读到的操作符为#
3. 每次读矩阵的过程中，根据操作符的不同，更新数组里的值。

需要注意的是：本题中从始至终我们只需要定义一个二维数组，而不用每次读一个矩阵就定义一个，因为我们可以在读的过程中去更新数值（大家在下面的过程中体会）：

具体实现过程

首先是初始化数组，并读入第一个矩阵：

```
// 主函数中
    int n;
    char op;  // 操作符
    int matrix[10][10] = {{0}};  // 矩阵

    scanf("%d", &n);

    for (int i = 0; i < n; i++) {  // 读矩阵
        for (int j = 0; j < n; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }
```

然后我们要循环读入一个操作符，直到读入的操作符为#。

在循环体中，我们继续读取矩阵的每一个元素——在读入之后可以立即更新数组的值（如果有矩阵乘法，那这样就行不通了，那样我们只能把该矩阵整体读到另一个数组后再处理）：

```
// 主函数中
    while (scanf(" %c", &op) != EOF && op != '#') {  // 注意读字符前吞掉空格
        int tmp;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                scanf("%d", &tmp);
                // 直接根据操作符更新数组
                if (op == '+') matrix[i][j] += tmp;
                else matrix[i][j] -= tmp;
            }
        }
    }
```

最后对数组进行输出即可：

```
// 主函数中
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%5d", matrix[i][j]);
        }
        printf("\n");
    }
```

参考代码

```
#include <stdio.h>

int main()
{
    int n;
    char op;  // 操作符
    int matrix[10][10] = {{0}};  // 矩阵

    scanf("%d", &n);

    for (int i = 0; i < n; i++) {  // 读矩阵
```

```
        for (int j = 0; j < n; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }

    while (scanf(" %c", &op) != EOF && op != '#') {  // 注意读字符前吞掉空格
        int tmp;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                scanf("%d", &tmp);
                // 直接根据操作符更新数组
                if (op == '+') matrix[i][j] += tmp;
                else matrix[i][j] -= tmp;
            }
        }
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%5d", matrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

5. 文件拷贝

思路分析——双指针法

本题披上了文件操作的外套，如果我们去掉这层外套，可以把题面改写如下：给定一个字符串，将其中连续的多个空白字符替换为一个空格，并输出新的字符串。

这道题的想法很简单，却十分考验大家对代码的流程控制，以及算法功底，相信很多同学写着写着就发现代码写得“又长又烂”，逻辑十分混乱。

我这里介绍一种双指针的处理思路，它的时间复杂度是O(n)，比大家的暴力解法要快很多，逻辑也更清晰。

如果开头的字符不是空白字符（先考虑这种情况，到时候一行代码特判一下空白字符开头的情况就可以），我们可以把输入的字符串看成这样（“若干”的意思是大于等于0）：

第0个非空白字符 若干空白字符 第一个非空白字符 若干空白字符 。。。。。。

可以发现，**除了最开始（第0个）非空白字符外，其他的空白字符一定出现在非空白字符之前。**这里假设我们要处理的字符串变量为`buf`，我们的目的是去除掉`buf`中连续的空白字符。

我们可以定义两个整数`slow`和`fast`，代表遍历的字符数组的下标（也成为快慢“指针”——指向数组位置的针，不是真的指针变量），初始化时都为0.

接下来让`fast`指针遍历`buf`，当该位置上的字符是空白字符时，我们不去管它，只有当遇到了非空白字符时，我们才进行处理——给`slow`所指向的位置添加一个空格，然后`slow`向后移动一位，接下来`slow`和`fast`一起向前移动，并给`slow`所指向的位置赋值，直到遇到了下一个空白字符。最后重置一下`buf`的长度即可。

同时，由于字符串开头还有可能是空白字符，所以需要特判一下。

大家看这份文字版的题解可能觉得难以理解，这里我先给出完整代码：

参考代码

```
#include <stdio.h>
#include <string.h>

int main()
{
    char buf[1024] = {0};
    FILE *in = fopen("fcopy.in", "r");
    FILE *out = fopen("fcopy.out", "w");

    while (fgets(buf, sizeof(buf), in) != NULL) {  // 一行一行读文件
        int fast = 0, slow = 0;  // 用双指针法去除空格
        int len = strlen(buf);
        for (fast = 0; fast < len; fast++) {  // 快指针遍历数组
            if (buf[fast] != ' ' && buf[fast] != '\t') {  // 遇到非空白字符再处理
                if (!(slow == 0 && fast == 0)) buf[slow++] = ' ';  // 特判文件开头
就是空格的情况
                while (fast < len && buf[fast] != ' ' && buf[fast] != '\t') {  //
注意此处的约束条件
                    buf[slow++] = buf[fast++];
                }
            }
            if (fast == len - 1 && buf[fast] == ' ' && buf[fast] != '\t')
buf[slow++] = ' ';  // 特判文件尾是空格
        }
        buf[slow] = '\0';  // 重置字符串长度
        fprintf(out, "%s", buf);
    }

