程序设计分组训练 实验报告 1

专　　业：计算机科学与技术

班　　级：_____

学生姓名：_____

学　　号：_____

北京交通大学计算机与信息技术学院

2021 年 09 月 15 日

目录

1. 代码 A 的调试实验

1.1 变量监视

（1）断点调试监控

在 Dev c++中在第 28 行设置断点，进行调试，将代码 A 中所示变量进行监视并将数据初始值记录在表一中

表 1 数据初始值表（在第一个 for 循环执行前，监控到的各个变量值）

变量/常量名	监视值	变量/常量名	监视值	变量/常量名	监视值	变量/常量名	监视值
a	{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}}	p	(int (*)[4]) 0x65fdd0	*p	{1, 2, 3, 4}	*a	{1, 2, 3, 4}
a[0]	{1, 2, 3, 4}	p[0]	{1, 2, 3, 4}	*(p+0)	{1, 2, 3, 4}	*(a+0)	{1, 2, 3, 4}
a[1]	{5, 6, 7, 8}	p[1]	{5, 6, 7, 8}	*(p+1)	{5, 6, 7, 8}	*(a+1)	{5, 6, 7, 8}
a[3]	{13, 14, 15, 16}	p[3]	{13, 14, 15, 16}	*(p+3)	{13, 14, 15, 16}	*(a+3)	{13, 14, 15, 16}
&a[0][0]	(int *) 0x65fdd0	p[0]+0	(int *) 0x65fdd0	*p+0	(int *) 0x65fdd0	*a	{1, 2, 3, 4}
&a[0][1]	(int *) 0x65fdd4	p[0]+1	(int *) 0x65fdd4	*p+1	(int *) 0x65fdd4	*a+1	(int *) 0x65fdd4
&a[0][2]	(int *) 0x65fdd8	p[0]+2	(int *) 0x65fdd8	*p+2	(int *) 0x65fdd8	*a+2	(int *) 0x65fdd8
&a[0][3]	(int *) 0x65fddc	p[0]+3	(int *) 0x65fddc	*p+3	(int *) 0x65fddc	*a+3	(int *) 0x65fddc
&a[1][0]	(int *) 0x65fde0	p[1]+0	(int *) 0x65fde0	*p+4	(int *) 0x65fde0	*a+4	(int *) 0x65fde0
&a[1][1]	(int *) 0x65fde4	p[1]+1	(int *) 0x65fde4	*p+5	(int *) 0x65fde4	*a+5	(int *) 0x65fde4
&a[1][2]	(int *) 0x65fde8	p[1]+2	(int *) 0x65fde8	*p+6	(int *) 0x65fde8	*a+6	(int *) 0x65fde8
&a[1][3]	(int *) 0x65fdec	p[1]+3	(int *) 0x65fdec	*p+7	(int *) 0x65fdec	*a+7	(int *) 0x65fdec
&a[2][0]	(int *) 0x65fdf0	p[2]+0	(int *) 0x65fdf0	*p+8	(int *) 0x65fdf0	*a+8	(int *) 0x65fdf0
a[0][0]	1	*p[0]	1	**p	1	*a[0]	1

a[0][1]	2	*(p[0]+1)	2	*(*p+1)	2	*(a[0]+1)	2
a[0][3]	4	*(p[0]+3)	4	*(*p+3)	4	*(a[0]+3)	4
a[1][0]	5	p[1][0]	5	*(*p+4)	5	*(a[0]+4)	5

根请据表 1 监控到的数据，分析变量 p 的特点，类比表 1 的形式，将表 2 补充填写完毕；

表 2 数据初始值表（在第一个 for 循环执行前，监控到的各个变量值）

变量/常量名	监视值	变量/常量名	监视值	变量/常量名	监视值	变量/常量名	监视值
a	<不用填>	p	<不用填>	*p	<不用填>	*a	<不用填>
a[3][0]	13	p[3][0]	13	*(*(p+3)+0)	13	*(*(a+3)+0)	13
a[3][1]	14	p[3][1]	14	*(*(p+3)+1)	14	*(*(a+3)+1)	14
a[3][2]	15	p[3][2]	15	*(*(p+3)+2)	15	*(*(a+3)+2)	15
a[3][3]	16	p[3][3]	16	*(*(p+3)+3)	16	*(*(a+3)+3)	16

