C/C++从入门到精通-高级程序员之路

【 数据结构 】

栈及其企业级应用

第 1 节 栈的故事导入

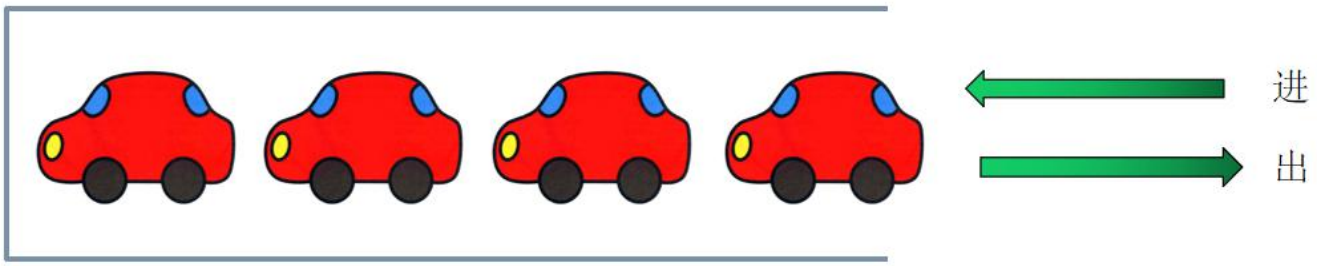
故事情节

程序员 Jack 拼搏了几年后，终于年薪过万买了车，但是，却碰到了麻烦事，他家住在胡同的尽头，胡同非常窄，只能容纳一辆车通过，而且是死胡同，每天 Jack 都为停车发愁，回家早了停在里面，早上上班就要让所有的人挪车，先让胡同口那辆出去，然后挨着一辆一辆出去，Jack 才能去上班,加上堵车，Jack 上班经常迟到。没办法，Jack 下班也不敢早回家了，经常下班后在公司加加班，等天黑了，别的车都停进去了再回家，再回去把车停在胡同口，这样早上就可以第一个去上班了。就这样，Jack 过起了"起早贪黑"的有车生活。

死胡同

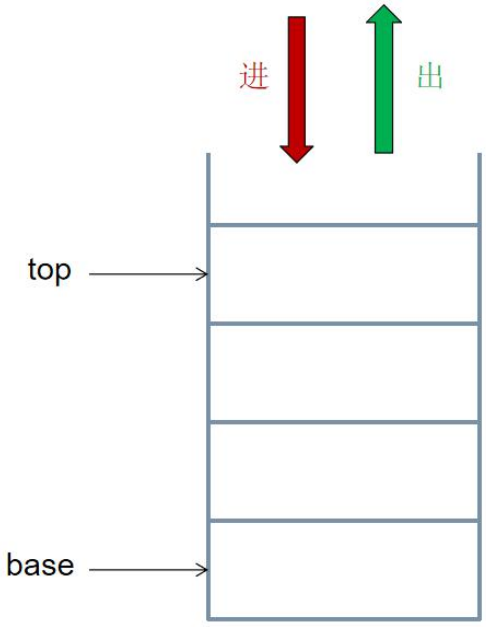
第 2 节 栈的原理精讲

Jack 家的胡同很窄，只能通过一辆车，而且是死胡同，只能从胡同口进出，画图：



胡同里的小汽车是排成一条直线，是线性排列，而且只能从一端进出，后进的汽车先出去，后进先出（Last In First Out，LIFO），这就是"栈"。栈也是一种线性表，只不过它是操作受限的线性表，只能在一端操作。

进出的一端称为栈顶（top），另一端称为栈底（base）。栈可以用顺序存储，也可以用链式存储。我们先看顺序存储方式：



其中，base 指向栈底，top 指向栈顶。

注意：栈只能在一端操作，后进先出，这是栈的关键特征，也就是说不允许在中间查找、取值、插入、删除等操作，我们掌握好顺序栈的初始化、入栈，出栈，取栈顶元素等操作即可。

