加油包

- STL: Vector, Map等。（软件基础实践课程根据要求，数据结构课程原则不使用STL，栈和队列作为辅助数据结构时除外)关键是掌握如何实现的。
- 容器、迭代器概念，可以扩展，不要求掌握。
- 使用在线帮助；彼此改bug；bug诊断；最后的求助？
- 不着急，不懒惰。
- 程序的健壮性分析问题：功能 VS 性能

数据结构

教师：姜丽红 IST 实验室 jianglh@sjtu.edu.cn

助教：芮召普 ruishaopu@qq.com 江嘉晋 IST实验室 软件大楼5号楼5308

《软件基础实践》教师：杜东 IPADS 实验室 软件大楼3号楼4层

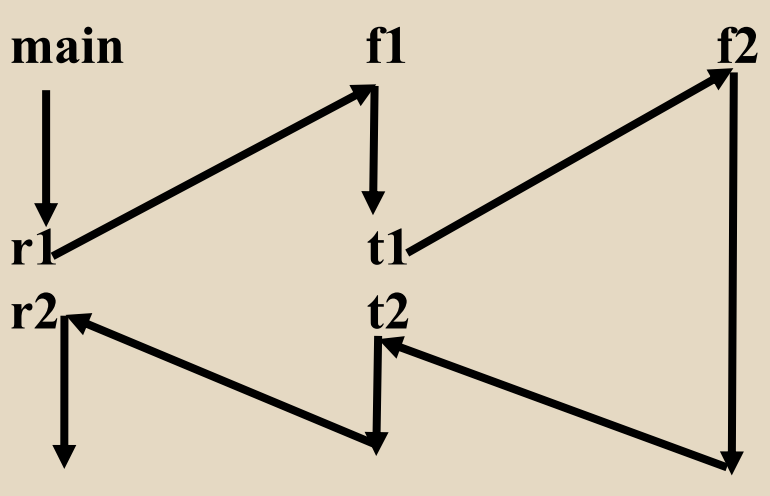
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

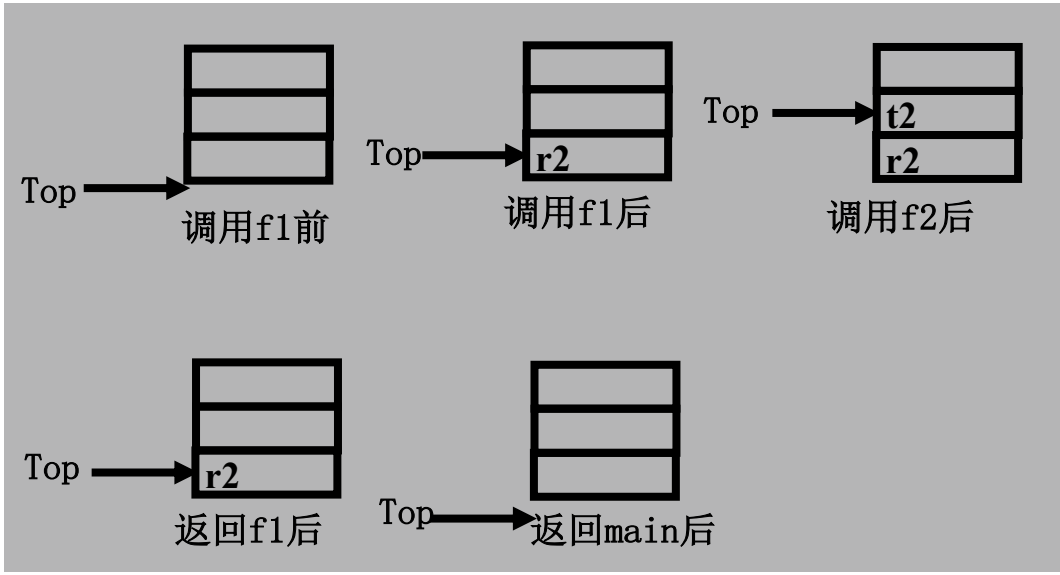
函数执行过程

```
void main()
{ …
   r1: f1();
   r2:
…
}
```

```
void f1()
{ …
   t1: f2();
   t2:
…
}
```

```
void f2()
{ …
…
}
```





