数据结构与算法设计

2021-09

课程内容简介

第1章 绪论	第8章 排序与分治	串与串匹配算法
第2章 线性表	第9章 外部排序	红黑树
第3章 栈和队列	第10章 动态规划算法	k-d树
第4章 数组和广义表	第11章 有限自动机	复杂图算法
第5章 树、二叉树、回溯法	第12章 图灵机	文本检索技术
第6章 图与贪心算法	第13章可判定性	分支限界算法
第7章 查找	第14章 时间复杂性	随机化算法
		上下文无关文法

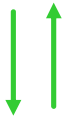
第3章 线性表

3.1 栈的表示与实现
3.2 栈的应用
3.3 队列的表示和实现
3.4 队列应用

3.1.1什么是栈(stack)

栈是仅能在表头或表尾进行插入、删除操作的线性表。

$(a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$



插入

删除

3.1.1什么是栈(stack)

$(a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$

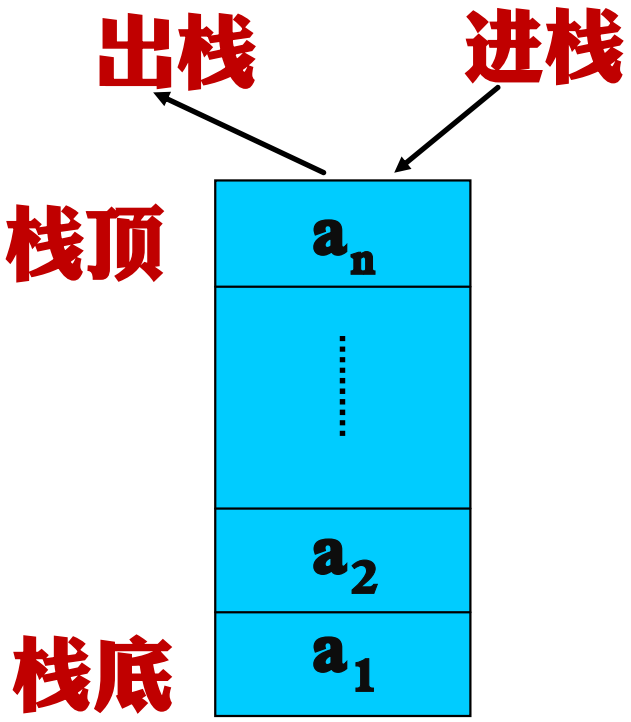
删除 ⇅ 插入

能进行插入和删除的一端称为栈顶(top),另一端称为栈底(bottom)。

称插入操作为进栈(push)，删除操作为出栈(pop)。

栈的特征是后进先出（LIFO， Last_in First_out）

第一个进栈的元素在栈底
最后一个进栈的元素在栈顶
第一个出栈的元素为栈顶元素
最后一个出栈的元素为栈底元素



栈的示意图

3.1.1栈的基本操作-1

1)初始化操作 InitStack(&S)

功能：构造一个空栈S。

2)销毁栈操作 DestroyStack(&S)

功能：销毁一个已存在的栈。

3)置空栈操作 ClearStack(&S)

功能：将栈S置为空栈。

3.1.1栈的基本操作-2

4)取栈顶元素操作 GetTop(S, &e)

功能：取栈顶元素，并用e 返回。

5)进栈操作 Push(&S, e)

功能：元素e进栈。

6)出栈操作 Pop(&S, &e)

功能：栈顶元素出栈，并用e返回。

3.1.1 栈的基本操作-3

7)判空操作 StackEmpty(S)

功能：若栈S为空，则返回True，否则，栈不空返回False

3.1.2 栈的物理实现方式

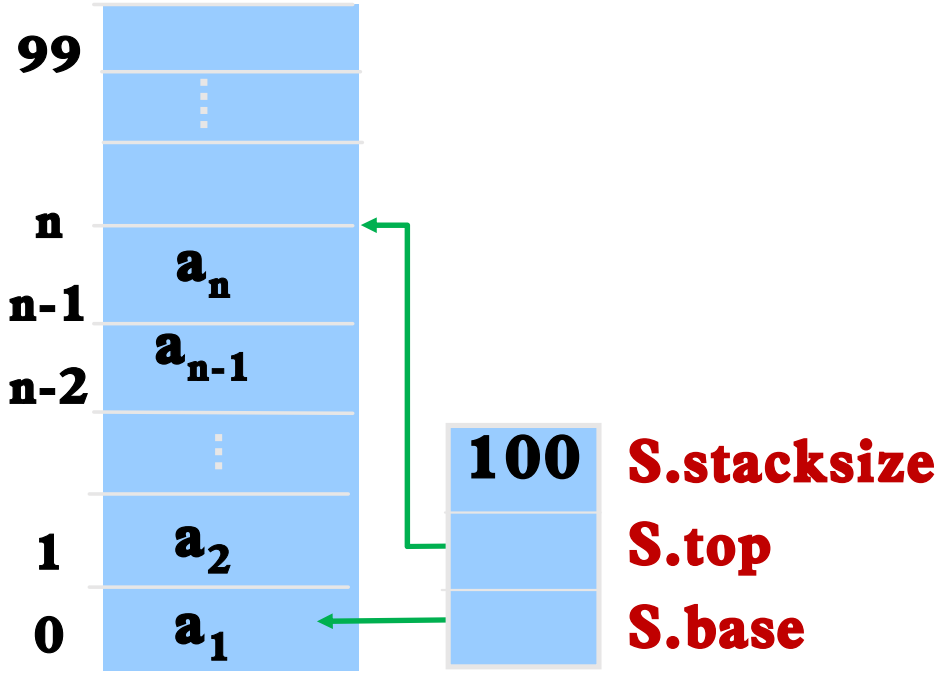
顺序结构（数组）

链表方式

3.1.2 栈的顺序存储结构

```
#define STACK_INIT_SIZE 100
                                          // 栈存储空间的初始分配量
#define STACKINCREMENT 10   // 空间的分配增量
typedef struct
{   ElemType  *base;          //栈空间基址
    ElemType  *top;           //栈顶指针
    int stacksize;          //当前分配的栈空间大小
} SqStack;
```