第 3 节　顺序栈的算法实现

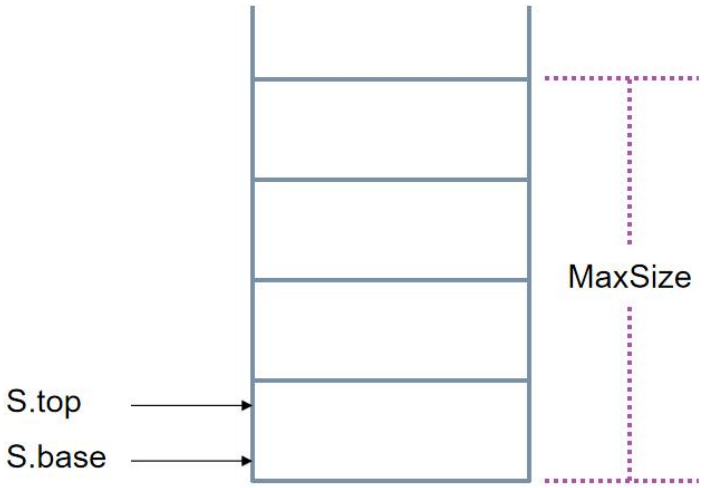
栈数据结构的定义

```
#define MaxSize  128  //预先分配空间，这个数值根据实际需要预估确定

typedef int ElemType;

typedef struct _SqStack{
    ElemType *base;    //栈底指针
    ElemType *top;     //栈顶指针
}SqStack;
```

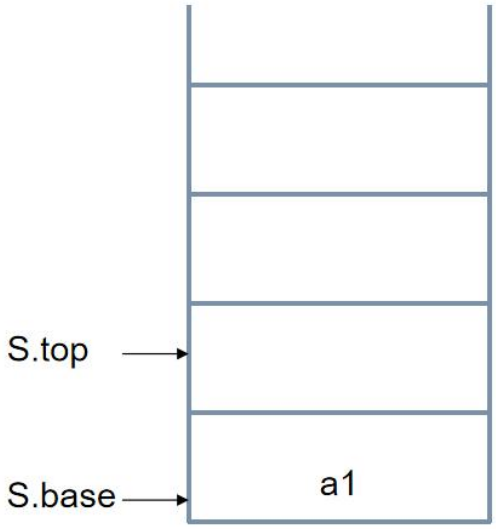
栈的初始化



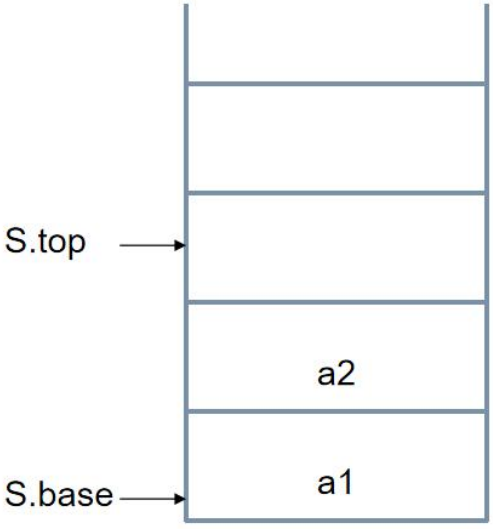

```
bool InitStack(SqStack &S) //构造一个空栈 S
{
    S.base = new int[MaxSize];//为顺序栈分配一个最大容量为 Maxsize 的空间
    if (!S.base) //空间分配失败
        return false;
    S.top=S.base; //top 初始为 base，空栈
    return true;
}
```

入栈

入栈操作：判断是否栈满，如果栈已满，则入栈失败，否则将元素放入栈顶，栈顶指针向上移动一个空间（top++）。



a. 存入第一个元素



b. 存入第二个元素

```
bool PushStack(SqStack &S, int e) // 插入元素 e 为新的栈顶元素

{
    if (S.top-S.base == MaxSize) //栈满
        return false;

    *(S.top++) = e; //元素 e 压入栈顶，然后栈顶指针加 1，等价于*S.top=e;
S.top++;

    return true;
}
```