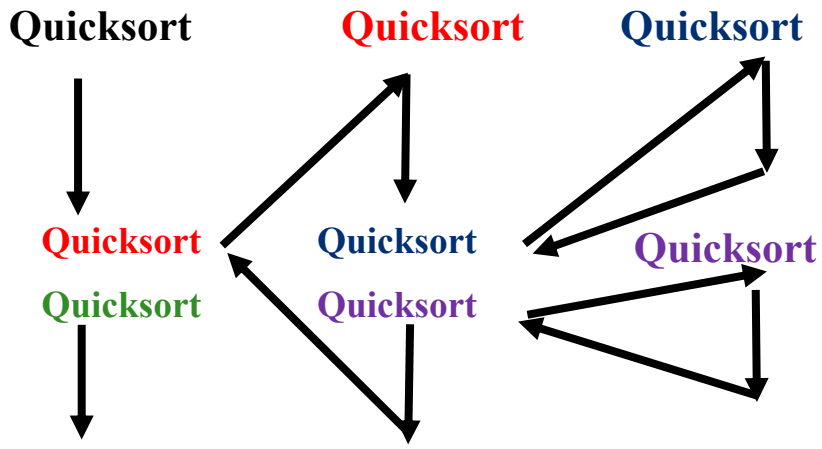
快速排序

6	3	8	1	4	9

4	3	1	6	8	9

mid

```
void quicksort(int a[], int low, int high)
{ int mid;
   if (low >= high) return;
   mid = divide(a, low, high);
   quicksort( a, low, mid-1);
   quicksort( a, mid+1, high);
}
```



递归函数的非递归实现：函数执行过程

- 递归是一种特殊的函数调用，是在一个函数中又调用了函数本身。
- 递归程序的本质是函数调用，而函数调用是要花费额外的时间和空间。
- 在系统内部，函数调用是用栈来实现，如果程序员可以自己控制这个栈，就可以消除递归调用。

快速排序的非递归实现

- 设置一个栈，将整个数组作为排序区间，其起始和终止位置进栈；
- 重复下列工作，直到栈空：
  - 从栈中弹出一个元素，即一个排序区间；
  - 将排序区间分成两半；
  - 检查每一半，如果多于两个元素，则进栈。
- 栈元素的格式：

```
struct  node
   { int left;//起始位置
     int right;//终止位置
    };
```