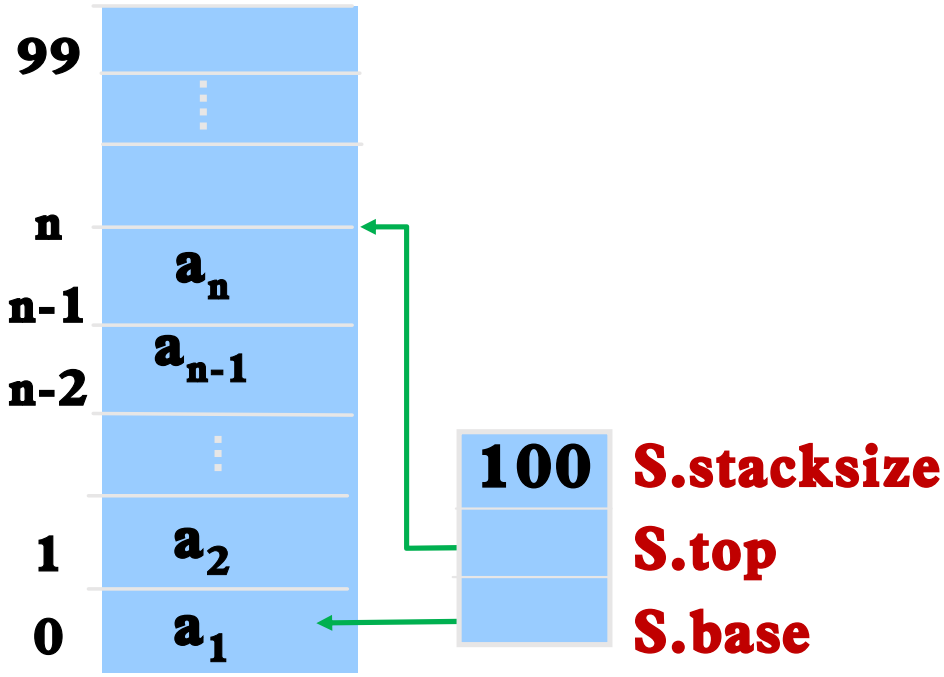
3.1.2 栈的顺序存储和实现



顺序栈的图示

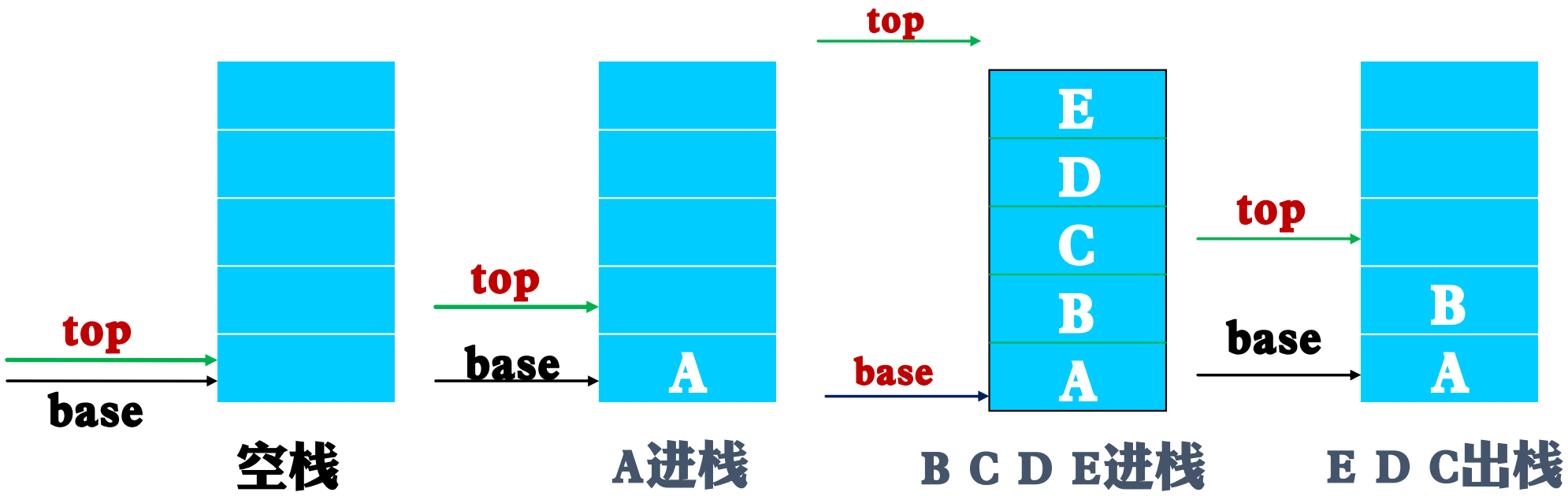
3.1.2 栈的顺序存储和实现

栈顶指针指向哪里?
A）栈顶第一个空闲位置
B）栈顶元素的位置



顺序栈的图示

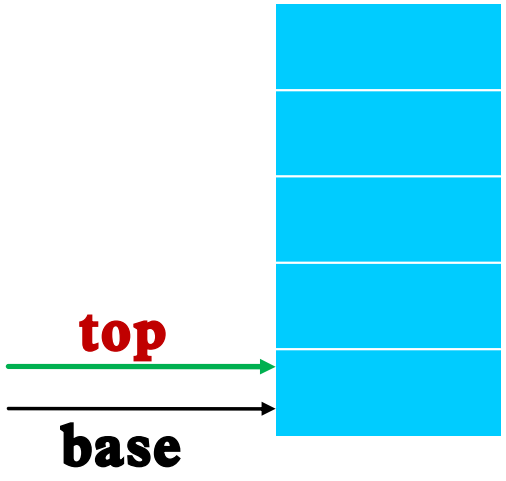
3.1.2 栈的顺序存储和实现



空栈 top = base

栈满 top-base = stacksize (无可分配空间)

