数据结构与算法（三）

张铭 主讲

采用教材：张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008. 6（“十一五”国家级规划教材）

http://www.jpk.pku.edu.cn/pkujpk/course/sjjg

第3章　栈与队列

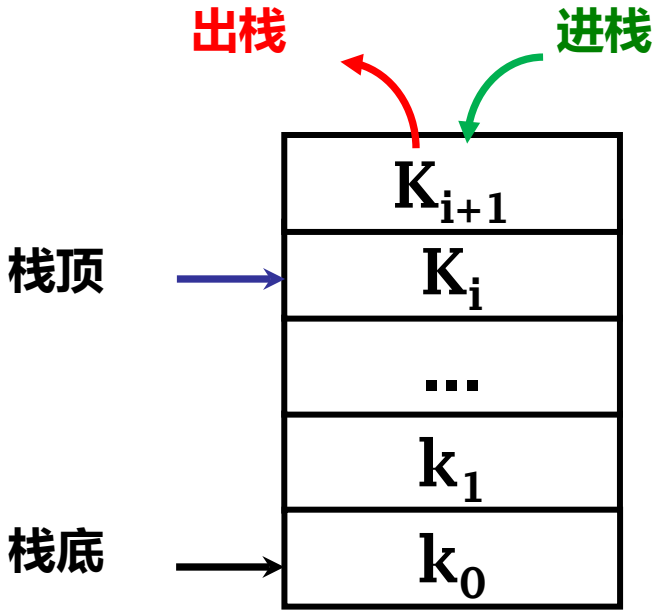
- 栈
- 栈的应用
  - 递归到非递归的转换
- 队列

操作受限的线性表

- **栈 (Stack)**
  - 运算只在表的一端进行
- **队列 (Queue)**
  - 运算只在表的两端进行

栈定义

- **后进先出 (Last In First Out)**
  - 一种限制访问端口的线性表
- **主要操作**
  - 进栈 (push)　出栈 (pop)
- **应用**
  - 表达式求值
  - 消除递归
  - 深度优先搜索