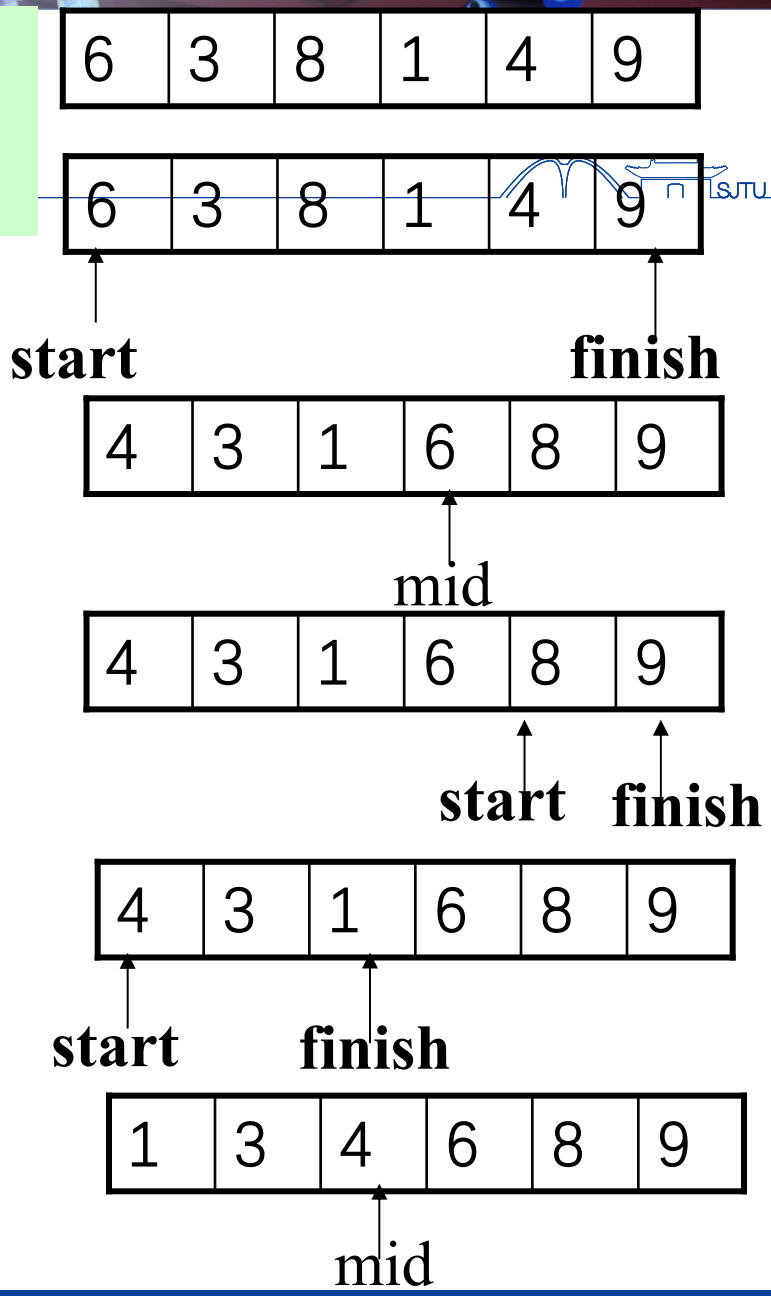
0			
5			

0	4		
2	5		

0			
2			

0			
1			

```
struct  node
        { int left;
          int right;
        };
```



```
void quicksort( int a[], int size)
{ seqStack <node> st;
  int mid, start, finish;
  node s;

  if (size <= 1) return;

  //排序整个数组
  s.left = 0;
  s.right = size - 1;
  st.push(s);
```

```
while (!st.isEmpty())
  { s = st.pop();
    start = s.left;
    finish = s.right;
    mid = divide(a, start, finish);
    if (mid - start > 1)
          { s.left = start;  s.right = mid - 1;
            st.push(s);   }
    if (finish - mid > 1)
          { s.left = mid + 1; s.right = finish;
            st.push(s);   }
  }
}
```