出栈

出栈操作：和入栈相反，出栈前要判断是否栈空，如果栈是空的，则出栈失败，否则将栈顶元素暂存给一个变量，栈顶指针向下移动一个空间（top--）。

```
bool PopStack(SqStack &S, ElemType &e) //删除 S 的栈顶元素，暂存在变量 e 中

{
    if (S.base == S.top){ //栈空
        return false;
    }

    e = *(--S.top); //栈顶指针减 1，将栈顶元素赋给 e

    return true;
}
```

获取栈顶元素

取栈顶元素和出栈不同，取栈顶元素只是把栈顶元素复制一份，栈的元素个数不变，而出栈是指栈顶元素取出，栈内不再包含这个元素。

```
ElemType GetTop(SqStack &S) //返回 S 的栈顶元素，栈顶指针不变
{
    if (S.top != S.base){ //栈非空
        return *(S.top - 1); //返回栈顶元素的值，栈顶指针不变
    }else{
        return -1;
    }
}
```

判断空栈

```
bool IsEmpty(SqStack &S){//判断栈是否为空
    if (S.top == S.base){
        return true;
    }else{
        return false;
    }
}
```

完整源码实现：

```
#include <Windows.h>
#include <iostream>
#include <stdlib.h>

using namespace std;

#define MaxSize  128  //预先分配空间，这个数值根据实际需要预估确定

typedef int ElemType;

typedef struct _SqStack{
    ElemType *base;    //栈底指针
    ElemType *top;     //栈顶指针
}SqStack;


bool InitStack(SqStack &S) //构造一个空栈 S
{
    S.base = new ElemType[MaxSize];//为顺序栈分配一个最大容量为 Maxsize 的空间
    if (!S.base) //空间分配失败
        return false;
    S.top=S.base; //top 初始为 base，空栈
    return true;
}


bool PushStack(SqStack &S, ElemType e) // 插入元素 e 为新的栈顶元素

{
    if (S.top-S.base == MaxSize) //栈满
        return false;

    *(S.top++) = e; //元素 e 压入栈顶，然后栈顶指针加 1，等价于*S.top=e; S.top++;

    return true;
}


bool PopStack(SqStack &S, ElemType &e) //删除 S 的栈顶元素，暂存在变量 e 中

{
    if (S.base == S.top){ //栈空
        return false;
```

```
    }

    e = *(--S.top); //栈顶指针减 1，将栈顶元素赋给 e

    return true;
}


ElemType GetTop(SqStack &S) //返回 S 的栈顶元素，栈顶指针不变
{
    if (S.top != S.base){ //栈非空
        return *(S.top - 1); //返回栈顶元素的值，栈顶指针不变
    }else{
       return -1;
    }
}

int GetSize(SqStack &S){//返回栈中元素个数
    return (S.top-S.base);
}

bool IsEmpty(SqStack &S){//判断栈是否为空
    if (S.top == S.base){
        return true;
    }else{
        return false;
    }
}

void DestoryStack(SqStack &S)
{
    if(S.base){
        free(S.base);
       S.base = NULL;
       S.top = NULL;
    }
}


int main()
{
    int n,x;
    SqStack S;
    InitStack(S);//初始化一个顺序栈 S
    cout <<"请输入元素个数 n： " <<endl;
    cin>>n;
    cout <<"请依次输入 n 个元素，依次入栈： " <<endl;
```

```
    while(n--)
    {
        cin>>x; //输入元素
        PushStack(S, x);
    }

    cout <<"元素依次出栈：" <<endl;
    while(!IsEmpty(S))//如果栈不空，则依次出栈
    {
        cout<<GetTop(S)<<"\t";//输出栈顶元素
        PopStack(S, x); //栈顶元素出栈
    }
    cout <<endl;

    DestoryStack(S);
    system("pause");
    return 0;
}
```

课后思考:

如果把栈的结构体定义改为如下形式，该如何实现操作栈的算法？

```
typedef struct _SqStack{
    int top;    //栈顶的位置
    ElemType *base;     //栈底指针
}SqStack;
```

