CPL 第四次编程练习 4-loops 题解

教学周历中本周课程的知识点为：

> More examples on loops; break/continue

教材章节：

> 8.2、8.3

共性问题

使用方向向量

在本周的题目中，有这样的处理逻辑——“统计周围8个格子中地雷的数量”，“观察上下左右四个格子是否有后续路径”，对于这种逻辑，我们可以统一使用方向向量来处理，在扫雷的备注中已经进行了提示。

以下代码段统计了周围四个格子中`?`的数量，对于 (i,j) 坐标的格子，周围四个格子分别是 $(i+1,j)$，$(i-1,j)$，$(i,j+1)$，$(i,j-1)$，可以视作 (i,j) 为基点，分别加上 $(1,0)$，$(-1,0)$，$(0,1)$，$(0,-1)$ 四个方向向量后得到的坐标，这也就是 `vectors` 变量的意义。**意思：向量**

```
int vectors[4][2] = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};

int count = 0;
for (int k = 0; k < 4; ++k) {
    int newI = i + vectors[k][0];
    int newJ = j + vectors[k][1];
    if (arr[newI][newJ] == '?') {
        count++;
    }
}
```

边界判断或者数组扩展

本周的题目中，有这样的处理逻辑——“统计周围8个格子中地雷的数量”，“观察上下左右四个格子是否有后续路径”，“统计周围三个格子的物种数量”，这种处理逻辑容易出现边界上部分原来需要判断的位置**不再合法**的情况，比如角落上周围只有3个元素而不是8个。

面对这种问题，有两种常见的处理方式，第一个是边界判断，第二个是数组扩展。

边界判断是指在使用了方向向量计算出$(newI,newJ)$后，对$(newI,newJ)$是否合法进行判断，如果不合法，那么就不做后续处理，题解中扫雷和路径追踪都使用了这种处理方式。

数组扩展是指用“无效值”把原来的数组包裹起来，这样即使超出了边界，也不需要进行额外的判断，题解中一线生机使用了这种处理方式。

两种方法各有优劣，但是我们更加推荐**边界判断**，因为：

1. **边界判断**保留了原有的数组下标使用习惯，也就是 $0,1,\cdots,n-1$，**数组扩展**会破坏这一习惯，比如一线生机的扩展就要用 $3,4,\cdots,len+2$。
2. **边界判断**复用性，扩展性更强。若使用**数组扩展**，周围8个元素的处理需要扩展一层，而周围24个元素就需要扩展两层；但使用**边界判断**始终是 `0 <= i < xlen && 0 <= j < ylen`

3. **边界判断**的正确性更加直接，它的含义就是“不合法的位置不处理”，而**数组扩展**则需要程序员寻找到一个“无效值”来填充外围数组，并在程序执行过程中保证外围数组始终只有该无效值，那么会有两个问题：
    1. 有时，“无效值”并不好找，你会发现无论在外围数组中填充什么值，都会对处理逻辑产生影响，比如，要求计算周围8个位置的数的阶乘的和，这时该填充什么“无效值”呢？非负整数都不行了，有的同学可能会说，我填一个负数然后判断可以吗？答：那为什么不直接进行边界判断呢？
    2. 可能因为没有保证外围数组只有无效值而出错，在一线生机中，有一些同学就是因为没有合理初始化，或者误操作修改了填充位置的值而导致出错

不要轻易复制粘贴

在学习了循环后，由于解决问题的过程中出现了大量重复或者类似的处理逻辑，有许多同学的做法是直接将代码进行大量复制粘贴，然后逐一修改，最终完成程序。

首先，这是一种本末倒置的处理手段——我们学习循环（还有后续的函数），目的就是为了用较为精简可读的代码表达重复类似的处理逻辑，将重复类似的处理逻辑**抽象**为循环或者函数而不是复制粘贴是一个合格的程序员的必修课。

其次，复制，粘贴，修改这个流程很容易出现“复制的代码是错的”这种死亡情境，一旦发生这种问题就导致程序员需要修复所有粘贴的代码段，修复不完全会导致更多的后续问题。

总的来说，对待复制粘贴一定要谨慎，能抽象成循环函数就不复制粘贴，在初学阶段，可能会有同学不会抽象而导致不得不复制粘贴，这是可以理解的，但是大家要先意识到这是个糟糕的做法，然后有意识地学习标程的抽象手段，最后减少自己复制粘贴的次数。