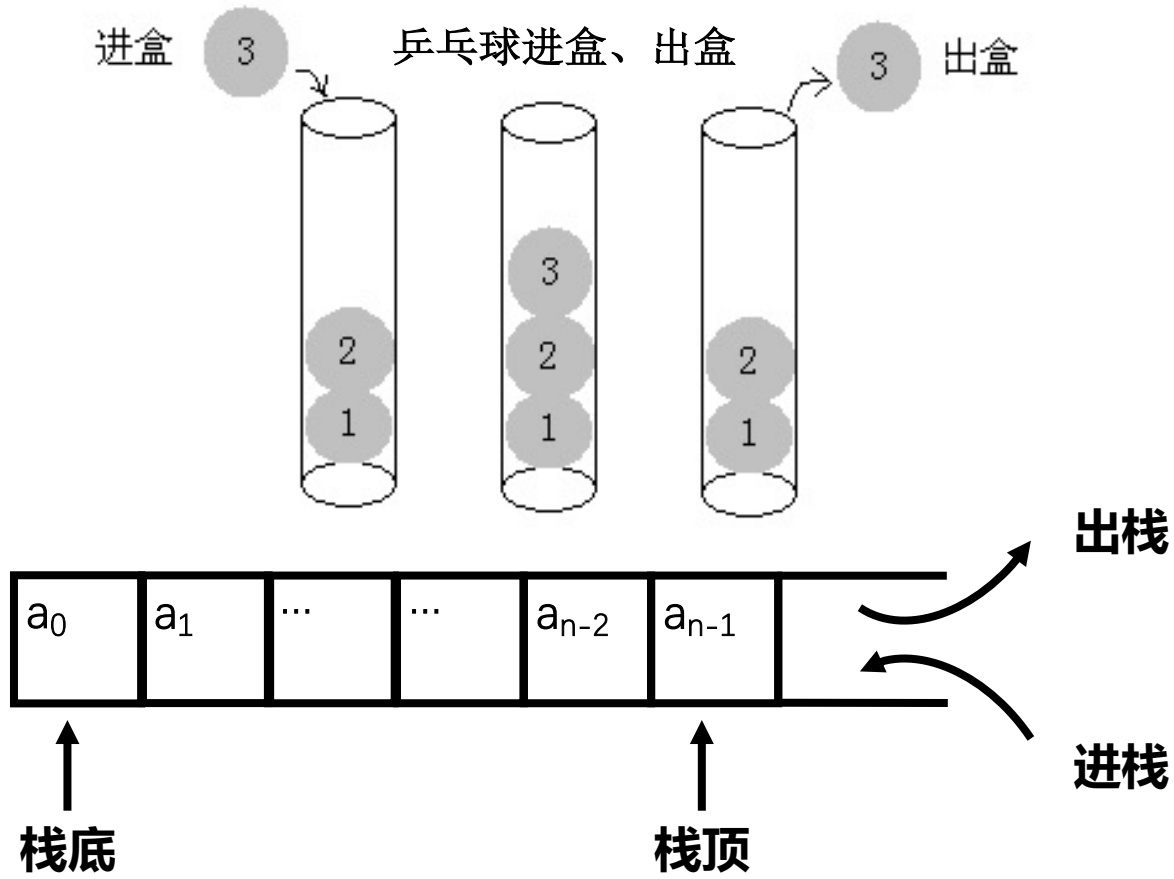
第三章 栈

- **栈的概念**
- **栈的顺序实现**
- **栈的链接实现**
- **栈的应用**

本章学习目标
1.分析应用对象，利用栈进行问题建模。
2.对栈实现创建、进栈、出栈以及销毁操作。

栈： 特殊类型的线性表

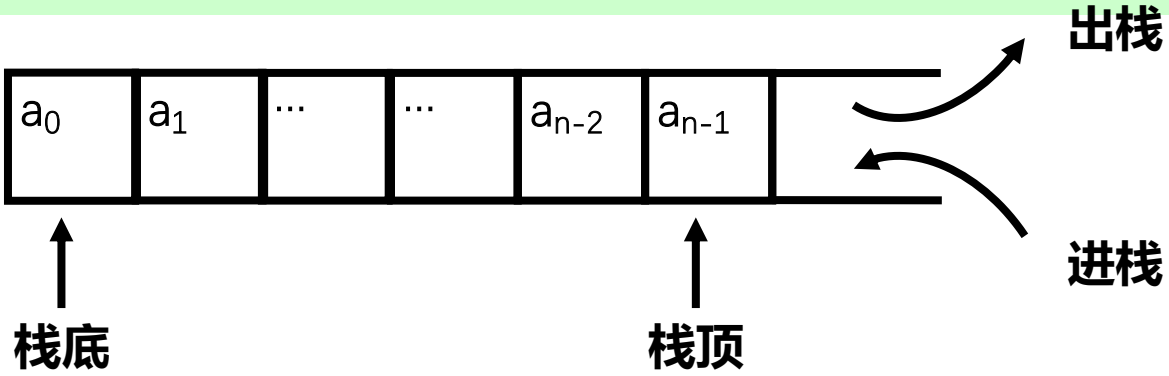
- **后进先出(LIFO, Last In First Out)或先进后出(FILO, First In Last Out)结构，最先(晚)到达栈的结点将最晚(先)被删除。**



概念与运算

- 栈底(bottom) ：结点最早插入的位置
- 栈顶（top）：结点最晚插入的位置
- 出栈（Pop）：结点从栈顶删除
- 进栈（Push）：结点在栈顶位置插入
- 空栈 ：栈中结点个数为零时

- 创建一个栈create()：创建一个空的栈；
- 进栈push(x)：将x插入栈中，使之成为栈顶元素；
- 出栈pop()：删除栈顶元素并返回栈顶元素值；
- 读栈顶元素top()：返回栈顶元素值但不删除栈顶元素；
- 判栈空isEmpty()：若栈为空，返回true，否则返回false。

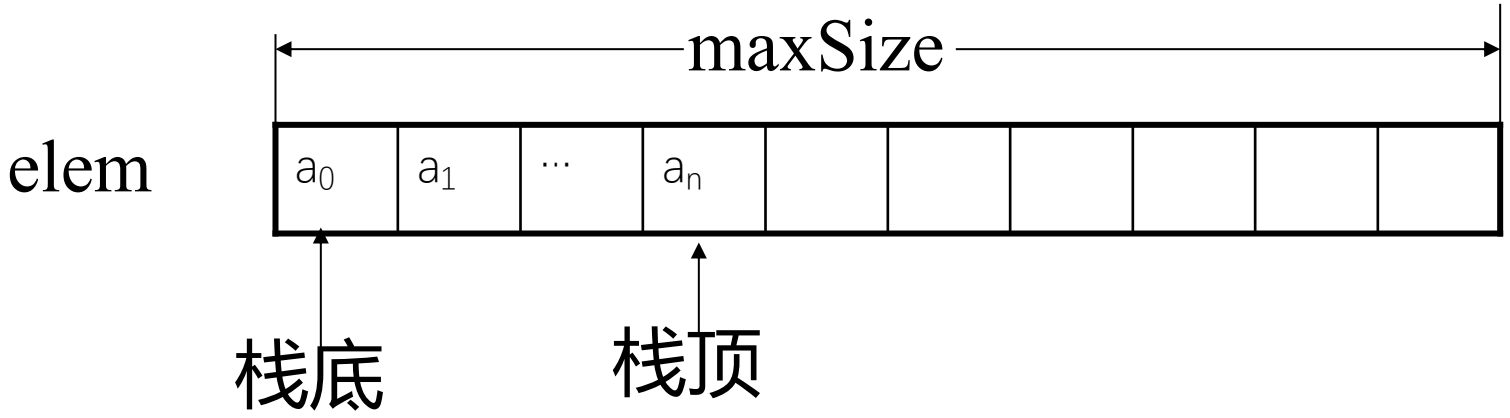


栈的抽象类

```
template <class elemType>
class stack
 { public:
        virtual bool isEmpty() const = 0;
        virtual void push(const elemType &x) = 0;
        virtual elemType  pop() = 0;
        virtual elemType  top() const = 0;
        virtual ~stack() {}
 };
```