（2）在语句“int *q = a[0];”后插入以下代码

尝试在语句“int *q = a[0];”后插入以下代码，看程序是否能够正常编译，如果无法编译，报什么错误。

…

```
int b[4][4] = {{9,19,29,39},{8,18,28,38},{7,17,27,37},{6,16,26,36}};
a=b;
```

……

错误 1：error C2106: “=”：左操作数必须为左值

错误 2 ：IntelliSense: 表达式必须是可修改的左值

（3）在语句“int *q = a[0];”后插入以下代码

……

```
int
b[4][4]= {{9,19,29,39},{8,18,28,38},{7,17,27,37},{6,16,26,36}};
p=b;
```

…

程序能够正常编译

（4）a 与 p 的比较

根据表 1 和上述调试结果，分析 a 与 p 的相同点和不同点如下。

相同点：由表 1 中所记录的数据可得 a 与 p 所处内存地址相同，都可以看成是指针

不同点：p 是数组指针，且其指向的整型数组大小为 4，而则指向一个大小为 16 个整型元素的数组。p 是一个指针变量，因此进行赋值操作

a 是一个数组名，它所代表的是已分配内存数组的首地址，不能够使用等号进行赋值

（5）指针 p 与 q 的区别思考

表 3 指针变量初始值表（在第一个 for 循环执行前，监控到的各个变量值）

常量/变量名	监视值	常量/变量名	监视值	变量名	监视值
a	{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}}	p	(int (*)[4]) 0x65fdd0	q	(int *) 0x65fdd0
*a	{1, 2, 3, 4}	*p	{1, 2, 3, 4}	*q	1
a[0]	{1, 2, 3, 4}	p[0]	{1, 2, 3, 4}	q[0]	1
a[3]	{13, 14, 15, 16}	p[3]	{13, 14, 15, 16}	q[3]	4
a[4]	{12588256, 0, 1, 0}	p[4]	{12588256, 0, 1, 0}	q[4]	5
&a[0][0]	(int *) 0x65fdd0	*p	{1, 2, 3, 4}	q	(int *) 0x65fdd0
&a[0][1]	(int *) 0x65fdd4	*p+1	(int *) 0x65fdd4	q+1	(int *) 0x65fdd4
&a[0][3]	(int *) 0x65fddc	*p+3	(int *) 0x65fddc	q+3	(int *) 0x65fddc
a[0][0]	1	*p[0]	1	q[0]	1
a[0][1]	2	*p[1]	2	q[1]	2
a[0][3]	4	*p[3]	4	q[3]	4
a[2][0]	9	*(p+2)[0]	9	q[8]	9
a[2][1]	10	*(p+2)[1]	10	q[9]	10
a[2][3]	12	*(p+2)[3]	12	q[11]	11
&a[1][0]	(int *) 0x65fde0	*(p+1)	(int *) 0x65fde0	q+4	(int *) 0x65fde0
a[1][1]	6	*(*(p+1)+1)	6	*(q+5)	6
a[1][3]	8	*(*(p+1)+3)	8	*(q+7)	8
				*q+7	8
				*q+5	6
				*q+4	5

结论：p 是一个二级指针，其指向的数组有四个 int 型元素

*p 为 int 类型变量的地址，**p 则在*p 基础上在取一次值，取得 int 型数值。q 为一级指针，指向 int 型变量。

1.2 程序调试与修改错误

表 4 转置过程记录表

第几次交换	交换前矩阵值	i	j	p	q	p+j	*(p+j)	*(p+j)+i	*(*(p+j)+i)	q+j	*(q+j)	交换后矩阵值
1	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	0	1	0x65fdb0	0x65fdb0	0x65fdc0	0x65fdc0	0x65fdc0	5	0x65fdb4	2	1 5 3 4 2 6 7 8 9 10 11 12 13 14 15 16
2	1 5 3 4 2 6 7 8 9 10 11 12 13 14 15 16	0	2	0x65fdb0	0x65fdb0	0x65fdd0	0x65fdd0	0z65fdd0	9	0x65fdb8	3	1 5 9 4 2 6 7 8 3 10 11 12 13 14 15 16
3	1 5 9 4 2 6 7 8 3 10 11 12 13 14 15 16	0	3	0x65fdb0	0x65fdb0	0x65fde0	0x65fde0	0x65fde0	13	0x65fdbc	4	1 5 9 13 2 6 7 8 3 10 11 12 4 14 15 16