    return 0;
}
```

其实这个想法最难以理解的地方是**只有当我们遇到非空白字符时才处理**，也就是每次都是读到了连续空白字符的最后一个才处理——可能大家常规的想法是读到连续空白字符的第一个时就处理。

其次，这个算法的所有操作都是在`buf`这一个字符串上操作的，我们并没有额外开辟新的存储空间。

有的同学可能有疑问：这不是两层循环吗？但其实数组的每个元素只会被遍历到1次或2次，所以时间复杂阿德是O(n)。

无论如何，文字版的讲解总会显得不那么直观，如果大家有问题，不妨自己模拟一下操作流程，或者和同学或助教一起讨论一下。同时，建议大家理解思路之后自己再手动敲一遍这道题——这是很考验大家流程控制的一个算法。

双指针举例

ds课程中的常规作业不对代码效率做要求，所以大家如果觉得双指针太难理解，也可以适当缓一缓。

双指针是在线性表（数组，还有后面学到的链表等）上非常实用的一种处理思路，大家上网上搜算法题时也能搜到很多双指针的妙用，这里我举一个非常简单也很经典的例子：

给一单链表（链表可以理解成数组，但是其中的每一个元素只能通过它的前一个元素去找到，比如要找到下标为3的结点，只能从下标为0的头结点开始，先找到下标为1的结点，再找到下标为2的结点，再找到3的结点），在不求出链表长度的情况下（显式或隐式都不可以），返回其倒数第n个结点，如没有，返回NULL。

正常的想法肯定是从头开始遍历到尾，记录出链表长度，然后再从头开始遍历相应的次数就可以，但是题目规定了不可以求出长度。

双指针解法如下：

1. 定义快慢指针`fast = 头结点; slow = 头结点`；
2. 先让`fast`向后移动n次，如果在这过程中超过了链表的最后一个结点（最后一个结点的下一个是NULL），就返回NULL；
3. 然后两指针一起向后移动，直到`fast`到达链表末尾，此时`slow`所指向的结点就是倒数第n个。

这个例子大家可以在学完链表后再来看一看，如果大家对双指针有兴趣，也可以自行去网上搜索相应题目。

本周问题汇总

char与ASCII码

`char`类型的本质其实就是"可表示范围小一点"的整数，在格式化字符串中可以用`%c`来进行占位。

那么`'a'`是什么呢？我们知道`a`的ASCII码是97，可以写如下的调试程序：

```
#include <stdio.h>

int main()
{
    printf("%d", 'a' == 97 ? 1 : 0);
    return 0;
}
```

可以发现输出结果就是1——所以大家在判断字符相等的时候直接用单引号括起字符就可以，不要在代码中出现97，41这些神仙数——这会大大降低代码可读性。

同时，ASCII码编码时大写字母和小写字母分别都是连续的，所以想判断一个字符是不是小写字母时可以这么写：`if (ch >= 'a' && ch <= 'z')`；如果有一个表示数字的字符（注意理解一下这句话），比如`char ch = '2'`，如果想得到他所代表的数值（而不是ASCII码），可以这么写：`char ch = '2'; int i = ch - '0'`；大家也应该清楚地知道`'1' + 2`就是`'3'`（而不是数字3），`'a' + 1`就是`'b'`。

同时要注意，**C语言中没有`'26'`这种字面量**，整数和字符是不同的！

二维数组初始化

我们都知道如果想创建一个int型一维数组，并把所有值都初始化为0的话，可以这么写：`int array[1024] = {0};`(但是要注意这**种初始化方式一定要指定数组的大小！**)，而对于二维数组，需要这么写：`int array[10][10] = {{0}};`

C语言的宏

其实在本门课程中完全没有使用宏的必要，C语言中宏也有常量宏，宏函数，控制宏等，每一种都有其自己的功能，大家如有兴趣可以自行去了解，这里要强调的一点是大家所熟知的常量宏的本质就是**字符替换**，比如下面的程序：

```
#define NUM 10
int main()
{
    printf("%d", NUM);
    return 0;
}
```

结果会输出10，但是其原理并不是程序中有一个变量`NUM`取值为10，而是编译器在预处理阶段会把所有的常量宏做简单的字符替换，变成这样：