栈的顺序实现

- **用连续的空间存储栈中的结点**
- **由于进栈和出栈总是在栈顶一端进行，因此，不会引起类似顺序表中的大量数据的移动。**



顺序栈类

```
template <class elemType>
class seqStack: public stack<elemType>
 { private:
        elemType *elem;
         int top_p;//栈顶
         int maxSize;
         void doubleSpace();
```

```
public:
    seqStack(int initSize = 10) ;
    ~seqStack();
     bool isEmpty() const;
     void push(const elemType &x) ;
     elemType  pop();
     elemType  top() const;
};
```

```
template <class elemType>
seqStack<elemType>::seqStack(int initSize){
      elem = new elemType[initSize];
       maxSize = initSize ;    top_p = -1;
}

template <class elemType>
seqStack<elemType>:: ~seqStack()
  {     delete [] elem;     }

 template <class elemType>
bool seqStack<elemType>:: isEmpty() const
  {    return top_p == -1;    }
```

```
template <class elemType>
void seqStack<elemType>::push(const elemType &x)
{      if (top_p == maxSize - 1)   doubleSpace();
       elem[++top_p] = x;
}

template <class elemType>
elemType seqStack<elemType>::pop()
 {     return elem[top_p--];  }

template <class elemType>
elemType seqStack<elemType>::top() const
 {     return elem[top_p];  }
```

doubleSpace


```
template <class elemType>
void seqStack<elemType>::doubleSpace(){
    elemType *tmp = elem;

    elem = new elemType[2 * maxSize];
    for (int i = 0; i < maxSize; ++i)
        elem[i] = tmp[i];
    maxSize *= 2;
    delete [] tmp;
}
```

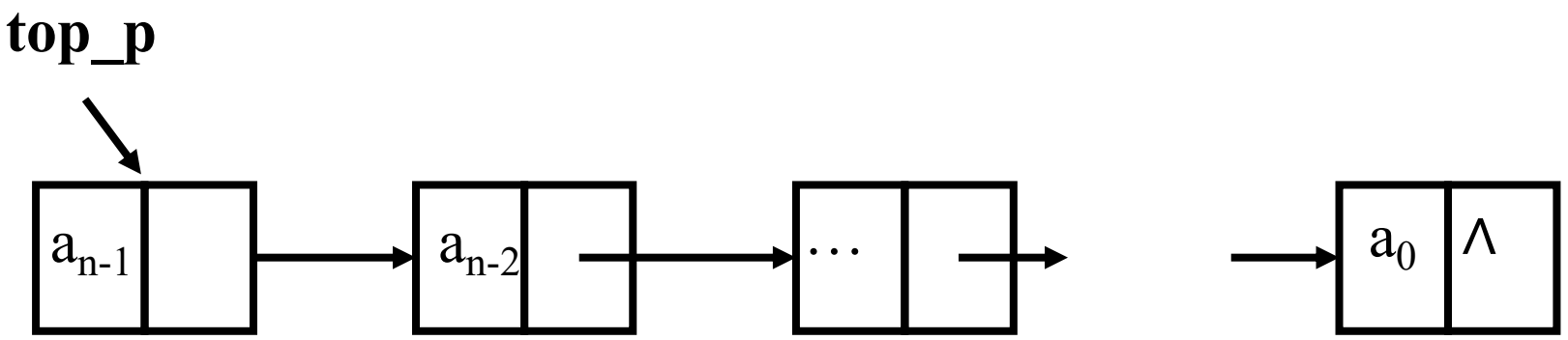
顺序栈操作性能分析

- **除了进栈操作以外，所有运算实现的时间复杂度都是O(1)。**
- **进栈运算在最坏情况下的时间复杂度是O(N)。** 原因？
- **但最坏情况在N次进栈操作中至多出现一次。如果把扩展数组规模所需的时间均摊到每个插入操作，每个插入只多了一个拷贝操作，因此从平均的意义上讲，插入运算还是常量的时间复杂度。这种分析方法称为均摊分析法。**

copy

栈的链接实现

- **栈的操作都是在栈顶进行的，因此不需要双链表，用单链表就足够了，而且不需要头结点**
- **对栈来讲，只需要考虑栈顶元素的插入删除。从栈的基本运算的实现方便性考虑，可将单链表的头指针指向栈顶。**



链式栈类

```
template <class elemType>
class linkStack: public stack<elemType>
 { private:
        struct node {
               elemType  data;
               node *next;
               node(const elemType &x, node *N = NULL)
                        {data = x; next = N;}
               node():next(NULL) {}
               ~node() {}
             };
         node *top_p;
```

```
public:
    linkStack() ;
    ~linkStack();
    bool isEmpty() const ;
    void push(const elemType &x) ;  //引用传递，有时可省空间
    elemType  pop();
    elemType top() const ;
};
```