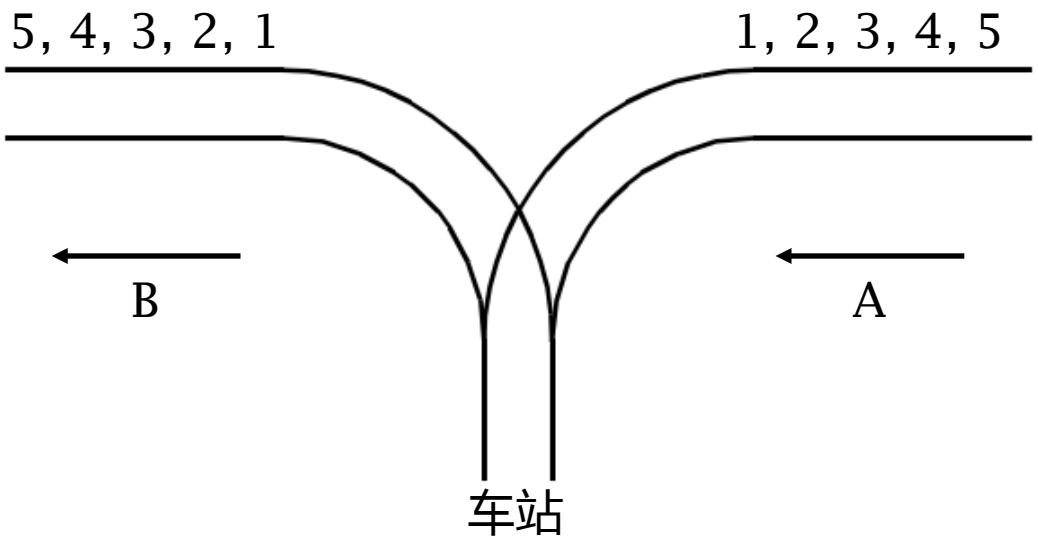


栈的抽象数据类型

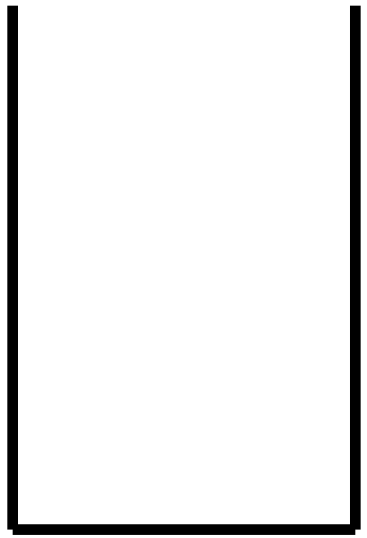
```
template <class T>
class Stack {
public:                          // 栈的运算集
    void clear();                // 变为空栈
    bool push(const T item);
                                 // item入栈，成功返回真，否则假
    bool pop(T& item);           // 返回栈顶内容并弹出，成功返回真，否则假
    bool top(T& item);           // 返回栈顶但不弹出，成功返回真，否则假
    bool isEmpty();              // 若栈已空返回真
    bool isFull();               // 若栈已满返回真
};
```

火车进出栈问题

- 判断火车的出栈顺序是否合法
  - http://poj.org/problem?id=1363
- 编号为1,2,…,n的n辆火车依次进站，给定一个n的排列，判断是否是合法的出站顺序？



利用合法的重构找冲突



思考

- 若入栈的顺序为1,2,3,4，
  那么出栈的顺序可以有哪些?

- 从初始输入序列1，2，…，n，希望利用一个栈得到输出序列p_1，p_2，…，p_n (它们是1， 2，…，n的一种排列)。若存在下标i，j，k，满足i<j<k 同时 $P_j<P_k<P_i$，则输出序列是否合法？

栈的实现方式

- **顺序栈 (Array-based Stack)**
  - 使用向量实现，本质上是顺序表的简化版
    - 栈的大小
  - 关键是确定**哪一端**作为栈顶
  - 上溢，下溢问题

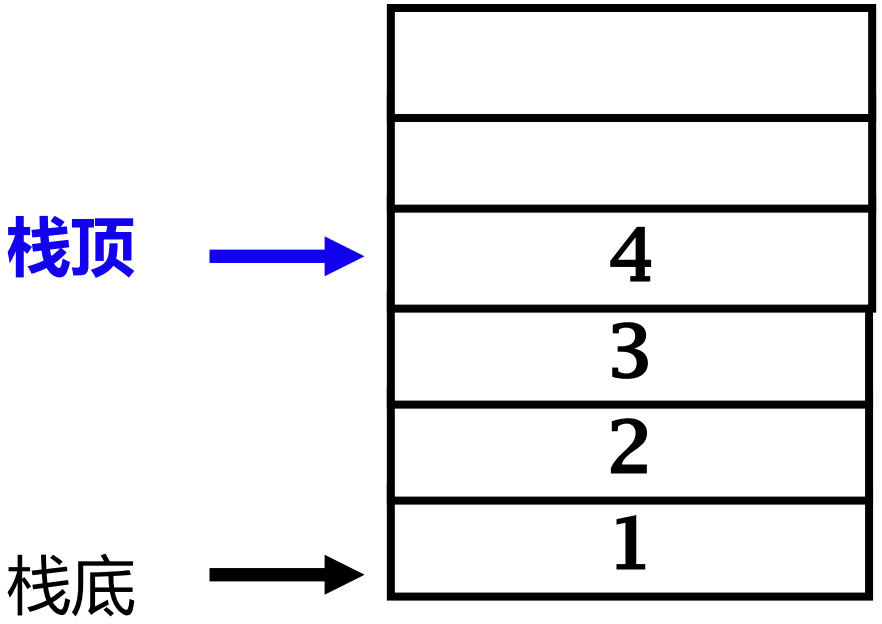
- **链式栈 (Linked Stack)**
  - 用单链表方式存储，其中指针的方向是从栈顶向下链接

顺序栈的类定义

```
template <class T> class arrStack : public Stack <T> {
private:                                  // 栈的顺序存储
    int mSize;                            // 栈中最多可存放的元素个数
    int top;                              // 栈顶位置，应小于mSize
    T *st;                                // 存放栈元素的数组
public:                                   // 栈的运算的顺序实现
    arrStack(int size) {                  // 创建一个给定长度的顺序栈实例
        mSize = size; top = -1; st = new T[mSize];
    }
    arrStack() {                          // 创建一个顺序栈的实例
        top = -1;
    }
    ~arrStack() { delete [] st; }
    void clear() { top = -1; }  // 清空栈
}
```