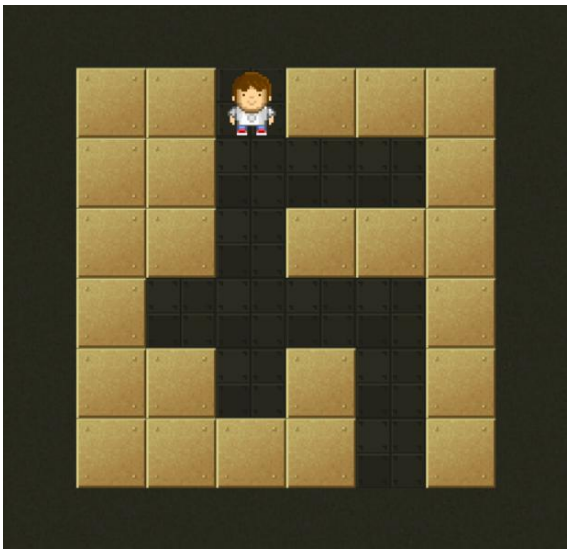
第 4 节 栈的链式存储结构

(作为课后作业自行实现，完成后交与 Martin 老师检查)

第 5 节 栈的企业级应用案例

5.1 迷宫求解

世界上最大的树篱迷宫：北爱尔兰 Castewallan 森林公园和平迷宫



```
{0,  0,  1,  0,  0,  0 },
{0,  0,  1,  1,  1,  0 },
{0,  0,  1,  0,  0,  0 },
{0,  1,  1,  1,  1,  0 },
{0,  0,  1,  0,  1,  0 },
{0,  0,  0,  0,  1,  0 },
```

找迷宫通路需要使用回溯法，找迷宫通路是对回溯法的一个很好的应用，实现回溯的过程用到数据结构—栈！

回溯法：对一个包括有很多个结点，每个结点有若干个搜索分支的问题，把原问题分解为若干个子问题求解的算法；当搜索到某个结点发现无法再继续搜索下去时，就让搜索过程回溯(回退)到该节点的前一个结点，继续搜索该节点外的其他尚未搜索的分支；如果发现该结点无法再搜索下去，就让搜索过程回溯到这个结点的前一结点继续这样的搜索过程；这样的搜索过程一直进行到搜索到问题的解或者搜索完了全部可搜索分支没有解存在为止。

完整源码实现：

maze.h

```
#pragma once
#include<stdio.h>
#include<stdlib.h>

#define MAXSIZE 100

typedef struct _Position{//迷宫坐标
    int  _x;
    int  _y;
}Position;

#define MaxSize  128  //预先分配空间，这个数值根据实际需要预估确定

typedef Position ElemType;

typedef struct _SqStack{
    ElemType *base;    //栈底指针
    ElemType *top;     //栈顶指针
}SqStack;


bool InitStack(SqStack &S) //构造一个空栈 S
{
    S.base = new ElemType[MaxSize];//为顺序栈分配一个最大容量为 Maxsize
的空间
    if (!S.base) //空间分配失败
        return false;
    S.top=S.base; //top 初始为 base，空栈
    return true;
}


bool PushStack(SqStack &S, ElemType e) // 插入元素 e 为新的栈顶元素

{
```

```
    if (S.top-S.base == MaxSize) //栈满
        return false;

    *(S.top++) = e; //元素 e 压入栈顶，然后栈顶指针加 1，等价于*S.top=e;
S.top++;

    return true;
}


bool PopStack(SqStack &S, ElemType &e) //删除 S 的栈顶元素，暂存在变量 e
中

{
    if (S.base == S.top){ //栈空
        return false;
    }

    e = *(--S.top); //栈顶指针减 1，将栈顶元素赋给 e

    return true;
}


ElemType* GetTop(SqStack &S) //返回 S 的栈顶元素，栈顶指针不变
{
    if (S.top != S.base){ //栈非空
        return S.top - 1; //返回栈顶元素的值，栈顶指针不变
    }else{
        return NULL;
    }
}

int GetSize(SqStack &S){//返回栈中元素个数
    return (S.top-S.base);
}

bool IsEmpty(SqStack &S){//判断栈是否为空
    if (S.top == S.base){
        return true;
    }else{
        return false;
    }
}

void DestoryStack(SqStack &S){//销毁栈
    if(S.base){
```

```
        free(S.base);

        S.base = NULL;
        S.top = NULL;
    }
}
```

maze.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "maze.h"
#include <assert.h>