读入字符串时尽可能使用 scanf("%s")

当输入明显为**不包含空白符的字符串**（本次作业的 B、C、D 题）时，使用 `%s` 输入是最好的，这样 `scanf()` 会帮我们自动规避掉所有的空白符（空格、回车等）。

如果不是第二次作业那种奇特的要求，一般来说是不需要使用 `getchar()` 和循环使用 `scnaf("%c")` 来读入字符串的。同时也不推荐使用诸如 `gets()` 与 `fgets()`，他们的兼容性与鲁棒性都不如 `scanf(%s)`。万一测试用例出现了奇怪的不合法的字符，如 `\r`，（我们会保证数据合法，但以后呢？）那只有 `scanf(%s)` 可以胜任读取字符串的操作，其他四者都很容易出问题。

数独检验（sudoku.c）

> 本题知识点：
> 整数输入，字符串常量输出，二维整数数组初始化，赋值与访问，循环，一维标记数组初始化与访问

给出一个填好的数独，判断是否合法。

数独合法的充要条件是每一行、每一列、每一个九宫格中的 9 个数都是 $1, 2, 3, \cdots, 9$ 出现**恰好一次**。

其实这道题有很多种写法，一些简洁的写法可能利用了较多代码处理上的技巧而丧失了易读性，这样的做法为代码的扩展和后期维护埋下了隐患；而易读的写法则需要较长的代码实现。

我们提倡大家写出易读性更高程序。

参考答案1：

```c
#include <stdio.h>

int sudoku[9][9];
int exists[10];

int main(void) {
    for (int i = 0; i < 9; ++i)
        for (int j = 0; j < 9; ++j)
            scanf("%d", &sudoku[i][j]);
```

```
    int valid = 1;
    // 检查行
    for (int i = 0; i < 9; ++i) {
        // 用exists数组标记1-9这些数字是否出现过
        // 先初始化，所有的数字都没出现过
        for (int j = 1; j <= 9; ++j) {
            exists[j] = 0;
        }
        for (int j = 0; j < 9; ++j) {
            if (exists[sudoku[i][j]]) {
                // sudoku[i][j]是重复出现，就标记valid为0
                valid = 0;
            } else {
                // sudoku[i][j]是第一次出现，修改exists[sudoku[i][j]]
                exists[sudoku[i][j]] = 1;
            }
        }
    }

    // 检查列
    for (int i = 0; i < 9; ++i) {
        for (int j = 1; j <= 9; ++j) {
            exists[j] = 0;
        }
        for (int j = 0; j < 9; ++j) {
            if (exists[sudoku[j][i]]) {
                valid = 0;
            } else {
                exists[sudoku[j][i]] = 1;
            }
        }
    }

    // 检查九宫格
    // 注意数组下标的范围和移动方式
    // i，j用来移动九宫格左上角的元素坐标，每个九宫格使用k，l进行3*3的遍历
    for (int i = 0; i < 9; i += 3) {
        for (int j = 0; j < 9; j += 3) {
            for (int k = 1; k <= 9; ++k) {
                exists[k] = 0;
            }
            for (int k = 0; k < 3; ++k) {
                for (int l = 0; l < 3; ++l) {
                    if (exists[sudoku[i + k][j + l]]) {
                        valid = 0;
                    } else {
                        exists[sudoku[i + k][j + l]] = 1;
                    }
                }
            }
        }
    }

    printf("%s", valid ? "YES" : "NO");
    return 0;
}
```

接下来是一份十分简短但是几乎没有可读性的代码，这个程序是水龙哥哥写的，采用了大量位运算的技巧，大致意思就是将“$1,2,3,\cdots,9$ 这 9 个数是否出现”分别作为一个 01 比特压入一个 int 当中。大家没有必要深究这个程序到底是在干什么，可以体会一下，你作为一个阅读代码的人员，是愿意阅读上一份“冗长”的代码，还是这份“简短”的代码。

水龙哥哥的炫技：

```
#include <stdio.h>
int main() {
    long long ans[20] = {0}, x, i, j, flag = 0;
    for (i = 0; i ⩽ 8; i++)
        for (j = 0; j ⩽ 8; j++) {
            scanf("%d", &x);
            if (x > 9)
                flag = 1;
            else {
                ans[i / 3 * 3 + j / 3] ⊨ 1 << (x + 17);
                ans[i] ⊨ 1 << (x - 1);
                ans[j] ⊨ 1 << (x + 8);
            }
        }
    for (i = 0; i ⩽ 8; i++)
        if (ans[i] ≠ 0x7ffffff)
            flag = 1;
    printf("%s", flag ? "NO" : "YES");
    return 0;
}
```