栈空



空栈

空栈　top == base

栈顶指针top，指向实际栈顶后的空位置，初值为 base

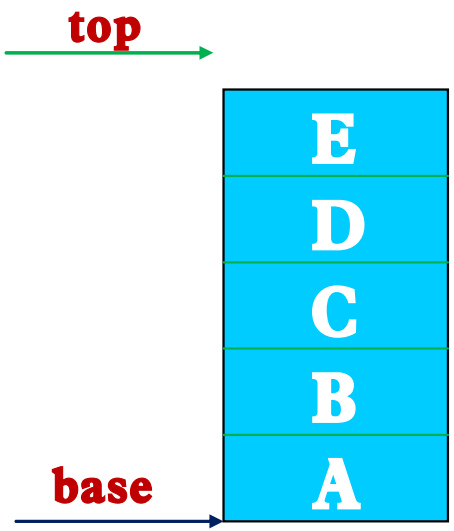
top==base，栈空，此时出栈，则下溢(underflow)

栈满　top-base = stacksize（无可分配空间）

设栈的初始分配量为 Stacksize = STACK_INIT_SIZE。

若 top == Stacksize，栈满，此时入栈，则需扩充栈空间，每次扩充 STACK_INCREMENT；

若无可利用的存储空间，则上溢(overflow)。



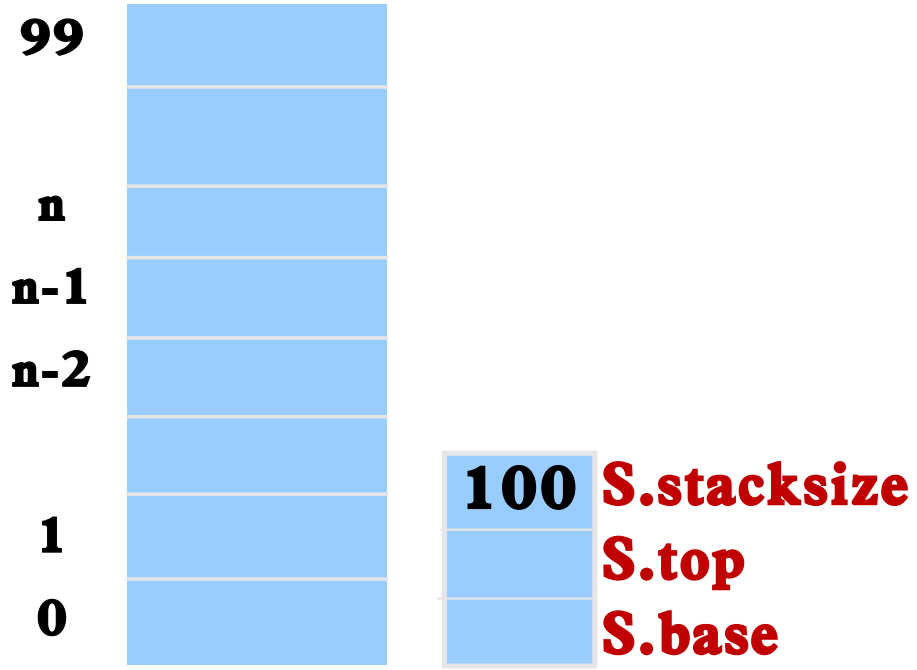
B C D E进栈

二、顺序栈基本操作的算法

1）初始化操作 InitStack（ SqStack &S ）

参数： S是存放栈的结构变量

功能： 建一个空栈S

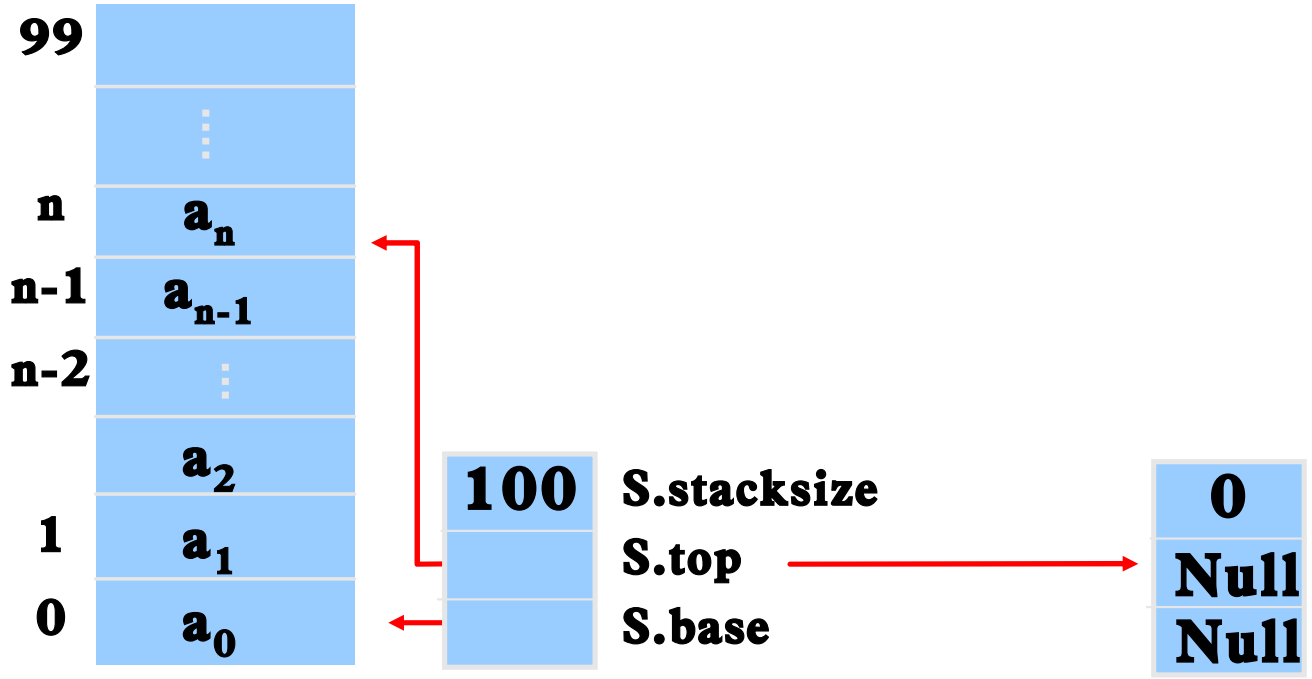


初始化操作 InitStack（ SqStack &S ）

```
Status InitStack（ SqStack  &S ）//构造一个空栈S
{   S.base =（ ElemType *）  malloc
                （ STACK_INIT_SIZE * sizeof(ElemType) );
    //为顺序栈动态分配存储空间
    if（ ! S. base)          exit( OVERFLOW );   //分配失
败
    S.top = S.base;
    S.stacksize = STACK_INIT_SIZE;
    return OK;
}  // InitStack
```