#define ROW 6
#define COL 6

typedef struct _Maze{
    int map[ROW][COL];
}Maze;


void InitMaze(Maze* m, int map[ROW][COL])  //迷宫的初始化
{
    for (int i = 0;i< ROW ;++i)
    {
        for (int j = 0; j < COL; ++j)
        {
            m->map[i][j] = map[i][j];
        }
    }
}
void PrintMaze(Maze* m)  //打印迷宫
{
    for (int i = 0; i < ROW; ++i)
    {
        for (int j = 0; j < COL; ++j)
        {
            printf("%d ",m->map[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

```
int IsValidEnter(Maze* m, Position cur)  //判断是否是有效的入口
{
    assert(m);

    if ((cur._x == 0 || cur._x == ROW - 1)
        || (cur._y == 0 || cur._y == COL - 1)
        && (m->map[cur._x][cur._y] == 1))
        return 1;
    else
        return 0;
}

int IsNextPass(Maze* m,Position cur, Position next) //判断当前节点的下一个节点能否走通
{
    assert(m);

    //判断 next 节点是否为 cur 的下一节点
    if(((next._x == cur._x) && ((next._y == cur._y+1)||(next._y == cur._y-1)))  //在同一行上并且相邻
        ||((next._y == cur._y) && ( (next._x == cur._x+1)||(next._x == cur._x-1))){//或在同一列上并且相邻

            //判断下一个节点是否在迷宫里面
            if (((next._x >= 0 && next._x < ROW) || (next._y >= 0 && next._y < COL))
                &&(m->map[next._x][next._y] == 1)){
                    return 1;
            }
    }

    return 0;
}

int IsValidExit(Maze* m, Position cur,Position enter)  //判断当前节点是不是有效的迷宫出口
{
    assert(m);

    //这里首先得保证该节点不是入口点，其次只要它处在迷宫的边界即可
    if ((cur._x  != enter._x || cur._y != enter._y)
        && ((cur._x == 0 || cur._x == ROW - 1)
        || (cur._y == 0 || cur._y == COL - 1)))
    {
        return 1;
    }
```

```
    else
        return 0;
}

int PassMaze(Maze* m,Position enter,SqStack* s)  //找迷宫通路
{
    assert(m && IsValidEnter(m,enter) == 1);  //对给的迷宫的入口进行合法性判断

    Position cur = enter;
    Position next;

    PushStack(*s, cur);  //首先将迷宫的入口压入栈中
    m->map[cur._x][cur._y] = 2;  //将入口值改为 2
    //PrintMaze(m);

    while (!IsEmpty(*s)) {
        cur = *GetTop(*s);
        //printf("cur: %d %d\n",cur._x, cur._y);
        if (IsValidExit(m,cur,enter) == 1)  //判断当前位置是否出口
            return 1;

        //尝试向左一步：看当前节点的左一个节点能不能走通
        next = cur;
        next._y = cur._y - 1;
        if (IsNextPass(m, cur, next) == 1)
        {
            PushStack(*s, next);
            m->map[next._x][next._y] = m->map[cur._x][cur._y] + 1;
            //PrintMaze(m);
            continue;
        }

        //尝试向上一步:看当前节点的上一个节点能不能走通
        next = cur;
        next._x = cur._x - 1;
        if (IsNextPass(m,cur,next) == 1)  //next 节点能够走通时，将其压入栈中
        {
            PushStack(*s,next);
            m->map[next._x][next._y] = m->map[cur._x][cur._y]+1;  //将 next 节点的值等于 cur 节点的值加 1
            //PrintMaze(m);
            continue;
        }

        //右：看当前节点的向右的一个节点能不能走通
```

```
        next = cur;
        next._y = cur._y + 1;
        if (IsNextPass(m, cur,next) == 1)
        {
            PushStack(*s, next);
            m->map[next._x][next._y] = m->map[cur._x][cur._y] + 1;
            //PrintMaze(m);
            continue;
        }
        //下：看当前节点的下一个节点能不能走通
        next = cur;
        next._x = cur._x + 1;
        if (IsNextPass(m, cur,next) == 1)
        {
            PushStack(*s, next);
            m->map[next._x][next._y] = m->map[cur._x][cur._y] + 1;
            //PrintMaze(m);
            continue;
        }

        //走到这里说明当前节点的四个方向都走不通，进行回溯，看前一个节点未被遍历的方向是否还能走通
        Position tmp;
        PopStack(*s, tmp);
    }

    return 0;
}

int main()
{
    int map[ROW][COL] = {  //用二维数组描绘迷宫：1 代表通路，0 代表墙
        0,0,1,0,0,0,
        0,0,1,1,1,0,
        0,0,1,0,0,0,
        0,1,1,1,1,0,
        0,0,1,0,1,0,
        0,0,0,0,1,0
    };

    Maze m;
    Position enter;  //迷宫入口
    enter._x = 0;
    enter._y = 2;
    InitMaze(&m, map);
    PrintMaze(&m);
    //system("pause");
```

```
    //exit(0);
    SqStack s;  //定义栈，保存已走过的坐标轨迹，便于回溯
    InitStack(s);  //栈的初始

    int ret = PassMaze(&m,enter,&s); //使用栈和回溯法解开迷宫
    if(ret){
        printf("恭喜你！终于找到了出口~\n");
    }else {
        printf("不是我笨！实在没有出口~\n");
    }
    PrintMaze(&m);
    system("pause");
    return 0;
}
```