你可能会注意到水龙哥哥代码的第10行，并好奇 ans[i / 3 * 3 + j / 3] 是什么含义。这个下标的意义其实不难理解，就是遍历九宫格中的每个数字！当你推理出这个下标的形式，那么 参考答案1 中那么多的循环（甚至四重循环）完全可以仿照水龙哥哥的思路，写得非常简洁！

参考答案2：

```
#include <stdio.h>

int sudoku[9][9];
int row[9][10], col[9][10], sub[9][10];  // 行列宫 3*9 个桶

int main(void) {
    for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++) {
            scanf("%d", &sudoku[i][j]);
            if (sudoku[i][j] > 9 || sudoku[i][j] < 1) {
                printf("NO");
                return 0;  // 提前结束程序
            }
            row[i][sudoku[i][j]]++;  // 行
            col[j][sudoku[i][j]]++;  // 列
            sub[i / 3 * 3 + j / 3][sudoku[i][j]]++;  // 宫
        }

    for (int i = 0; i < 9; i++) {
        for (int j = 1; j < 10; j++) {
            if (row[i][j] ≠ 1 || col[i][j] ≠ 1 || sub[i][j] ≠ 1) {
                printf("NO");
                return 0;
```

```
            }
        }
    }
    printf("YES");
    return 0;
}
```

为什么我判断所有数的和加起来都是 45 的程序没法通过?

大家可以思考一下，“$1, 2, 3, \cdots, 9$ 分别出现一次”与“累和是 45 ”这两个条件哪个更强。

一个题外话是，水龙哥哥为了卡掉这个错误的做法，研究了几个小时，最终也确实没让这样的程序通过。有兴趣的同学也可以来思考一下怎么构造出一个非法的数独，但它的每一行、每一列、每一宫中的和都是 45 。

但是有同学如果同时判断了“累和为45”与“累乘为362880”，那确实是可以通过的了。

最后，题目中其实漏写了：输入数据保证每一个数字均为 1~9。有的同学没判断数据合法性也可以通过。

扫雷（mines.c）

> 本题知识点：
> 整数与字符数组输入，二维整数数组输出，二维字符数组初始化与访问，二维整数数组初始化与赋值

用*和o给出一张 9×9 的扫雷的地图，将没有雷的格子标上数字（数字为与这个格子相邻的 8 个格子中有地雷的格子的数量）。

考虑使用二重循环枚举每个格子 (i, j)，然后枚举周围的八个格子，统计是否为地雷的格子，注意判断边界情况。

参考答案：

```
#include <stdio.h>

char mines[105][105];
int vectors[8][2] = {{-1, -1}, {-1, 0}, {-1, 1}, {1, -1},
                     {1, 0}, {1, 1}, {0, 1}, {0, -1}};

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; ++i) {
        scanf("%s", mines[i]);
    }

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (mines[i][j] == 'o') {
                int count = 0;
                for (int k = 0; k < 8; ++k) {
                    int newI = i + vectors[k][0];
                    int newJ = j + vectors[k][1];
                    // 注意 newI 和 newJ 必须是合法的坐标值
                    if (newI >= 0 && newI < n && newJ >= 0 &&
                        newJ < n && mines[newI][newJ] == '*')
                        count++;
                }
                printf("%d", count);
```

```
            } else
                printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

路径追踪（trace.c）

> 本题知识点：
> 整数与字符数组输入，二维整数数组输出，二维字符数组初始化与访问，二维整数数组（布尔数组）初始化与赋值，多重（不超过三重）循环

本题的思路是明确的，读入后使用循环检查当前点上下左右是否存在没有走过的#符号，如果有的话就走到该符号所在的位置，记录路径然后进行下一轮检查。

上下左右的检查可以参见**共性问题-使用方向向量**，另一个问题是如何标记某个#符号是否走过，按照提示，可以使用一个同样大小的数组进行标记。