```
int main()
{
    printf("%d", 10);
    return 0;
}
```

有程序员开玩笑写了下面的代码，相信大家也能理解了：

```
#define itn int
#define mian main
#define retrun return
```

字符读入问题

如果有一串输入序列，其格式为“字符 [空格] 整数 字符 [空格] 整数 。。。。。。”

读入的时候如果这么写：`scanf("%c%d", &ch, &n);`就会发现从第二个字符开始读入的数据就是错误的了——因为整数后面跟着空格，我们相当于读入了那个空格，可以在`%c`前面加空格来使得函数跳过空白字符，读取第一个非空白字符：`scanf(" %c%d", &ch, &n);`

正好借此机会给大家详解一下有关scanf的相关知识：

scanf函数的一般形式

scanf函数是一个**标准库函数**，它的函数原型在头文件“**stdio.h**”中。

scanf函数的一般形式为：**scanf(“格式控制字符串”, 地址表列);**

注：地址列表项有些书说的是输入参数列表，说输入参数列表其实不够精确。

实现的功能即：将键盘输入的数据，按照规定的格式，存储到变量所在内存地址内。

其中，格式控制字符串的作用与printf函数相同，但不能显示非格式字符串，也就是不能显示提示字符串。地址表列中给出各变量的地址。地址是由地址运算符“&”后跟变量名组成的。

例如：&a、&b分别表示变量a和变量b的地址。

这个地址就是编译系统在内存中给a、b变量分配的地址。在C语言中，使用了地址这个概念，这是与其它语言不同的。 应该把变量的值和变量的地址这两个不同的概念区别开来。变量的地址是C编译系统分配的，用户不必关心具体的地址是多少。

变量的地址和变量值的关系

在赋值表达式中给变量赋值，如：

a=567;

则，a为变量名，567是变量的值，&a是变量a的地址。

但在赋值号左边是变量名，不能写地址，而scanf函数在本质上也是给变量赋值，但要求写变量的地址，如&a。这两者在形式上是不同的。&是一个取地址运算符，&a是一个表达式，其功能是求变量的地址。

在这个部分中，对于字符串来说，字符串本质是一个字符数组，其符号代表字符数组的**首地址**，因此在scanf输入的过程中不需要添加&取址，而是直接写符号就可以了。例如下面的例子：

```
// 错误例子
char a[100],b[100];
scanf("%s",&a);
scanf("%s",&b);

// 正确的写法
char a[100],b[100];
scanf("%s",a);
scanf("%s",b);
```

大家一定要注意区分，对地址的理解直接影响后面指针的使用。

【例1】

```
#include <stdio.h>
int main(void){
    int a,b,c;
    printf("input a,b,c\n");
    scanf("%d%d%d",&a,&b,&c);
    printf("a=%d,b=%d,c=%d",a,b,c);
    return 0;
}
```

在本例中，由于scanf函数本身不能显示提示串，故先用printf语句在屏幕上输出提示，请用户输入a、b、c的值。执行scanf语句，等待用户输入。在scanf语句的格式串中由于没有非格式字符在“%d%d%d”之间作输入时的间隔，因此在输入时要用一个以上的空格或回车键作为每两个输入数之间的间隔。如：

```
7 8 9
或
7
8
9
```

```c
#include <stdio.h>
int main(void){
    int a,b,c;
    //printf("input a,b,c\n");
    scanf("input a,b,c%d%d%d",&a,&b,&c);  //在格式控制字符串中出现普通字符时，
                                          //在输入时必须原样输入，否则会发生错误
    printf("a=%d,b=%d,c=%d",a,b,c);
    return 0;
}
```

输入：input a,b,c 7 8 9

输出：a=7,b=8,c=9　　（只有原样输出字符才能输出正确的结果，故在C语言中的scanf中输入普通字符时自找麻烦，但有些语言如Python时可以在输入的时候给出字符信息的。）

输入：7 8 9
输出：a=0,b=1,c=0　　（直接输入7 8 9 出现了错误）

格式字符串

格式字符串的一般形式为：**%[*] [输入数据宽度] [长度] 类型**
其中有方括号[]的项为任选项。各项的意义如下。

1. 类型
   表示输入数据的类型，其格式符和意义如下表所示。