链式栈的运算实现

- **构造函数：将top_p设为空指针。**

```
template <class elemType>
linkStack<elemType>::linkStack()
{
      top_p = NULL;
}
```

复杂度？

析构函数

```
template <class elemType>
linkStack<elemType>::~linkStack()
{
    node *tmp;
    while (top_p != NULL) {
        tmp = top_p;
        top_p = top_p ->next;
        delete tmp;}
}
```

复杂度？

Push函数

```
template <class elemType>
void linkStack<elemType>::push(const elemType &x)
{
        top_p = new node(x, top_p);
}
```

复杂度？

在表头插入

Pop函数

```
template <class elemType>
elemType linkStack<elemType>::pop()
{
    node *tmp = top_p;
    elemType x = tmp->data;

    top_p = top_p ->next;
    delete tmp;
    return x;
}
```

复杂度？

删除表头元素

Top函数

```
template <class elemType>
elemType linkStack<elemType>::top() const
{
    return top_p ->data;
}
```

复杂度？

返回top_p指向的结点的值

isEmpty函数

```
template <class elemType>
bool linkStack<elemType>::isEmpty() const
{
    return top_p == NULL;
}
```

判top_p是否为空指针

总结：链式栈的性能分析

由于所有的操作都是对栈顶的操作，与栈中的元素个数无关。所以，所有运算的时间复杂度都是O（1）。

链式栈的销毁操作时间复杂度O(n)。

栈的应用：括号配对检查

- 编译程序的任务之一，就是检查括号是否配对。如：括号(、[和 { 后面必须依次跟随相应的 }、] 及)，“后面必须有”。
- 简单地用开括号和闭括号的数量是否相等来判断开括号与闭括号是否配对是不行的。例如，符号串[( )]是正确的，而符号串([ )]是不正确的。因为当遇到) 那样的闭括号时，它与最近遇到开括号匹配。

算法描述

1. 首先创建一个空栈。
2. 从源程序中读入符号。
3. 如果读入的符号是开符号，那末就将其进栈。
4. 如果读入的符号是一个闭符号但栈是空的，出错。否则，将栈中的符号出栈。
5. 如果出栈的符号和和读入的闭符号不匹配，出错。
6. 继续从文件中读入下一个符号，非空则转向3，否则执行7。
7. 如果栈非空，报告出错，否则括号配对成功。

算法描述

初始化栈为空；
While (lastChar = 读文件，直到读入一括号)
  Switch (lastChar)
    {case ‘{‘, ‘[‘, ‘(‘：进栈
  case ‘}’,’]’,’)’:
        if (栈空）
            输出某行某符号不匹配；出错数加1;
                else { match = 出栈的符号；
            检查lastChar与match是否匹配；
            如不匹配，输出出错信息，出错数加1；
                }
    }
if (栈非空) 栈中元素均没有找到匹配的闭符号，输出这些错误
return 出错数

算法逻辑简单，程序实现取决于代码能力。自测一下。

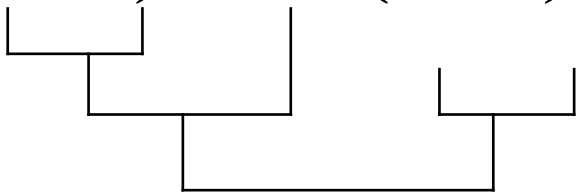
栈应用：简单计算器的实现

输入一个中缀表达式,计算的对象为类型为int的正整数，能计算加、减、乘、除和乘方运算，允许用括号改变优先级。

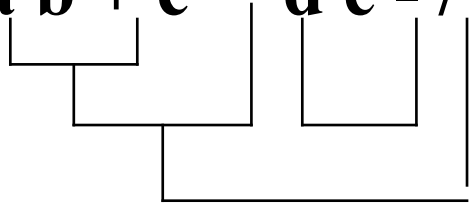
- **对于一个表达式 a + b**
  前缀式：+ab　波兰式
  中缀式：a+b
  后缀式：ab+　逆波兰式

后缀式的优点

- 可以不考虑运算符的优先级
- 例如： (a + b) * c / (d - e)



- 后缀式为： a b + c * d e - /



以表达式5*(7-2*3)+8/2为例，
它的后缀式为5 7 2 3*-*8 2/+

步骤	剩余后缀式	栈中内容	步骤	剩余后缀式	栈中内容
1	5 7 2 3*-*8 2/+		10	/+	5 8 2
2	7 2 3*-*8 2/+	5	11	+	5 4
3	2 3*-*8 2/+	5 7	12		9
4	3*-*8 2/+	5 7 2	13		
5	*-*8 2/+	5 7 2 3	14		
6	-*8 2/+	5 7 6	15		
7	*8 2/+	5 1	16		
8	8 2/+	5	17		
9	2/+	5 8	18		