4	**1 5 9 13** **2 6 7 8** **3 10 11 12** **4 14 15 16**	1	2	0x65fdb0	0x65fdb0	0x65fdd0	0x65fdd0	0x65fd d4	4	0x65fdb8	9	1 5 10 13 2 6 7 8 3 9 11 12 4 14 15 16
5	1 5 10 13 2 6 7 8 3 9 11 12 4 14 15 16	1	3	0x65fdb0	0x65fdb0	0x65fde0	0x65fde0	0x65fd e4	5	0x65fdbc	13	1 5 10 14 2 6 7 8 3 9 11 12 4 13 15 16
6	1 5 10 14 2 6 7 8 3 9 11 12 4 13 15 16	2	3	0x65fdb0	0x65fdb0	0x65fde0	0x65fde0	0x65fd e8	6	0x65fdcc	8	1 5 10 14 2 6 7 15 3 9 11 12 4 13 8 16

错误分析：程序在第四次转置后发生错误转置，交换的并不是行数与列数相反的两个元素，从而导致无法正确进行元素交换。

修改后为 q=q+4

```
//进行数组转置操作
for (i = 0; i < 4; i++)     //遍历每一行
{

    for (j = i+1; j<4; j++)    //遍历需要交换的元素
    {
        //进行元素交换
        temp = *(*(p + j) + i);
        *(*(p + j) + i) = *(q + j);
        *(q + j) = temp;
    }

    //需要交换的位置指针移到下一行
    q = q + 4;
}
```

图 1-1

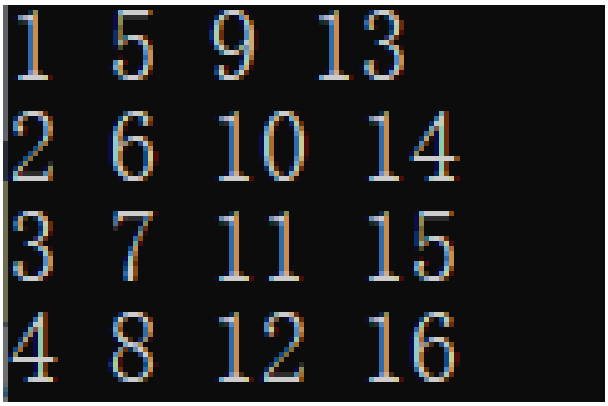

图 1-2

2. 代码 B 的调试实验

2.1 使用测试样例粗调

共四组样例，对第一个与第二个样例进行测试时，发现程序正常。
第三个和第四个样例进行测试时，程序输出一个正确一个错误两个鞍点。
经过简单代码逻辑分析，认为是函数 work 出现问题，开始进入代码分步调试。

2.2 代码分步调试

Work 函数中的 ColIndex 储存的数据对应矩阵中每一行的元素，一次循环后应该要初始化，以便下一次循环
应改为：
在

```
for (j = 1; j <= ColIndex[0]; j = j + 1)
{
if (check(i, ColIndex[j]) == 1)
remember(i, ColIndex[j]);
}
```

之后插入代码

```
……
for(j = 0; j < 100; j++)
ColIndex[j] = 0;
 ……
```

修改后三四样例正常输出如下

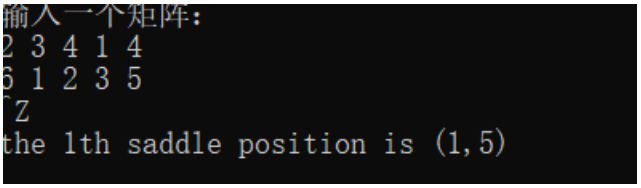

图 2-1

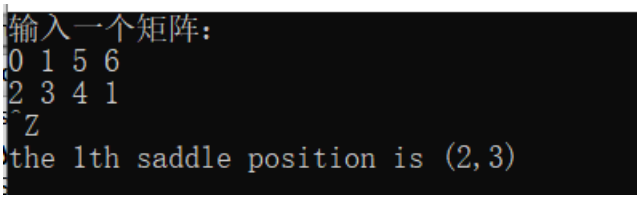

图 2-2

2.3 监控数组 ColIndex 的变化其中的非零元素

设置断点，监控数组 ColIndex 的变化，指出对于四个测试用例，数组 ColIndex 里的非零元素分别是什么，并在实验报告中加以说明？

断点设置在 work 函数中第一个 for 循环处

测试用例 1，ColIndex[0] == 2, ColIndex[1] == 1,ColIndex[2] == 2.
测试用例 2，ColIndex[0] == 2, ColIndex[1] == 2,ColIndex[2] == 2.
测试用例 3，ColIndex[0] == 3, ColIndex[1] == 2,ColIndex[2] == 4
测试用例 4，ColIndex[0] == 2, ColIndex[1] == 3,ColIndex[2] ==2