```
格式   字符意义
d    输入十进制整数（decimal）
o    输入八进制整数（octal）
x    输入十六进制整数(hexadecimal)
u    输入无符号十进制整数(unsigned)
f或e 输入实型数(用小数形式或指数形式) (float  exponent)
c    输入单个字符 (character)
s    输入字符串(string)
```

2. “*”符
   用以表示该输入项，读入后不赋予相应的变量，即跳过该输入值。如：scanf("%d %*d %d",&a,&b);
   当输入为：1　2　3时，把1赋予a，2被跳过，3赋予b。

3. 宽度
   用十进制整数指定输入的宽度（即字符数）。

   例如（数据截取）：scanf("%5d",&a);输入12345678只把12345赋予变量a，其余部分被截去。

   又如（数据切分）：scanf("%4d%4d",&a,&b);输入12345678将把1234赋予a，而把5678赋予b。

   区分：输出printf（）右端对齐 printf("%4d ",a);输入3 输出结果为：　3（前面有3个空格，总共宽度为4）

4. 长度
   长度格式符为 l 和 h，l表示输入长整型数据（如%ld）和双精度浮点数（如%lf）。h表示输入短整型数据。

使用scanf函数还必须注意以下几点：

scanf函数中没有精度控制，如：scanf("%5.2f",&a);是非法的。不能企图用此语句输入小数为2位的实数。

scanf中要求给出变量地址，如给出变量名则会出错。如 scanf("%d",a);是非法的，应改为scnaf("%d",&a);才是合法的。

在输入多个数值数据时，若格式控制串中没有非格式字符作输入数据之间的间隔则可用空格，TAB或回车作间隔。C编译在碰到空格，TAB，回车或非法数据(如对“%d”输入“12A”时，A即为非法数据)时即认为该数据结束。

在输入字符数据时，若格式控制串中无非格式字符，则认为所有输入的字符均为有效字符。
例如： scanf("%c%c%c",&a,&b,&c);
输入 d、e、f 则把'd'赋予a，'、 ' 赋予b，'e'赋予c。只有当输入为 def 时，才能把'd'赋于a，'e'赋予b，'f'赋予c。

如果在格式控制中加入空格作为间隔，如：

scanf("%c %c %c",&a,&b,&c); //%c一个空格%c，在输入时可以有任意多个空格
则输入时各数据之间可加任意个空格。

【例2】

```
#include <stdio.h>
int main(void){
    char a,b;
    printf("input character a,b\n");
    scanf("%c%c",&a,&b);
    printf("%c%c\n",a,b);
    return 0;
}
```

由于scanf函数"%c%c"中没有空格，输入M N，结果输出只有M （实际上隐藏了一个空格）。而输入改为MN时则可输出MN两字符。

【例3】

```
#include <stdio.h>
int main(void){
    char a,b;
    printf("input character a,b\n");
    scanf("%c %c",&a,&b);
    printf("\n%c%c\n",a,b);
    return 0;
}
```

本例表示scanf格式控制串"%c %c"之间有空格时，输入的数据之间可以有任意个空格间隔。

5. 如果格式控制串中有非格式字符则输入时也要输入该非格式字符。
   例如： scanf("%d,%d,%d",&a,&b,&c);
   其中用非格式符“ , ”作间隔符，故输入时应为：5,6,7。

   又如：scanf("a=%d,b=%d,c=%d",&a,&b,&c);
   则输入应为：a=5,b=6,c=7。

6. 如输入的数据与输出的类型不一致时，虽然编译能够通过，但结果将不正确。

【例4】

```c
#include <stdio.h>
int main(void){
    int a;
    printf("input a number\n");
    scanf("%d",&a);
    printf("%ld",a);
    return 0;
}
```

由于输入数据类型为整型，而输出语句的格式串中说明为长整型，因此输出结果和输入数据不符。

（不会报错，用DEVC++测试结果，感觉不出什么问题....）

如改动程序如下（【例4-1】）：

```c
#include <stdio.h>
int main(void){
    long a;
    printf("input a long integer\n");
    scanf("%ld",&a);
    printf("%ld",a);
    return 0;
}
```

运行结果为:
input a long integer
1234567890
1234567890

当输入数据改为长整型后，输入输出数据相等。

【例5】

输入三个小写字母，输出其ASCII码和对应的大写字母。

```c
#include <stdio.h>
int main(void){
    char a,b,c;
    printf("input character a,b,c\n");
    scanf("%c %c %c",&a,&b,&c);
    printf("%d,%d,%d\n%c,%c,%c\n",a,b,c,a-32,b-32,c-32);
    return 0;
}
```

【例6】

输出各种数据类型的字节长度。

```c
#include <stdio.h>
int main(void){
    int a;
    long b;
    float f;
    double d;
    char c;
    printf("\nint:%d\nlong:%d\nfloat:%d\ndouble:%d\nchar:%d\n",
             sizeof(a),sizeof(b),sizeof(f),sizeof(d),sizeof(c));
    return 0;
}
```