数据结构实验报告（实验二：栈的应用）

- 题目：实验二 栈的应用
- 姓名：叶栩言
- 学号：2023200033
- 完成日期：2025-11-02

本次实验围绕“栈”的应用展开，分成两个部分：Part A 用算符优先法求解中缀表达式并动态展示两个栈的状态；Part B 通过递归与显式栈两套方案探索骑士巡逻问题。以下记录各部分的设计与调试过程。

Part A：表达式求值（算符优先）

1. 需求分析

需要编写一个小程序来演示算符优先算法的完整运行过程。程序读取一行以 `=` 结尾的中缀表达式，字符限定为 `0-9 + - * / ( ) =`，不出现空格。程序在处理每个记号时打印操作符栈、操作数栈的内容以及当前采取的动作，最后输出运算结果。如果遇到非法操作，例如除数为 0，就及时给出错误提示并终止。实现范围限定在二元整数运算，括号必须配对，除法采用整数除法。

2. 概要设计

核心数据结构是两个顺序栈：`OperatorStack` 使用数组 `optr[MAXSIZE]` 与栈顶指针 `top_optr` 保存运算符，`OperandStack` 用数组 `opnd[MAXSIZE]` 与 `top_opnd` 保存操作数。主程序流程如下：

1. 初始化操作符栈并压入哨兵 `=`。
2. 从左到右扫描表达式；读到数字时累加成多位数后入操作数栈。
3. 读到运算符时查表比较优先级，决定是入栈、执行一次运算还是完成括号匹配。
4. 当输入末尾的 `=` 与栈底哨兵配对后，循环结束，弹出操作数栈顶作为最终结果。

3. 详细设计

栈操作的伪码如下：

```
Push(stack[], top, element):
    if top = MAXSIZE - 1 then report overflow
    top ← top + 1
    stack[top] ← element

Pop(stack[], top):
    if top < 0 then report underflow
    element ← stack[top]
    top ← top - 1
    return element
```

主算法的伪码：

```
EvaluateExpression():
    read expr
    Push(optr, '=', top_optr)
    i ← 0
    while expr[i] ≠ '\0':
        ch ← expr[i]
        if ch is digit:
            num ← 0
            while expr[i] is digit:
                num ← num × 10 + (expr[i] − '0')
                i ← i + 1
            Push(opnd, num, top_opnd)
            ShowState(num, "数字入栈")
        else:
            top ← Peek(optr, top_optr)
            relation ← Compare(top, ch)
            if relation = '<':
                Push(optr, ch, top_optr)
                ShowState(ch, "符号入栈")
                i ← i + 1
            else if relation = '=':
                Pop(optr, top_optr)
                ShowState(ch, "括号匹配")
                i ← i + 1
            else:
                op ← Pop(optr, top_optr)
                b ← Pop(opnd, top_opnd)
                a ← Pop(opnd, top_opnd)
                res ← Apply(a, b, op)
                Push(opnd, res, top_opnd)
                ShowState(op, "计算")
    result ← Pop(opnd, top_opnd)
    output result
```

`compare` 函数通过一张 7×7 的优先级表返回 `<`、`>` 或 `=`；`apply` 负责四则运算并检查除零。

4. 调试分析

最初版本把结束符 `=` 与右括号当成同一种情况，导致程序在算完表达式后仍尝试继续比较，循环无法退出。补充优先级表中 `=` 的比较关系并在等号分支弹出哨兵后，问题解决。算法整体只扫描一次表达式，时间复杂度 O(m)，栈最大深度约为 O(m)。若继续扩展功能，可以考虑加入一元运算和浮点支持。调试时实时输出两个栈的内容，能快速核对课堂上的手工推导步骤。

5. 用户使用说明

1. 编译：`gcc expr_eval.c -o expr_eval`。
2. 运行 `./expr_eval`，按提示输入如 `3+5*2=`，即可看到过程信息与最终结果。

6. 测试结果

输入	期望输出	实际观察
3+5*2=	13	程序按步骤打印栈状态后输出 13。
(12+8)/5=	4	括号处理正确，最终得到 4。
3/0=	错误提示	检测到除数为 0 并立即终止。