销毁栈操作 DestroyStack(SqStack &S)

功能：销毁一个已存在的栈

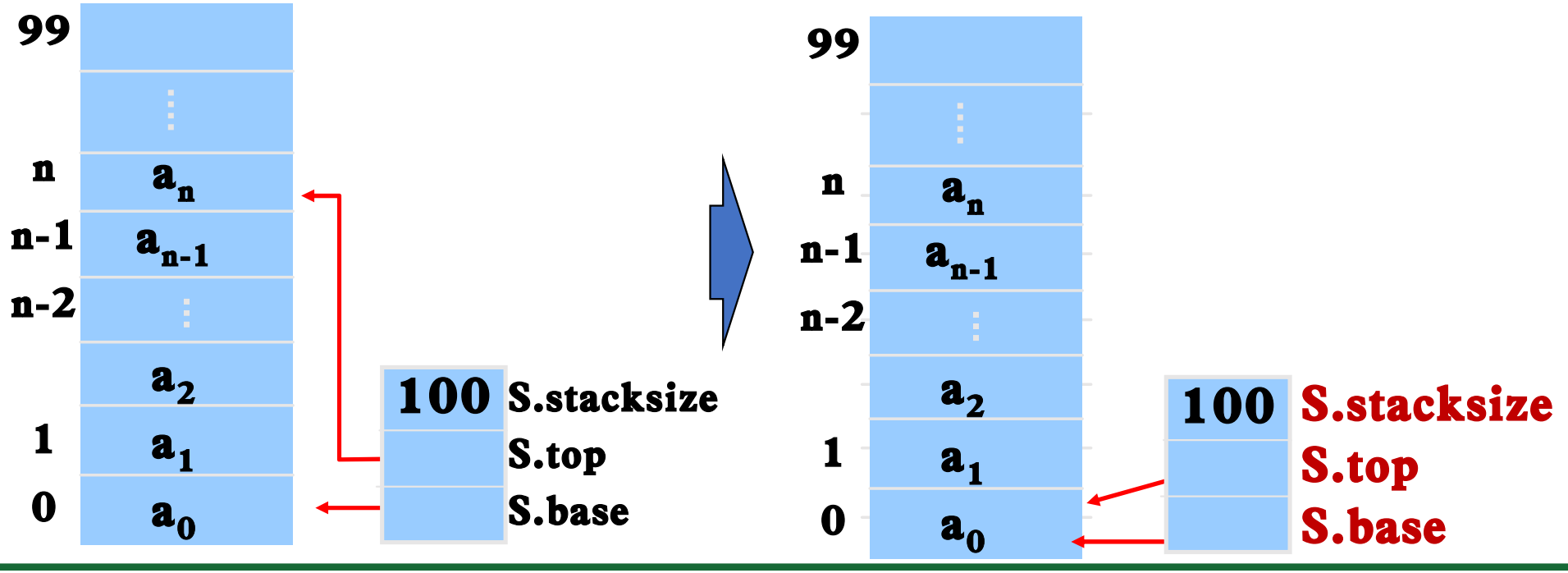


销毁栈操作 DestroyStack(SqStack &S)

```
Status DetroyStack ( SqStack  &S )
 {  if ( ! S.base )
        return ERROR;
              //若栈未建立（尚未分配栈空间）
     free ( S.base );                                  //回收栈空间
     S.base = S.top = NULL;
     S.stacksize = 0;
     return OK;
}  //DetroyStack
```

置空栈操作ClearStack（SqStack　&S）

功能：将栈S置为空栈



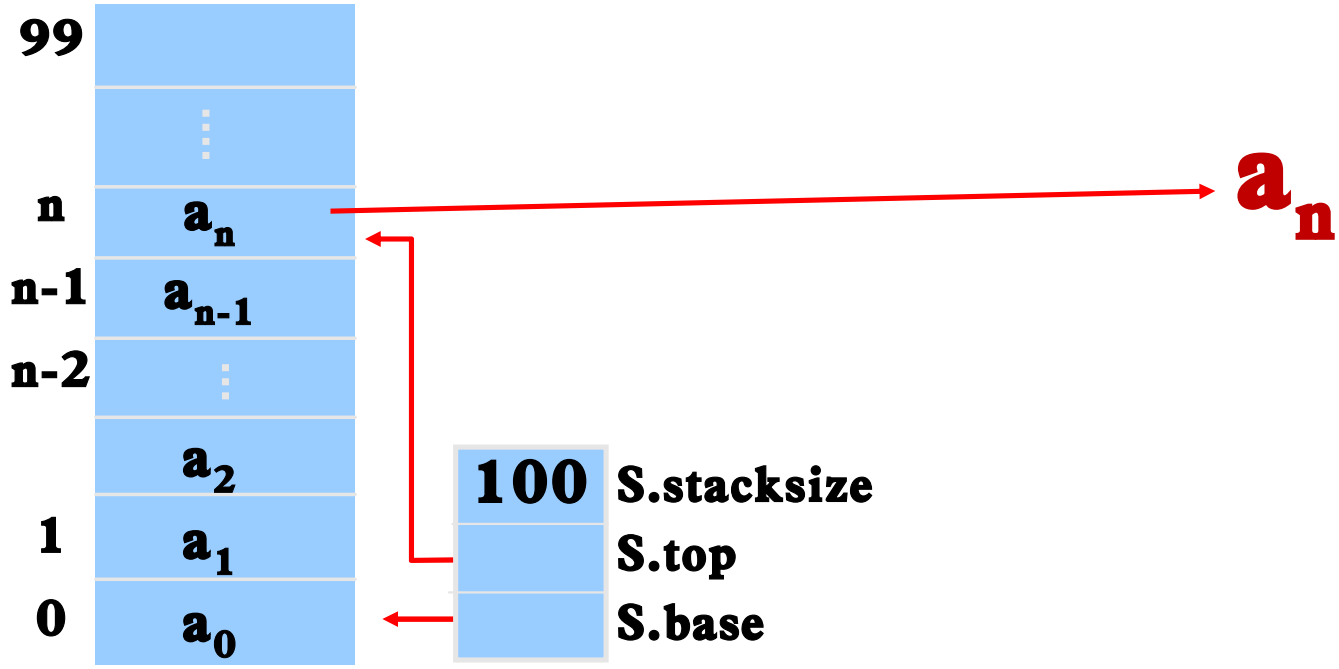
置空栈操作ClearStack（SqStack　&S）

```
Status ClearStack（ SqStack　&S ）
 {  if（ ! S.base ）
          return ERROR;     // 若栈未建立（尚未分配栈空间）
     S.top = S.base;
     return OK;
}  //ClearStack
```