中缀式转换为后缀式算法运算过程

序号	读剩的表达式	栈	输出
1	(5 + 6 * (7 + 3) / 3) / 4 + 5		
2	5 + 6 * (7 + 3) / 3) / 4 + 5	(	
3	+ 6 * (7 + 3) / 3) / 4 + 5	(	5
4	6 * (7 + 3) / 3) / 4 + 5	(+	5
5	* (7 + 3) / 3) / 4 + 5	(+	5 6
6	(7 + 3) / 3) / 4 + 5	(+*	5 6
7	7 + 3) / 3) / 4 + 5	(+*(	5 6
8	+ 3) / 3) / 4 + 5	(+*(	5 6 7
9	3) / 3) / 4 + 5	(+*(+	5 6 7
10	) / 3) / 4 + 5	(+*(+	5 6 7 3
11	/ 3) / 4 + 5	(+*	5 6 7 3 +
12	3) / 4 + 5	(+/	5 6 7 3 + *
13	) / 4 + 5	(+/	5 6 7 3 + *3
14	/ 4 + 5		5 6 7 3 + * 3 / +
15	4 + 5	/	5 6 7 3 + * 3 / +
16	+ 5	/	5 6 7 3 + * 3 / + 4
17	5	+	5 6 7 3 + * 3 / + 4 /
18		+	5 6 7 3 + * 3 / + 4 / 5
19			5 6 7 3 + * 3 / + 4 / 5 +

	表达式运算过程	运算符栈	运算数栈
1	(5 + 6 * (7 + 3) / 3) / 4 + 5		
2	5 + 6 * (7 + 3) / 3) / 4 + 5	(	
3	+ 6 * (7 + 3) / 3) / 4 + 5	(	5
4	6 * (7 + 3) / 3) / 4 + 5	(+	5
5	* (7 + 3) / 3) / 4 + 5	(+	5 6
6	(7 + 3) / 3) / 4 + 5	(+*	5 6
7	7 + 3) / 3) / 4 + 5	(+*(	5 6
8	+ 3) / 3) / 4 + 5	(+*(	5 6 7
9	3) / 3) / 4 + 5	(+*(+	5 6 7
10	) / 3) / 4 + 5	(+*(+	5 6 7 3
11	) / 3) / 4 + 5	(+*(	5 6 10
12	/ 3) / 4 + 5	(+	5 60
13	3) / 4 + 5	(+/	5 60
14	) / 4 + 5	(+/	5 60 3
15	/ 4 + 5	(+	5 20
16	/ 4 + 5		25
17	4 + 5	/	25
18	+ 5	/	25 4
19	5	+	6
20		+	6 5
21			11

运算过程需要用到两个栈：运算符栈、运算数栈。

calc类的定义

```
class calc{
        char *expression;
        enum token {OPAREN, ADD, SUB, MULTI, DIV,
                            EXP, CPAREN, VALUE, EOL}; //枚举类型
        void BinaryOp( token op,  seqStack<int> &dataStack );
        token getOp( int &value );//引用参数及返回值
    public:
        calc(char *s)
            { expression = new char[ strlen(s) + 1];
              strcpy( expression, s ); }
        ~calc( ) { delete expression; }
        calc &operator=(const calc &other);
        int result( );
};
```

算法思路

```
int calc::result()
{ 依次从表达式中取出一个合法的符号，直到表达式结束；
    { switch(当前符号)
        {
          case  数字：将数字存入运算数栈；
          case ‘(’: 开括号进运算符栈;
          case ‘)’: 运算符栈中的运算符依次出栈，执行相应的运算，
  直到 ‘(’ 出栈；
          case ‘^’:乘方运算符进栈；右结合，其余运算左结合
          case ‘*’: case ‘/’: 运算符栈中的/，*，^退栈执行，当前运
  算符进栈；
          case ‘+’:case ‘-’: 运算符栈中的运算符依次出栈执行，直
  到栈为空或遇到 ‘(’ 括号。当前运算符进栈；
        }
    }
……
}
```