顺序栈

- 按压入先后次序，最后压入的元素编号为4，然后依次为3,2,1



顺序栈的溢出

- **上溢 (Overflow)**
  - 当栈中已经有maxsize个元素时，如果再做进栈运算，所产生的现象
- **下溢 (Underflow)**
  - 对空栈进行出栈运算时所产生的现象

压入栈顶

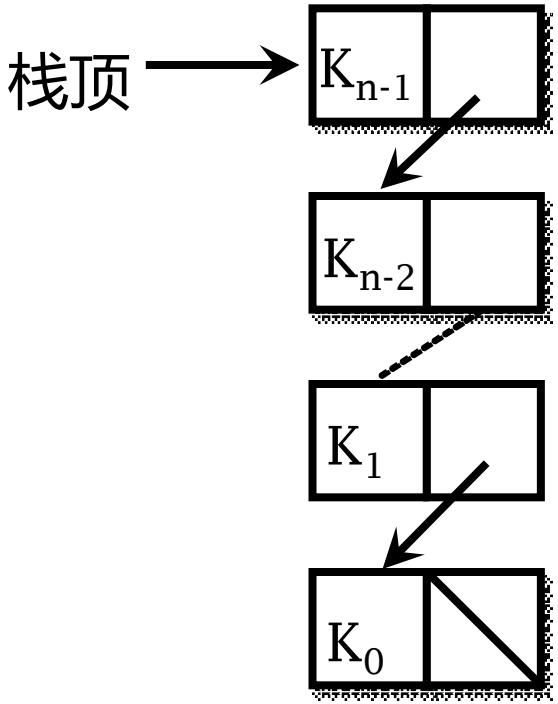
```
bool arrStack<T>::push(const T item) {
    if (top == mSize-1) {        // 栈已满
        cout << "栈满溢出" << endl;
        return false;
    } else {                     // 新元素入栈并修改栈顶指针
        st[++top] = item;
        return true;
    }
}
```

从栈顶弹出

```
bool arrStack<T>::pop(T & item) {   // 出栈
    if (top == -1) {                // 栈为空
        cout << "栈为空，不能执行出栈操作"<< endl;
        return false;
    } else {
        item = st[top--];   // 返回栈顶，并缩减1
        return true;
    }
}
```

链式栈的定义

- 用单链表方式存储
- 指针的方向从**栈顶向下**链接



链式栈的创建

```
template <class T> class lnkStack : public Stack <T> {
private:                                    // 栈的链式存储
    Link<T>* top;                           // 指向栈顶的指针
    int size;                               // 存放元素的个数
public:                                     // 栈运算的链式实现
    lnkStack(int defSize) {                 // 构造函数
        top = NULL; size = 0;
    }
    ~lnkStack() {                           // 析构函数
        clear();
    }
}
```

压入栈顶

```
// 入栈操作的链式实现
bool lnksStack<T>:: push(const T item) {
    Link<T>* tmp = new Link<T>(item, top);
    top = tmp;
    size++;
    return true;
}

Link(const T info, Link* nextValue)  {// 具有两个参数的Link构造函数
        data = info;
        next = nextValue;
}
```

从单链栈弹出元素

```
// 出栈操作的链式实现
bool lnkStack<T>:: pop(T& item) {
    Link <T> *tmp;
    if (size == 0) {
        cout << "栈为空，不能执行出栈操作"<< endl;
        return false;
    }
    item = top->data;
    tmp = top->next;
    delete top;
    top = tmp;
    size--;
    return true;
}
```

顺序栈和链式栈的比较

- **时间效率**
  - 所有操作都只需常数时间
  - 顺序栈和链式栈在时间效率上难分伯仲

- **空间效率**
  - 顺序栈须说明一个固定的长度
  - 链式栈的长度可变，但增加结构性开销

顺序栈和链式栈的比较

- 实际应用中，顺序栈比链式栈用得更广泛
  - 顺序栈容易根据栈顶位置，进行相对位移，快速定位并读取栈的内部元素
  - 顺序栈读取内部元素的时间为$O(1)$，而链式栈则需要沿着指针链游走，显然慢些，读取第k个元素需要时间为$\mathbf{O}(k)$