取栈顶元素操作

GetTop（SqStack S，ElemType &e）

功能：取栈顶元素，并用e返回



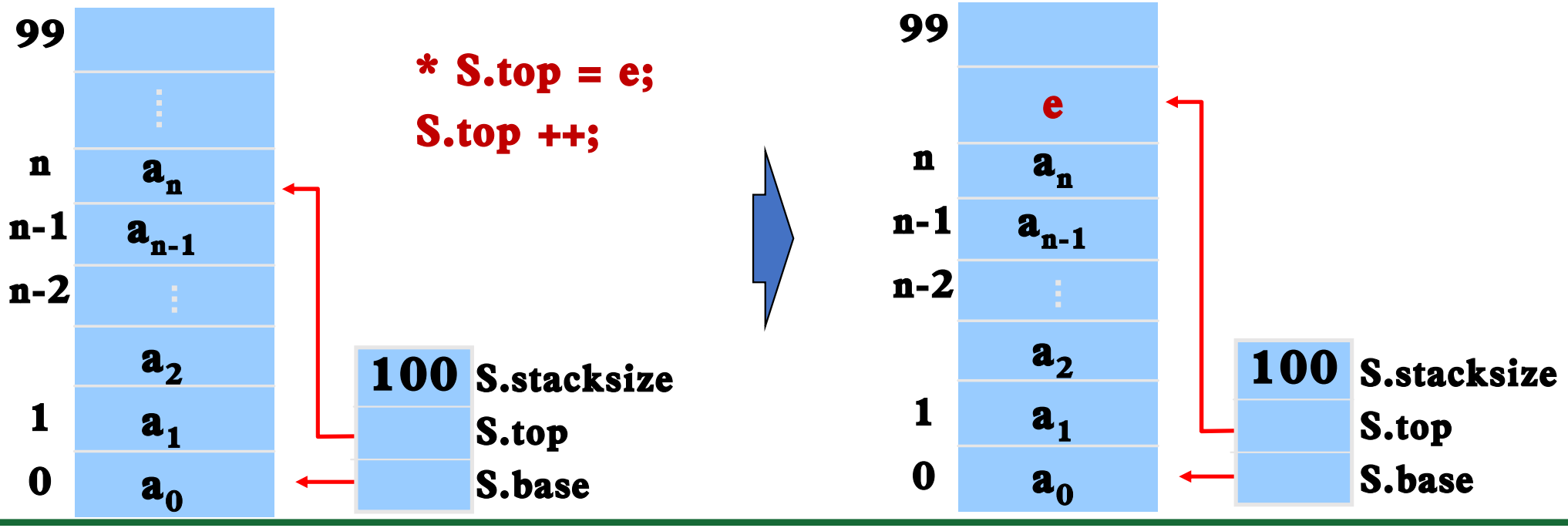
取栈顶元素操作

```
Status GetTop（ SqStack S, ElemType &e ）
{  if（ S.top==S.base ）
          return ERROR;          //栈空
     e = *(S.top-1);
     return OK;
}  //GetTop
```

进栈操作

Push（ SqStack &S, ElemType e ）

功能：元素 e 进栈。



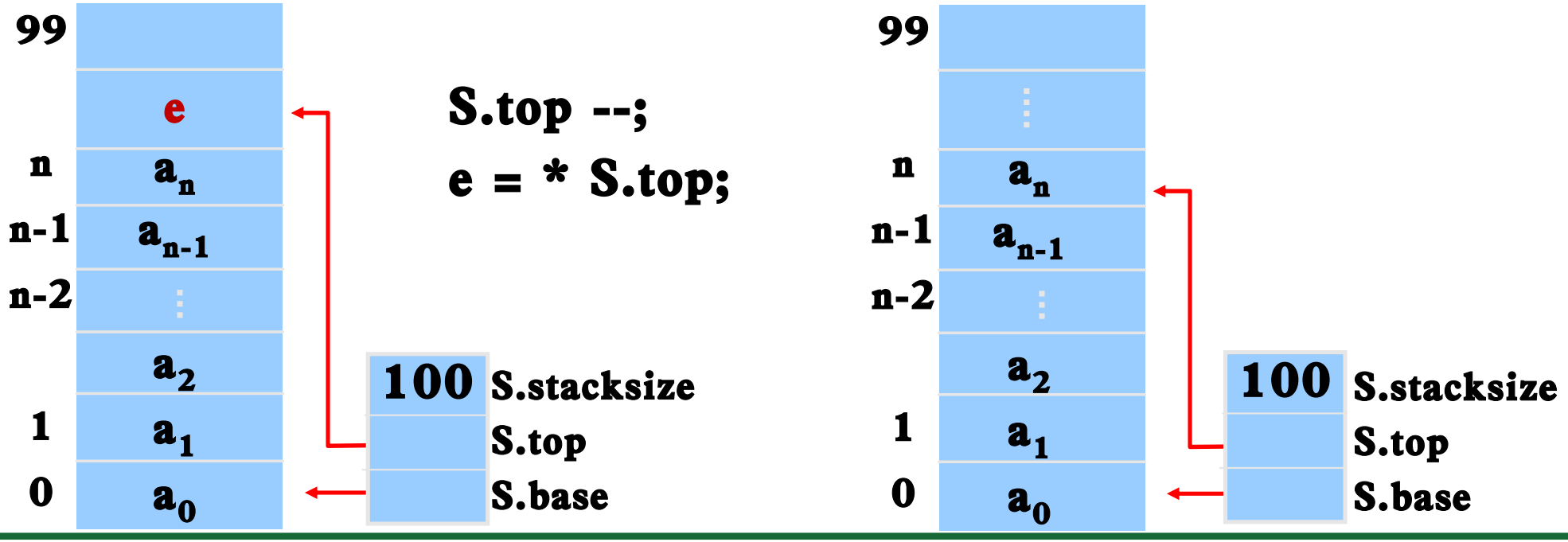
进栈操作

```
Status Push ( SqStack &S, ElemType e )
{   //将元素e插入栈中，使其成为新的栈顶元素
    if ( S.top-S.base>=S.stacksize ) // 若栈满则追加存储空间
    {  S.base = (ElemType * ) realloc ( S.base,
(S.stacksize +STACKINCREMENT) * sizeof(ElemType));
        if ( ! S. base )    exit(OVERFLOW);   //存储分配失败
        S.top = S.base + S.stacksize;
        S.stacksize += STACKINCREMENT;
    }
    * S.top ++ = e;   //元素e 插入栈顶，后修改栈顶指针
    return OK;
}  //Push
```