5.2 表达式求值

给定一个只包含加减乘除法运算的算术表达式，请你编程计算表达式的值。

输入格式

输入一行，为需要你计算的表达式，表达式中只包含数字、加法运算符“+”、 减法运算符 “-”、乘法运算符“*”和 除法运算符“/”，且没有括号，不考虑数值的范围（溢出），待求解的表达式以“=”号结束。

如： `12+3*6/3+4*5=`

请大家仔细阅读源码，理清思路，用文字描述出算法的思路！(可以使用流程图)

完整源码实现：

expression.h

```
#pragma once
#include<stdio.h>
#include<stdlib.h>

#define MAXSIZE 100



#define MaxSize  128  //预先分配空间，这个数值根据实际需要预估确定

typedef int ElemType;

typedef struct _SqStack{
    ElemType *base;    //栈底指针
    ElemType *top;     //栈顶指针
}SqStack;


bool InitStack(SqStack &S) //构造一个空栈 S
{
    S.base = new ElemType[MaxSize];//为顺序栈分配一个最大容量为 Maxsize 的空间
    if (!S.base) //空间分配失败
        return false;
    S.top=S.base; //top 初始为 base，空栈
    return true;
}


bool PushStack(SqStack &S, ElemType e) // 插入元素 e 为新的栈顶元素

{
    if (S.top-S.base == MaxSize) //栈满
       return false;

    *(S.top++) = e; //元素 e 压入栈顶，然后栈顶指针加 1，等价于*S.top=e; S.top++;

    return true;
}
```

```
bool PopStack(SqStack &S, ElemType &e) //删除 S 的栈顶元素，暂存在变量 e 中

{
    if (S.base == S.top){ //栈空
        return false;
    }

    e = *(--S.top); //栈顶指针减 1，将栈顶元素赋给 e

    return true;
}


ElemType* GetTop(SqStack &S) //返回 S 的栈顶元素，栈顶指针不变
{
    if (S.top != S.base){ //栈非空
        return S.top - 1; //返回栈顶元素的值，栈顶指针不变
    }else{
        return NULL;
    }
}

int GetSize(SqStack &S){//返回栈中元素个数
    return (S.top-S.base);
}

bool IsEmpty(SqStack &S){//判断栈是否为空
    if (S.top == S.base){
        return true;
    }else{
        return false;
    }
}

void DestoryStack(SqStack &S){//销毁栈
    if(S.base){
        free(S.base);

        S.base = NULL;
        S.top = NULL;
    }
}
```