- 一般来说，栈不允许“读取内部元素”，只能在栈顶操作

思考： STL中关于栈的函数

- top函数表示取栈顶元素，将结果返回给用户
- pop函数表示将栈顶元素弹出（如果栈不空）
  - pop函数仅仅是一个操作，并不将结果返回。
  - pointer = aStack.pop()？ **错误！**

- STL为什么这两个操作分开？ 为什么不提供ptop？

栈的应用

- 栈的特点：**后进先出**
  - 体现了元素之间的透明性
- 常用来处理具有递归结构的数据
  - 深度优先搜索
  - **表达式求值**
  - 子程序 / 函数调用的管理
  - **消除递归**

计算表达式的值

- 表达式的递归定义
  - 基本符号集：{0，1，...，9，+，-，*，/，（，）}
  - 语法成分集：{<表达式>，<项>，<因子>，<常数>，<数字>}
- 中缀表达式　　23+(34*45)/(5+6+7)
- 后缀表达式　　23 34 45 * 5 6 + 7 + / +

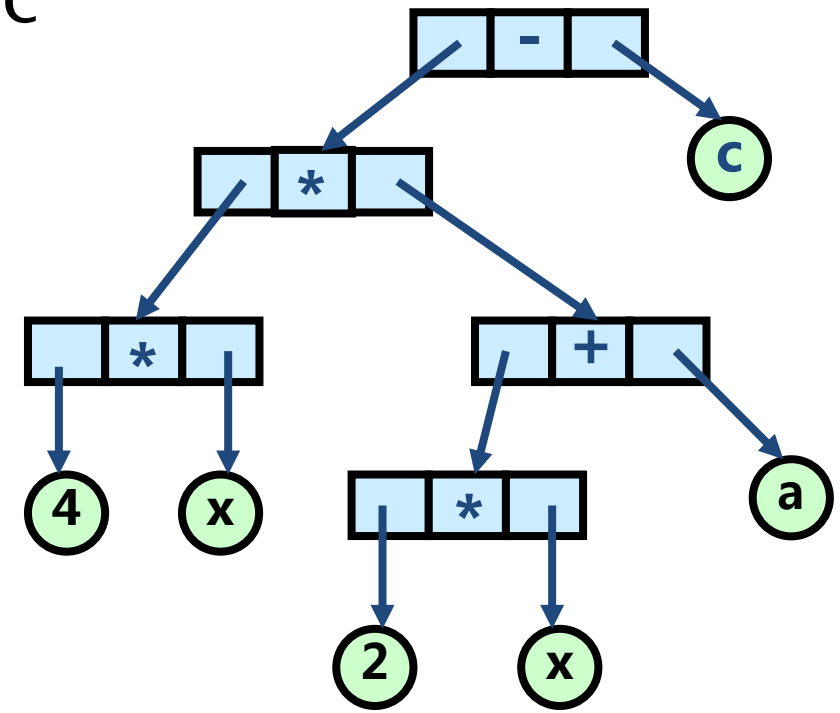
中缀表达式

- **中缀表达式**

  4 * x * (2 * x + a) – c

  – 运算符在中间

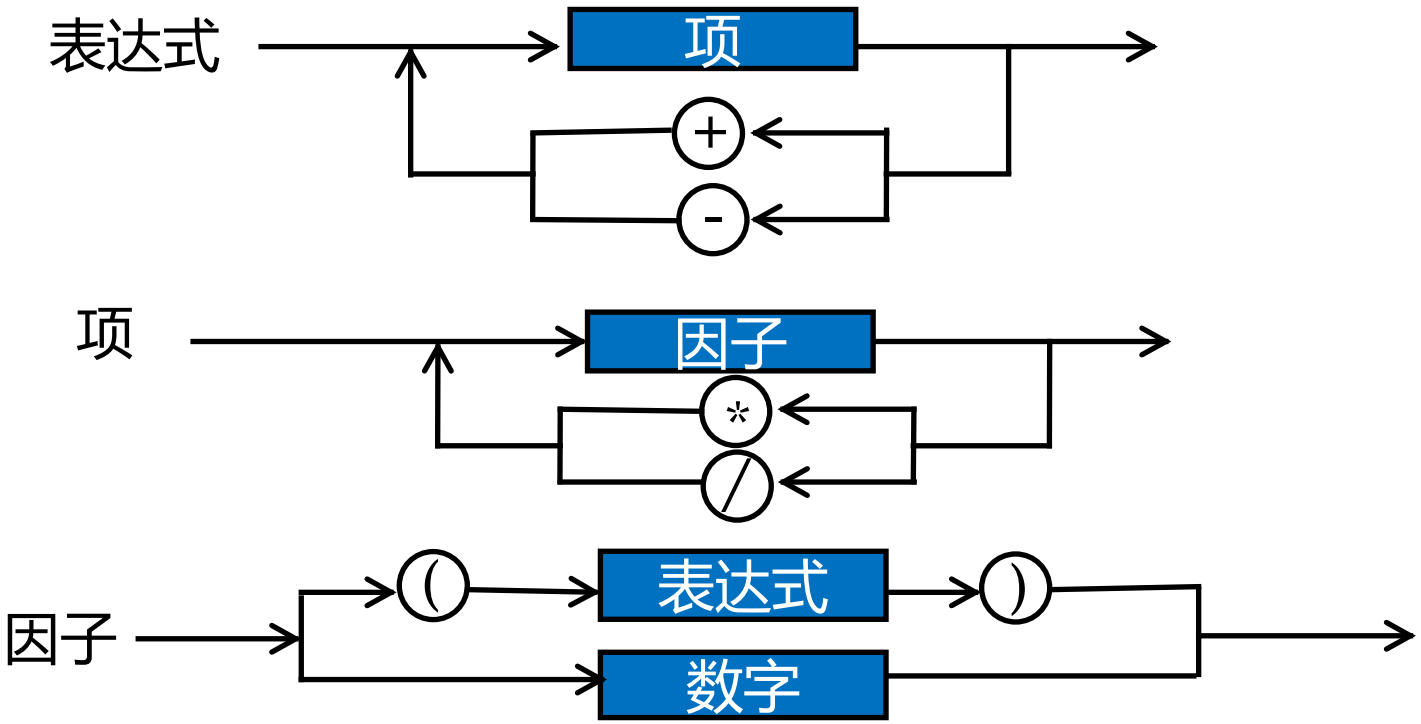