出栈操作

Pop（ SqStack &S, ElemType &e ）

功能：栈顶元素退栈，并用 e 返回。



出栈操作

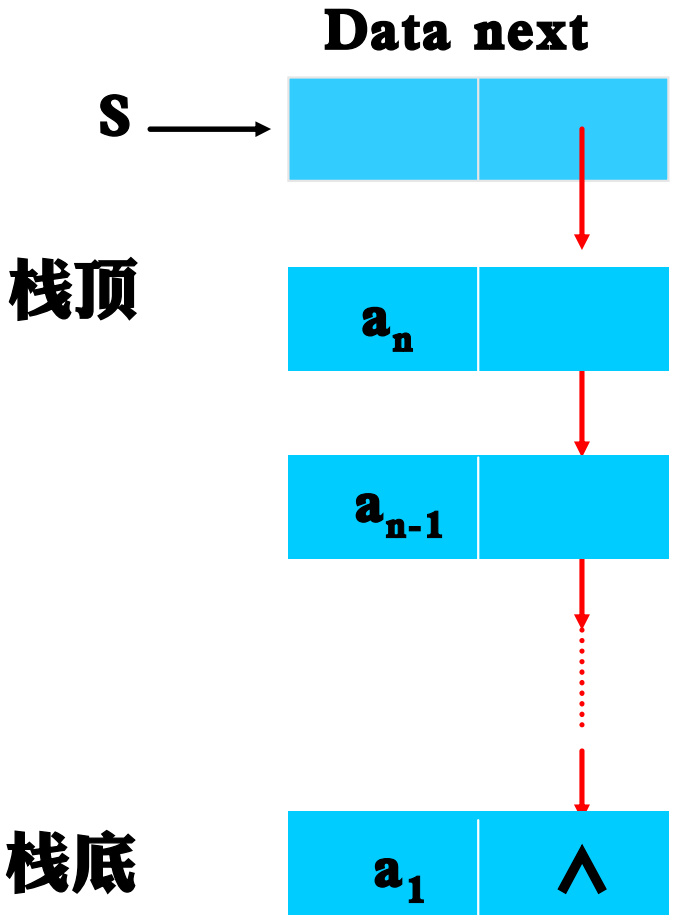
```
Status  Pop（ SqStack &S,  ElemType &e ）
 {  if（ S.top==S.base ）
          return ERROR;     // 栈空 ，下溢
     e = * -- S.top;    // 相当于--S.top; e=*S.top;
     return OK;
}  //Pop
```

另一种约定：

栈顶指针指向栈顶元素。该如何处理?

3.1.3 栈的链式存储和实现

在前面学习了线性链表的插入、删除操作算法，不难写出链式栈初始化、进栈、出栈等操作的算法。



顺序栈和链式栈的比较

栈的特点：后进先出

体现了元素之间的透明性

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O(k)