```
#include <stdio.h>

char map[55][55];
int vis[55][55];
int trace[2505][2];
int vectors[4][2] = {{-1, 0},
                     {1, 0},
                     {0, 1},
                     {0, -1}};

int main() {
    int n, m, x0, y0;
    scanf("%d %d %d %d", &n, &m, &x0, &y0);
    for (int i = 0; i < n; ++i) {
        scanf("%s", map[i]);
    }
    x0--;
    y0--;

    vis[x0][y0] = 1;
    trace[0][0] = x0;
    trace[0][1] = y0;
    int x = x0, y = y0, notEnd = 0, traceLen = 1;
    do {
        notEnd = 0;  // 检查上下左右是否有符合条件的位置
        for (int i = 0; i < 4; ++i) {
            int newX = x + vectors[i][0];
            int newY = y + vectors[i][1];
            if (newX >= 0 && newX < n && newY >= 0 && newY < m && map[newX][newY] ==
'#' && !vis[newX][newY]) {
                // 标记该位置已经走过
                vis[newX][newY] = 1;  // 移动到该位置
                x = newX;
                y = newY;
                // 记录路径
```

```
                trace[traceLen][0] = newX;
                trace[traceLen][1] = newY;
                traceLen++;
                // 标记需要进入下一轮循环
                notEnd = 1;
            }
        }
    } while (notEnd);

    printf("%d\n", traceLen);
    for (int i = 0; i < traceLen; ++i) {
        printf("%d %d\n", trace[i][0] + 1, trace[i][1] + 1);
    }

    return 0;
}
```

一线生机(life.c)

> 本题知识点:
> 整数输入，一维字符数组输入，访问，赋值

本题思路比较明确，使用一个循环模拟n轮演化，然后使用两个数组交替存储的方式计算当前轮次的演化结果。

本题的一个难点在于如何根据现有位置的值计算出下一轮的该位置的值，由于情况看起来比较多，有些同学使用多次循环进行判断，这太复杂了，比较好的做法是先统计当前位置周围AB物种的个数，然后根据当前位置的值进行判断。

参考答案:

```c
#include <stdio.h>

char curr[1009];
char next[1009];
char b[1005];

int main() {
    int n, len = 0;
    scanf("%d", &n);
    scanf("%s", b);
    // 数组扩展，避免边界出错
    for (int i = 0; i < 1000; i++) {
        curr[i + 3] = b[i];
        next[i + 3] = curr[i + 3];
    }

    // 统计字符串长度
    for (len = 0; curr[len + 3] != '\0'; len++);
    // 或直接使用 len = strlen(b);

    for (int t = 1; t <= n; t++) {
        for (int i = 3; i < len + 3; i++) {
            // 统计周围三个位置上的AB物种数量
            int num_A = 0, num_B = 0;
            for (int k = 1; k <= 3; k++) {
                if (curr[i + k] == 'A') num_A++;
```

```
                if (curr[i - k] == 'A') num_A++;
                if (curr[i + k] == 'B') num_B++;
                if (curr[i - k] == 'B') num_B++;
            }
            // 计算下一轮该位置的值
            switch (curr[i]) {
                case '.':
                    if (num_A <= 4 && num_A >= 2 && num_B == 0) next[i] = 'A';
                    if (num_B <= 4 && num_B >= 2 && num_A == 0) next[i] = 'B';
                    break;
                case 'A':
                    if (num_B != 0 || num_A >= 5 || num_A <= 1) next[i] = '.';
                    break;
                case 'B':
                    if (num_A != 0 || num_B >= 5 || num_B <= 1) next[i] = '.';
                    break;
                default:
                    next[i] = curr[i];
                    break;
            }
        }
        for (int i = 3; i < len + 3; i++) {
            curr[i] = next[i];
        }
    }
    for (int i = 3; i < len + 3; i++) {
        printf("%c", next[i]);
    }
    return 0;
}
```

用了switch-case语句来分情况讨论

矩阵乘法(matrix-multi.c)

> 本题知识点:
> 整数输入，二维整数数组输入，访问，赋值

本题思路明确，直接使用两个双重循环读入，一个三重循环计算，一个双重循环输出(也可以直接边算边输出，下面标程是这么做的)

参考答案:

```
#include <stdio.h>

int A[100][100];
int B[100][100];
int C[100][100];

int main() {
    int m, n, p;
    scanf("%d %d %d", &m, &n, &p);

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

```
    }

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

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < p; ++j) {
            for (int k = 0; k < n; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```