  – 需要括号改变优先级



中缀表达法的语法公式

<表达式> ::= <项> + <项>
| <项> - <项>
| <项>
<项> ::= <因子> * <因子>
| <因子> / <因子>
| <因子>
<因子> ::= <常数>
| (<表达式>)
<常数> ::= <数字>
| <数字> <常数>
<数字> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

表达式的递归图示

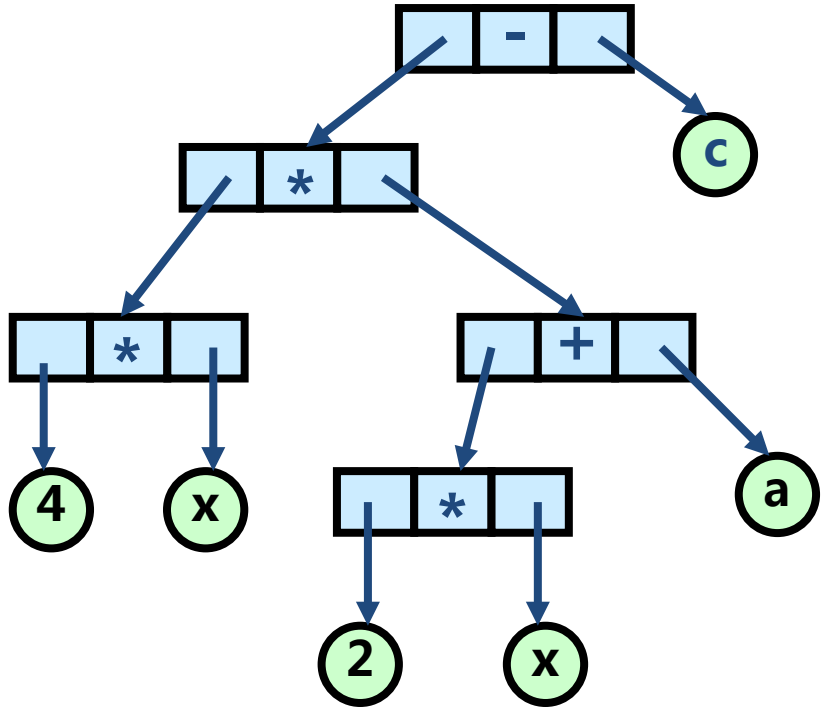


后缀表达式

- **后缀表达式**

  4 x * 2 x * a + * c -

  - 运算符在后面
  - 完全不需要括号



后缀表达式

<表达式> ::= <项> <项> +
| <项> <项> -
| <项>
<项> ::= <因子> <因子> *
| <因子> <因子> /
| <因子>
<因子> ::= <常数>
<常数> ::= <数字>
| <数字> <常数>
<数字> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

后缀表达式的计算

- 23 34 45 * 5 6 + 7 + / + = ?