核心：运算优先级管理
同级别运算和不同级别运算优先级
优先级通过代码逻辑体现(本教材方法）还是数据存储（矩阵）？
你喜欢哪种？

进一步细化

- 在上述伪代码中，大多数的操作都是进栈出栈，这些操作在栈类中都已实现。
- 除此之外，还有两个操作需要细化：
  - 从表达式中取出一个合法的符号

    token getOp(int &value);
  - 执行一个算术运算

    void BinaryOp(token op, seqStack<int> &dataStack)；

result函数的实现


```
int calc::result()
{ token lastOp, topOp;
  int result_value, CurrentValue;
  seqStack<token> opStack;
  seqStack<int> dataStack;
  char *str = expression;
```

```
while (true){
    lastOp = getOp(CurrentValue);
    if (lastOp == EOL) break;
    switch (lastOp){
      case VALUE: dataStack.push(CurrentValue) ; break;//操作数入栈
      case CPAREN:    //处理")"情况
        while( !opStack.isEmpty() && (topOp = opStack.pop())  != OPAREN )
//OPAREN代表"("
            BinaryOp(topOp, dataStack);
        if ( topOp  != OPAREN）cerr << "缺左括号!" << endl;  //错误信息输出流
            break;
      case OPAREN: opStack.push(OPAREN); break;//左括号入栈
      case EXP: opStack.push(EXP); break;//乘方入栈 右结合，其余运算左结合
      case MULTI:case DIV:
        while ( !opStack.isEmpty() && opStack.top() >= MULTI) //枚举特殊处理
            BinaryOp(opStack.pop(), dataStack);
          opStack.push(lastOp);          break;
      case ADD:case SUB:
        while ( !opStack.isEmpty() && opStack.top() != OPAREN )
            BinaryOp(opStack.pop(), dataStack );
          opStack.push(lastOp);
        }
}
```

result函数的实现

result函数的实现

```
while (!opStack.isEmpty())
      BinaryOp(opStack.pop(),dataStack);
if (dataStack.isEmpty())
   { cout  << "无结果\n";   return 0; }
result_value = dataStack.pop();
if (!dataStack.isEmpty())
   { cout  << "缺操作符";   return 0; }
expression = str;
return result_value ;
}
```

BinaryOp函数的实现

```
void calc::BinaryOp( token op,  seqStack<int> &dataStack)
{ int num1, num2;
   if ( dataStack.isEmpty())
          { cerr << "缺操作数! ";  exit(1); }
    else num2 = dataStack.pop();
    if ( dataStack.isEmpty())
           { cerr << "缺操作数! "; exit(1);}
    else num1 = dataStack.pop();
switch(op) {
     case ADD: dataStack.push(num1 + num2);
                      break;
     case SUB: dataStack.push(num1 - num2);
                      break;
     case MULTI: dataStack.push(num1 * num2);
                      break;
     case DIV: dataStack.push(num1 / num2);
                      break;
     case EXP: dataStack.push(pow(num1,num2));
    }//计算结果进栈
}
```

教材getOp函数的功能

- 依次扫描表达式expression直到表达式结束，每次取一个符号。
- 很多程序员在写算术表达式时都习惯在运算符的前后插入一些空格，使表达式看上去更加清晰。这些空格对表达式的计算是没有意义的，在扫描过程中要忽略这些空格。
- 当遇到了一个有意义的语法单位时，需要判断是否遇到的是运算数，如果是运算数，则，转换成整型数存入参数value，返回符号VALUE。如果不是运算数，则根据不同的运算符返回不同的token类型的值。

```
calc::token calc::getOp(int &value)
{
  while (*expression)
    { while( *expression  && *expression == ' ')
          ++expression ;                                    //忽略空格

      if ( *expression <= '9' && *expression >= '0')
        { value = 0;
          while (*expression >='0' && *expression <='9')
              { value = value * 10 + *expression - '0';
                ++expression;
              }
          return VALUE;//读入操作数
        }
      switch (*expression){//读入运算符
        case '(': ++expression ; return OPAREN;
        case ')': ++expression ; return CPAREN;
        case '+': ++expression ; return ADD;
        case '-': ++expression ; return SUB;
        case '*': ++expression ; return MULTI;
        case '/': ++expression ; return DIV;
        case '^': ++expression ; return EXP;
       }
    }
 return EOL;
}
```

练习:合理利用助教，答疑是改Bug的进阶版？

- 本章书面练习不需要交，供期末（完全不保真）参考。若有疑问，可答疑。
- 简答题1、2、4、5
- 程序设计题：1（可重点考虑进栈）、4

- **布尔表达式求值算法设计与实现。（上机题目，算平时分数）**
- ## 输入

一行字符串，代表一个布尔表达式

输出

一个bool值 (1或0)，表示布尔表达式的结果为true或false

- 详细说明参见canvas

总结：本章学习目标

1.利用栈进行问题求解。以递归改非递归以及表达式计算为例。

2.对顺序栈和链栈实现创建、进栈、出栈以及销毁操作。

考核重点：进出栈可能序列；栈操作代码；表达式计算

Thanks! & QA