2.4 明确结构体数组 ans 记录的信息

通过程序跟踪，明确结构体数组 ans 记录了哪些信息，并在实验报告中加以说明？
数组 ans 记录了鞍点的个数以及每一个鞍点的坐标。
测试用例 1，ans[0].x == 1, ans[0].y ==0;
ans[1].x == 0, ans[1].y == 1;
测试用例 2，ans[0].x == 1, ans[0].y ==0;
ans[1].x == 0, ans[1].y == 2;

测试用例 3，ans[0].x == 1 ans[0].y ==0;
ans[1].x == 0, ans[1].y == 4;
测试用例 4，ans[0].x == 1, ans[0].y ==0;
ans[1].x == 1, ans[1].y == 2

2.5 check 函数

程序中 check 函数的作用是什么，请在实验报告中加以说明。

check 函数参数是一行中最大的数的位置，其作用为检验该数是否为这一列最小的数。是则返回 1，不是返回 0

2.6 程序局限

认真阅读 CodeForLabB 例程中的 prepare 函数，根据该函数的实现逻辑，指出 CodeForLabB 例程的局限性，什么情况下程序能正常运行，什么情况下不能？请在实验报告中加以说明。

prepare 函数使用 getchar 函数从控制台输入，则只有在矩阵元素都为 0 到 9 时才能成功

3. 实验体会与总结

3.1 调试的技巧

step-in 是逐语句调试，遇到子函数单步进入子函数，在子函数内部调试
step-over 是逐过程调试，将遇到的子函数其当成一条语句，一步就将函数执行完
step-out 是跳出，配合 step-in 使用，进入子函数后，在单步执行子函数的时候，step-out 可以执行完整个子函数，然后回到上层函数

3.2 非法内存访问有关问题

请说明为什么计算机有足够的内存空间时，却还要提醒我们的一些代码做了

一些非法内存访问操作。

非法内存访问，就是程序试图访问一块不受系统管理的内存区域。一般有两种情况，第一种是访问越界，是指应用程序申请了一定的内存，但是访问超出了申请的范围；第二种是无效访问，包括没有申请内存直接访问和程序逻辑漏洞导致访问无效的指针地址。需要指出的是，非法内存访问与计算机是否有足够的内存没有直接关系

请举例说明，程序被提醒做非法内存访问操作与现在世界哪些事情相类似。

例如，没有得到住建局的批准就建房，这是无效访问，是违法的。再如，已经向住建局申请了地皮，但是建房时超出了申请的面积，超出部分是违法的。

3.3 Memory Leak 有关问题

内存泄漏（Memory Leak）是指程序中已动态分配的内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。内存泄漏到一定程度会导致非法内存访问。

内存泄漏是指向系统申请分配内存进行使用(new)，可是使用完了以后却不归还(没有执行 delete)，那么那块内存程序之后也不能再访问，这就类似于内存丢失了，故名内存泄漏。

内存泄漏有四类：

常发性内存泄漏，发生内存泄漏的代码会被多次执行到，每次被执行时都会导致一块内存泄漏。

偶发性内存泄漏，发生内存泄漏的代码只有在某些特定环境或操作过程下才会发生。常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。

一次性内存泄漏，发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一块且仅有一块内存发生泄漏。

隐式内存泄漏，程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格的说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。

3.4 问题及解决和心得体会

本次实验中，遇到了大大小小很多问题，其中值得一提的收获就是 debug 的

时候，我之前总是尝试去从代码逻辑上发现错误，忽视了监视的作用。为了解答特定性问题，笔者积极设置变量，在完全没有明白 lab_b 中 ColIndex 函数实现方法的情况下，从监控值直接锁定了问题位置，尝试修改后，再次运行顺利得出鞍点位置。其实这也体现了一种工具理性的编程思想，不要重复造轮子，专注于自己的目的和函数功能，代码只是解决问题的手段。

3.5 小组互评

在没有编程基础的情况下，十分感谢大佬队友们的不嫌弃，在这次实验也多亏了他们的各种指导。在小组合作中，暴露出来的问题也很多，比如报告格式不统一、各成员进度不一的问题、部分监视值错误等。其中，word 的使用问题比较突出，在过程中也学习了很多关于标题段落固定、目录生成以及论文格式字体字号的知识，更增进了小组内感情，增加了合作的经验。