计算特点？

中缀和后缀表达式的主要异同？

23 + 34 * 45 / (5 + 6 + 7) = ?

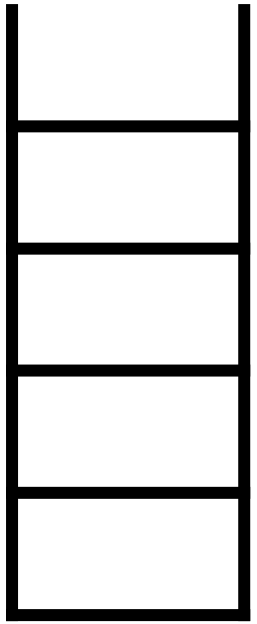
23 34 45 * 5 6 + 7 + / + = ?

待处理后缀表达式：

23 34 45 * 5 6 + 7 + / +

栈状态的变化

（看视频）



后缀表达式求值

- 循环：依次顺序读入表达式的符号序列（假设以 = 作为输入序列的结束），并根据读入的元素符号逐一分析
  1. 当遇到的是一个操作数，则压入栈顶
  2. 当遇到的是一个运算符，就从栈中两次取出栈顶，按照运算符对这两个操作数进行计算。然后将计算结果压入栈顶
- 如此继续，直到遇到符号 = ，这时栈顶的值就是输入表达式的值

后缀计算器的类定义

```
class Calculator {
private:
    Stack<double> s;              // 这个栈用于压入保存操作数
    // 从栈顶弹出两个操作数opd1和opd2
    bool GetTwoOperands(double& opd1,double& opd2);
    // 取两个操作数，并按op对两个操作数进行计算
    void Compute(char op);
public:
    Calculator(void){} ;          // 创建计算器实例，开辟一个空栈
    void Run(void);               // 读入后缀表达式，遇“=”符号结束
    void Clear(void);             // 清除计算器，为下一次计算做准备
};
```

后缀计算器的类定义

```
template <class ELEM>
bool Calculator<ELEM>::GetTwoOperands(ELEM& opnd1, ELEM& opnd2) {
    if (S.IsEmpty()) {
        cerr << "Missing operand!" <<endl;
        return false;
    }
    opnd1 = S.Pop(); // 右操作数
    if (S.IsEmpty()) {
        cerr << "Missing operand!" <<endl;
        return false;
    }
    opnd2 = S.Pop(); // 左操作数
    return true;
}
```

后缀计算器的类定义

```
template <class ELEM> void Calculator<ELEM>::Compute(char op) {
    bool result;  ELEM operand1, operand2;
    result = GetTwoOperands(operand1, operand2);
    if (result == true)
        switch(op) {
            case '+' : S.Push(operand2 + operand1); break;
            case '-' : S.Push(operand2 - operand1); break;
            case '*' : S.Push(operand2 * operand1); break;
            case '/' : if (operand1 == 0.0) {
                       cerr << "Divide by 0!" << endl;
                       S.ClearStack();
                   } else S.Push(operand2 / operand1);
                   break;
        }
    else S.ClearStack();
}
```

后缀计算器的类定义

```
template <class ELEM> void Calculator<ELEM>::Run(void) {
    char c; ELEM newoperand;
    while (cin >> c, c != '=') {
        switch(c) {
            case '+': case '-': case '*': case '/':
                Compute(c);
                break;
            default:
                cin.putback(c); cin >> newoperand;
                S.Push(newoperand);
                break;
        }
    }
    if (!S.IsEmpty())
        cout << S.Pop() << endl;          // 印出求值的最后结果
}
```

思考

- 1. 栈往往用单链表实现。可以用双链表吗？哪个更好？

- 2. 请总结前缀表达式的性质，以及求值过程。

数据结构与算法

谢谢聆听

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6。“十一五”国家级规划教材