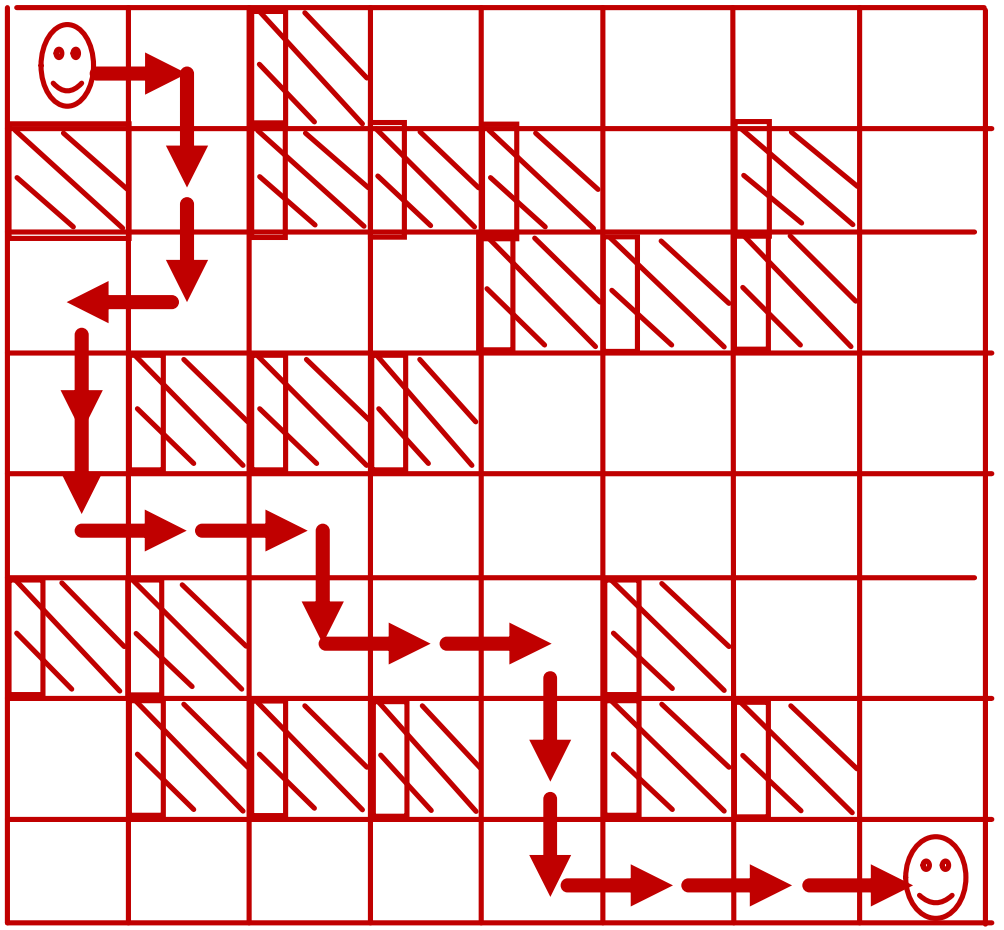
一般来说，栈不允许“读取内部元素”，只能在栈顶操作

3.2 栈的应用

1. 数制转换
2. 括号匹配的检验
3. 行编辑程序
4. 迷宫求解
5. 表达式求值

迷宫求解

入口



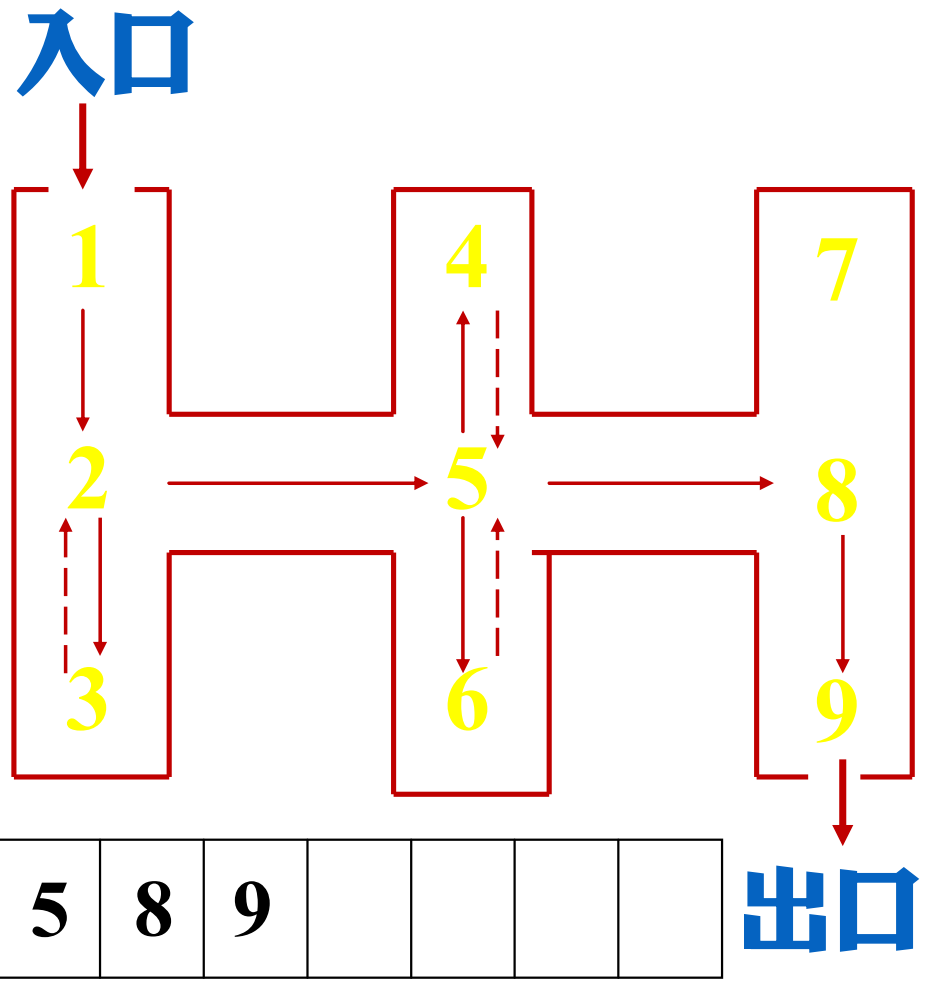
出口

求迷宫路径算法

若当前位置“可通”，则纳入路径，继续前进；

若当前位置“不可通”，后退一个位置，换方向继续探索；

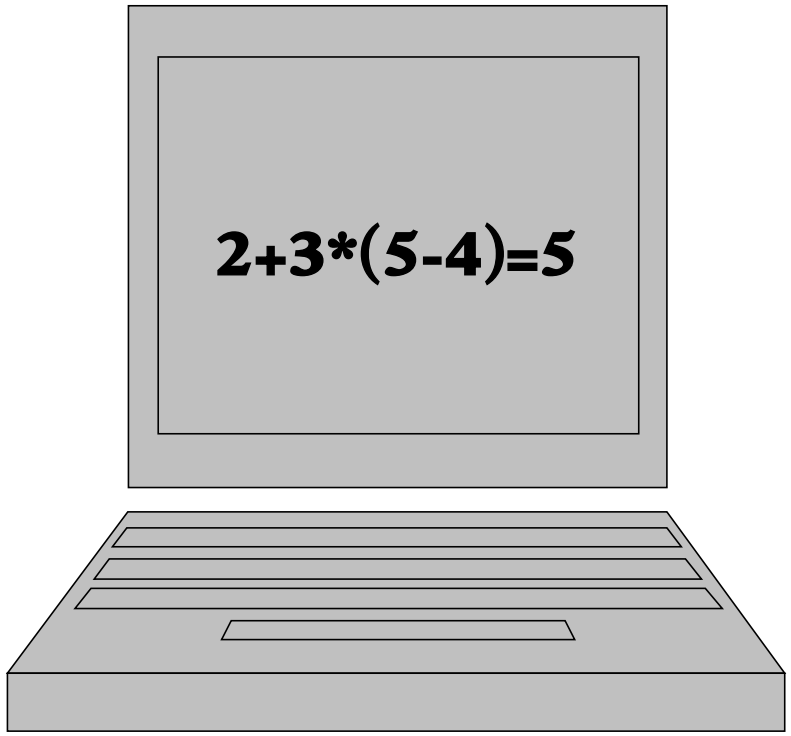
若四周“均已探索”，则后退一步，将当前位置从路径中删除出去，换方向继续探索。



1	2	5	8	9				

栈的应用举例－表达式求值

从键盘一次性输入一串算术表达式，给出计算结果。



表达式求值

操作数

( )、#

2）表达式的构成　操作数+运算符+界符

+、-、*、/

3）表达式的求值：

例：5+6×(1+2)-4

按照四则运算法则，上述表达式的计算过程为：

5+6×(1+2)-4 = 5+6×3-4 = 5+18-4 = 23-4 = 19

1　2　3　4

算符优先关系表

4） 表达式中任何相邻运算符 1、运算符2 的优先关系有：

1 < 2： 1的优先级 低于 2

1 = 2： 1的优先级 等于 2

1 > 2： 1的优先级 高于 2

注：θ_1、θ_2是相邻算符，θ_1在左，θ_2在右

算符优先关系表

表达式中任何相邻运算符

θ_1、θ_2 的优先关系有:

$\theta_1 < \theta_2$: θ_1的优先级 低于 θ_2

$\theta_1 = \theta_2$: θ_1的优先级 等于 θ_2

$\theta_1 > \theta_2$: θ_1的优先级 高于 θ_2

	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	=
#	<	<	<	<	<		=

表达式运算

在算符优先算法中，建立了两个工作栈。一个是OPTR栈，用以保存运算符；一个是OPND栈，用以保存操作数或运算结果。

算法的基本思想是：

1、首先置操作数栈为空栈，表达式起始符“#”为运算符栈的栈底元素。

2、依次读入表达式中每个字符，若是操作数，则进OPND栈；若是运算符，则与OPTR栈的栈顶运算符比较优先级后作相应操作。