7. 附录：源程序

Project/expr_eval.c

```c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define MAXSIZE 100

char optr[MAXSIZE];
int top_optr = -1;

int opnd[MAXSIZE];
int top_opnd = -1;

void push_optr(char c) { optr[++top_optr] = c; }
char pop_optr() { return optr[top_optr--]; }
char peek_optr() { return optr[top_optr]; }

void push_opnd(int x) { opnd[++top_opnd] = x; }
int pop_opnd() { return opnd[top_opnd--]; }

void show_state(const char* token, const char* action) {
    printf("当前输入: %s\n", token);
    printf("主要操作: %s\n", action);

    printf("操作符栈: ");
    if (top_optr == -1) printf("空");
    else for (int i = 0; i <= top_optr; i++) printf("%c ", optr[i]);
    printf("\n");

    printf("操作数栈: ");
    if (top_opnd == -1) printf("空");
    else for (int i = 0; i <= top_opnd; i++) printf("%d ", opnd[i]);
    printf("\n\n");
}

char compare(char op1, char op2) {
    char table[7][7] = {
      /*        +    -    *    /    (    )    =   */
      /* + */  {'>', '>', '<', '<', '<', '>', '>'},
      /* - */  {'>', '>', '<', '<', '<', '>', '>'},
      /* * */  {'>', '>', '>', '>', '<', '>', '>'},
      /* / */  {'>', '>', '>', '>', '<', '>', '>'},
```

```
        /* ( */  {'<', '<', '<', '<', '<', '=', ' '},
        /* ) */  {'>', '>', '>', '>', ' ', '>', '>'},
        /* = */  {'<', '<', '<', '<', '<', ' ', '='}
    };

    int idx1 = 0, idx2 = 0;

    switch (op1) {
        case '+': idx1 = 0; break;
        case '-': idx1 = 1; break;
        case '*': idx1 = 2; break;
        case '/': idx1 = 3; break;
        case '(': idx1 = 4; break;
        case ')': idx1 = 5; break;
        case '=': idx1 = 6; break;
    }

    switch (op2) {
        case '+': idx2 = 0; break;
        case '-': idx2 = 1; break;
        case '*': idx2 = 2; break;
        case '/': idx2 = 3; break;
        case '(': idx2 = 4; break;
        case ')': idx2 = 5; break;
        case '=': idx2 = 6; break;
    }

    return table[idx1][idx2];
}

int apply(int a, int b, char op) {
    switch (op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/':
            if (b == 0) {
                printf("错误: 除数为 0。\n");
                exit(1);
            }
            return a / b;
        default:
            printf("未知运算符: %c\n", op);
            exit(1);
    }
    return 0;
}

int main() {
    char expr[256];
    printf("请输入以=结尾的表达式: (中间不要有空格)  ");
    scanf("%s", expr);

    push_optr('=');
```

```
    int i = 0;
    char ch = expr[i];

    while (ch != '\0') {
        if (isdigit(ch)) {
            int num = 0;
            while (isdigit(expr[i])) {
                num = num * 10 + (expr[i] - '0');
                i++;
            }
            push_opnd(num);

            char buf[32];
            sprintf(buf, "%d", num);
            show_state(buf, "数字入栈");

            ch = expr[i];
        } else {
            char top = peek_optr();
            char t = compare(top, ch);

            if (t == '<') {
                push_optr(ch);
                char buf[2] = { ch, '\0' };
                show_state(buf, "符号入栈");
                ch = expr[++i];
            } else if (t == '=') {
                pop_optr();
                char buf[2] = { ch, '\0' };
                show_state(buf, "括号匹配");
                ch = expr[++i];
            } else if (t == '>') {
                char op = pop_optr();
                int b = pop_opnd();
                int a = pop_opnd();
                int res = apply(a, b, op);
                push_opnd(res);

                char buf[2] = { op, '\0' };
                show_state(buf, "计算");
            }
        }
    }

    printf("\n最终结果: %d\n", pop_opnd());
    return 0;
}
```

Part B：骑士巡逻（递归与栈模拟）

1. 需求分析

目标是在 1≤n≤8 的 n×n 棋盘上，从指定起点出发寻找骑士巡游路径，并分别用递归回溯和显式栈模拟回溯实现。交互方式为命令行菜单：先选模式，再输入棋盘大小和起点坐标（1 基）。若存在完整巡游路径，程序按访问顺序输出棋盘；若无解或输入越界，需要给出明确提示。此次实现不加入启发式，只使用标准的八种马走 L 形跳法。用于测试的场景包括：`n=5, 起点(1,1)`（应能找到路径）、`n=4, 起点(1,1)`（理论无解）、以及输入超范围的 `n=9`。

2. 概要设计

两个模式共享一套数据：`board_recursive` 与 `board_iterative` 保存棋盘步号；`Frame` 结构体包含 `x`、`y` 和 `next_move` 字段，用于非递归版的显式栈。主流程如下：