expression.cpp

```
#include <stdio.h>
#include <iostream>
#include "expression.h"

using namespace std;

//比较 lhs 的优先级是否不高于 rhs，rhs 表示栈顶的符号
bool isLarger(const int &lhs, const int &rhs)
{
    if ((rhs == '+' || rhs == '-') && (lhs == '*' || lhs == '/')){
        return true;
    }

    return false;
}


int operate(int left, int right, int op)//对运算符求值
{
    int result = 0;
    cout<<"left:"<<left<<" right:"<<right<< (char)op<<endl;
    switch (op)
    {
    case '+':
        result = left + right;
        break;
    case '-':
        result = left - right;
        break;
    case '*':
        result = left * right;
        break;
    case '/':
        result = left / right;
        break;
    default:
        break;
    }
    cout<<"result: "<<result<<endl;

    return result;
}

int calculate(string input)
{
    SqStack data_stack;//操作数堆栈
```

```
    SqStack opt_stack; //运算符堆栈
    int status = 0; //0-接受左操作数 1-接受运算符 "+-*/"  2-接受右操作数
    int ldata=0, rdata=0;
    char last_opt = '\0';

    //初始化栈：操作数和操作符
    InitStack(data_stack);
    InitStack(opt_stack);

    for( int i=0; i<input.length(); i++)
    {
        if(isspace(input[i])) continue;//空格键等忽略掉

        switch(status){
        case 0:
            if(isdigit(input[i])){
                ldata*=10;
                ldata+=input[i]-'0';
            }else {
                cout<<"得到左操作数:"<<ldata<<endl;
                PushStack(data_stack, ldata);//左操作数入栈
                i--;
                status=1;
            }
            break;
        case 1:
            if(input[i]=='+' || input[i]='-' || input[i]=='*' ||
input[i]=='/'){
                if(IsEmpty(opt_stack)){//第一个运算符,暂时不做任何处理,运
算符先入栈保存
                    cout<<"符号栈为空"<<endl;
                    PushStack(opt_stack, input[i]);//操作符入栈
                    cout<<"符号"<<(char)input[i]<<"入栈"<<endl;
                    status = 2;
                }else {//非第一个运算符，则与之前的运算符比较优先级
                    cout<<"isLarger:"<<(char)(*GetTop(opt_stack))<<" &
"<<input[i]<<endl;
                    if(isLarger(input[i], *GetTop(opt_stack))){
                        cout<<"true"<<endl;
                        PushStack(opt_stack, input[i]);//操作符入栈
                        cout<<"符号"<<(char)input[i]<<"入栈"<<endl;
                        status = 2;
                        rdata = 0;
                    }else {//当前运算符的优先级不高于前一个运算符，则计算
前一个运算符的值

                        int  left=0, right = 0;
```

```
                    int opt;
                    cout<<"false"<<endl;
                    do{
                        PopStack(data_stack, right);
                        PopStack(data_stack, left);
                        PopStack(opt_stack, opt);
                        cout<<"符号"<<(char)opt<<"出栈"<<endl;
                        cout<<"计算前一个运算符"<<(char)opt<<endl;
                        int result = operate(left, right, opt);
                        PushStack(data_stack, result);
                    }while(!IsEmpty(opt_stack) && !isLarger(input[i],
*GetTop(opt_stack)));


                    PushStack(opt_stack, input[i]);
                    cout<<"符号"<<(char)input[i]<<"入栈"<<endl;
                    status = 2;
                    rdata = 0;
                }
            }
        }else if(input[i]=='='){//计算结果
            int opt, result;
            do{
                PopStack(data_stack, rdata);
                PopStack(data_stack, ldata);
                PopStack(opt_stack, opt);

                result = operate(ldata, rdata, opt);
                PushStack(data_stack, result);
            }while(!IsEmpty(opt_stack));

            return result;

        }
        else {
            cerr<<"输入运算符错误！"<<endl;
        }

        break;
    case 2:
        if(isdigit(input[i])){
            rdata*=10;
            rdata+=input[i]-'0';
        }else {
            cout<<"得到右操作数:"<<rdata<<endl;
            PushStack(data_stack, rdata);//右操作数入栈
            i--;
```

```
                status=1;
            }

            break;
        }

    }
    return -1;//最后的结果为栈顶元素
}

int main(int argc, char const *argv[])
{
    string str = "12+3*6/3+4*5=";
    cout << calculate(str) << endl;
    system("pause");
    return 0;
}
```