1. `main` 函数负责菜单循环和输入解析。
2. 递归模式 `run_recursive_knight` 校验参数，调用 `reset_board` 清空棋盘，然后递归调用 `dfs_knight`。
3. 栈模拟模式 `run_iterative_knight` 初始化帧栈和步骤计数，使用 while 循环模拟回溯过程。
4. 两种模式都会在成功时打印棋盘，在失败或输入异常时输出提示。

模块关系图：

```
main
├─ clear_line
├─ run_recursive_knight
│  ├─ reset_board
│  ├─ dfs_knight (递归自调用)
│  └─ print_board
└─ run_iterative_knight
   ├─ reset_board
   └─ print_board
```

3. 详细设计

递归搜索伪码：

```
DFS(n, x, y, step):
    board[x][y] ← step
    if step = n × n then return true
    for k in 0..7:
        nx ← x + dx[k]; ny ← y + dy[k]
        if InsideBoard(n, nx, ny) and board[nx][ny] = 0:
            if DFS(n, nx, ny, step + 1) then return true
    board[x][y] ← 0
    return false
```

栈模拟伪码：

```
IterativeKnight(n, sx, sy):
    reset board
```

```
    top ← 0; step ← 1
    stack[0] ← {sx, sy, 0}
    board[sx][sy] ← 1
    while top ≥ 0:
        if step = n × n:
            print board; return success
        if stack[top].next_move ≥ 8:
            board[stack[top].x][stack[top].y] ← 0
            top ← top − 1
            step ← step − 1
            continue
        move_id ← stack[top].next_move
        stack[top].next_move ← move_id + 1
        nx ← stack[top].x + dx[move_id]
        ny ← stack[top].y + dy[move_id]
        if InsideBoard(n, nx, ny) and board[nx][ny] = 0:
            top ← top + 1
            step ← step + 1
            stack[top] ← {nx, ny, 0}
            board[nx][ny] ← step
    report failure
```

4. 调试分析

非递归版在最初的实现里没有在回溯时清理棋盘，导致路径残留。把清零步骤放到弹栈分支后，问题得到解决。另外，菜单输入如果出现字母会留下换行符，后续读取整数会失败，因此我增加了 `clear_line` 把读到换行符为止的字符清掉。算法在最坏情况下需要遍历大量分支，但由于 $n \le 8$，运行时间仍能接受。若要优化，可以引入 Warnsdorff 规则或对候选步进行排序。

5. 用户使用说明

1. 进入 `Lab/Lab2_栈/Project`。
2. 编译：`gcc knight_tour.c -o knight_tour`。
3. 执行 `./knight_tour`，根据菜单选择模式：
    - 输入 `1` 运行递归回溯。
    - 输入 `2` 运行栈模拟。
    - 输入 `0` 退出程序。
4. 按提示输入棋盘尺寸和起点坐标，即可查看输出。

6. 测试结果

模式	输入序列	期望行为	观察结果
递归	`1 → 5 → 1 1 → 0`	找到完整路径	输出 5×5 步号表，与书本示例一致。
递归	`1 → 4 → 1 1 → 0`	无解	提示“没有找到路线。”
递归	`1 → 9 → 0`	输入非法	提示“棋盘尺寸超出范围。”
非递归	`2 → 5 → 1 1 → 0`	找到完整路径	打印路径，与递归版一致。

```
$ ./knight_tour
====== 骑士巡逻 ======
1. 递归回溯
2. 栈模拟非递归
0. 退出
请选择: … → 找到了一条路线:
 1  6 15 10 21
14  9 20  5 16
19  2  7 22 11
 8 13 24 17  4
25 18  3 12 23
```

```
$ ./knight_tour
====== 骑士巡逻 ======
1. 递归回溯
2. 栈模拟非递归
0. 退出
请选择: 请输入棋盘大小n(<=8): 请输入骑士起点的行列(从1开始): 没有找到路线。
```

7. 附录：源程序

Project/knight_tour.c

```c
#include <stdio.h>

#define MAX_BOARD 8
#define MOVE_COUNT 8

static int board_recursive[MAX_BOARD][MAX_BOARD];
static int board_iterative[MAX_BOARD][MAX_BOARD];
static int dx[MOVE_COUNT] = {2, 1, -1, -2, -2, -1, 1, 2};
static int dy[MOVE_COUNT] = {1, 2, 2, 1, -1, -2, -2, -1};

void reset_board(int board[MAX_BOARD][MAX_BOARD], int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            board[i][j] = 0;
        }
    }
}

void print_board(int board[MAX_BOARD][MAX_BOARD], int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("%2d ", board[i][j]);
        }
```

```
            printf("\n");
        }
    }

    int inside_board(int n, int x, int y) {
        return x >= 0 && x < n && y >= 0 && y < n;
    }

    int dfs_knight(int n, int x, int y, int step) {
        int k;
        board_recursive[x][y] = step;
        if (step == n * n) {
            return 1;
        }
        for (k = 0; k < MOVE_COUNT; k++) {
            int nx = x + dx[k];
            int ny = y + dy[k];
            if (inside_board(n, nx, ny) && board_recursive[nx][ny] == 0) {
                if (dfs_knight(n, nx, ny, step + 1)) {
                    return 1;
                }
            }
        }
        board_recursive[x][y] = 0;
        return 0;
    }

    void run_recursive_knight() {
        int n, sx, sy;
        printf("请输入棋盘大小n(<=8): ");
        if (scanf("%d", &n) != 1) {
            printf("输入错误。\n");
            return;
        }
        if (n <= 0 || n > MAX_BOARD) {
            printf("棋盘尺寸超出范围。\n");
            return;
        }
        printf("请输入骑士起点的行列(从1开始): ");
        if (scanf("%d %d", &sx, &sy) != 2) {
            printf("输入错误。\n");
            return;
        }
        if (sx < 1 || sx > n || sy < 1 || sy > n) {
            printf("起点超出棋盘。\n");
            return;
        }

        reset_board(board_recursive, n);
        if (dfs_knight(n, sx - 1, sy - 1, 1)) {
            printf("找到了一条路线:\n");
            print_board(board_recursive, n);
        } else {
            printf("没有找到路线。\n");
```

```
    }
}

typedef struct {
    int x;
    int y;
    int next_move;
} Frame;

void run_iterative_knight() {
    int n, sx, sy;
    Frame stack[MAX_BOARD * MAX_BOARD];
    int top;
    int step;

    printf("请输入棋盘大小n(<=8): ");
    if (scanf("%d", &n) != 1) {
        printf("输入错误。\n");
        return;
    }
    if (n <= 0 || n > MAX_BOARD) {
        printf("棋盘尺寸超出范围。\n");
        return;
    }
    printf("请输入骑士起点的行列(从1开始): ");
    if (scanf("%d %d", &sx, &sy) != 2) {
        printf("输入错误。\n");
        return;
    }
    if (sx < 1 || sx > n || sy < 1 || sy > n) {
        printf("起点超出棋盘。\n");
        return;
    }

    reset_board(board_iterative, n);
    top = 0;
    step = 1;
    stack[0].x = sx - 1;
    stack[0].y = sy - 1;
    stack[0].next_move = 0;
    board_iterative[sx - 1][sy - 1] = 1;

    while (top >= 0) {
        if (step == n * n) {
            printf("非递归方法找到了一条路线:\n");
            print_board(board_iterative, n);
            return;
        }

        if (stack[top].next_move >= MOVE_COUNT) {
            board_iterative[stack[top].x][stack[top].y] = 0;
            top--;
            step--;
            continue;
```

```
        }

        {
            int move_id = stack[top].next_move;
            int nx = stack[top].x + dx[move_id];
            int ny = stack[top].y + dy[move_id];
            stack[top].next_move++;
            if (inside_board(n, nx, ny) && board_iterative[nx][ny] == 0) {
                top++;
                step++;
                stack[top].x = nx;
                stack[top].y = ny;
                stack[top].next_move = 0;
                board_iterative[nx][ny] = step;
            }
        }
    }

    printf("非递归方法没有找到路线。\n");
}

void clear_line() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF) {
    }
}

int main() {
    int choice;
    while (1) {
        printf("====== 骑士巡逻 ======\n");
        printf("1. 递归回溯\n");
        printf("2. 栈模拟非递归\n");
        printf("0. 退出\n");
        printf("请选择: ");
        if (scanf("%d", &choice) != 1) {
            printf("请输入数字选项。\n");
            clear_line();
            continue;
        }
        clear_line();
        if (choice == 1) {
            run_recursive_knight();
        } else if (choice == 2) {
            run_iterative_knight();
        } else if (choice == 0) {
            printf("程序结束。\n");
            break;
        } else {
            printf("没有这个选项。\n");
        }
        printf("\n");
    }
```

```
    return 0;
}
```