直至整个表达式求值完毕（即OPTR栈的栈顶元素和当前读入的字符均为“#”）。

5 + 4 x (1 + 2) - 6

从左向右扫描表达式，用两个栈分别保存扫描过程中遇到的操作数和运算符：

遇操作数——保存；

遇运算符号 j——与前面的刚扫描过的运算符 i比较：

若 i< j 则保存 j（因此已保存的运算符的优先关系）

若 i> j 则说明 i是已扫描的运算符中优先级最高者，可进行运算

若 i= j 则说明括号内的式子已计算完，需要消去括号

后保存的算符优先级高

读入表达式过程：

5+6X(1+2)-4#19

OPTR栈

4
2
1
6
5

OPND栈

#
-
)
+
(
*
+

1+2=3

6×3=18

5+18=23

23-4=19

3.3 栈与递归

递归在数学和程序设计等许多领域中都会用到。

任何一个递归程序都可以通过非递归程序实现。

栈在实现函数递归调用中所发挥的作用

栈的一个最广泛的应用：高级语言运行时环境(runtime)提供的函数调用机制。运行时环境(runtime)指的是目标计算机上用来管理存储器并保存指令执行过程中所需信息的寄存器及存储器的结构。

3.3 栈与递归

函数调用过程

在非递归情况下，数据区的分配可以在程序运行前进行，直到整个程序运行结束才释放，这种分配称为**静态分配**。

采用静态分配时，函数的调用和返回处理比较简单，不需要每次分配和释放被调函数的数据区。

3.3 栈与递归

函数的递归调用过程

在递归调用的情况下，被调函数的局部变量不能静态地分配某些固定单元，而必须每调用一次就分配一份新的局部变量，以存放当前函数所使用的数据，当函数结束返回时随即释放。

故其存储分配只能在执行调用时（程序运行过程中）才能进行，即所谓的动态分配。

在内存中要开辟一个称为运行栈（runtime stack）的足够大的动态区，以处理运行数据。

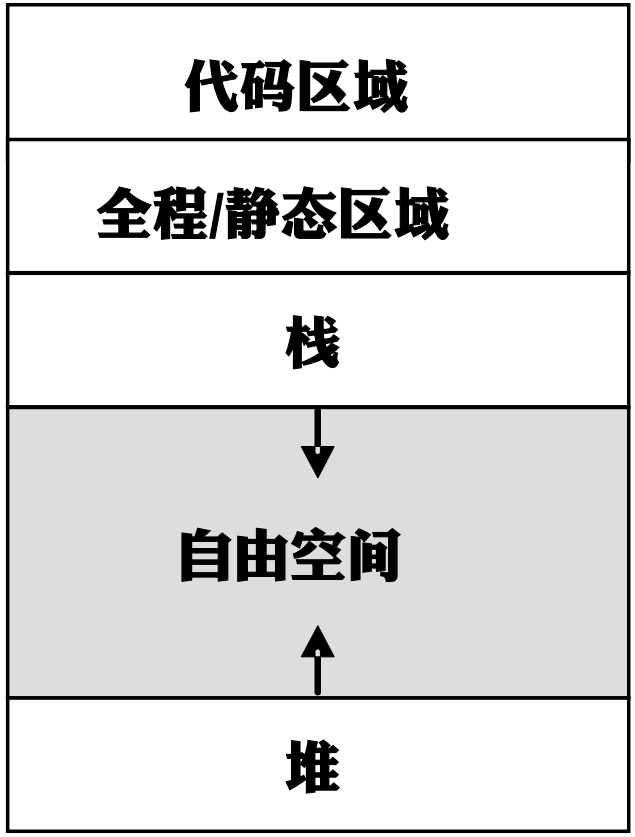
3.3 栈与递归

函数运行时的动态分配过程

用作动态数据分配的存储区可按多种方式组织。典型的组织是将这个存储器分为**栈（stack）**区域和**堆（heap）**区域：

栈区域用于分配发生在后进先出LIFO风格中的数据（诸如函数的调用）。

堆区域则用于不符合LIFO（诸如指针的分配）的动态分配。

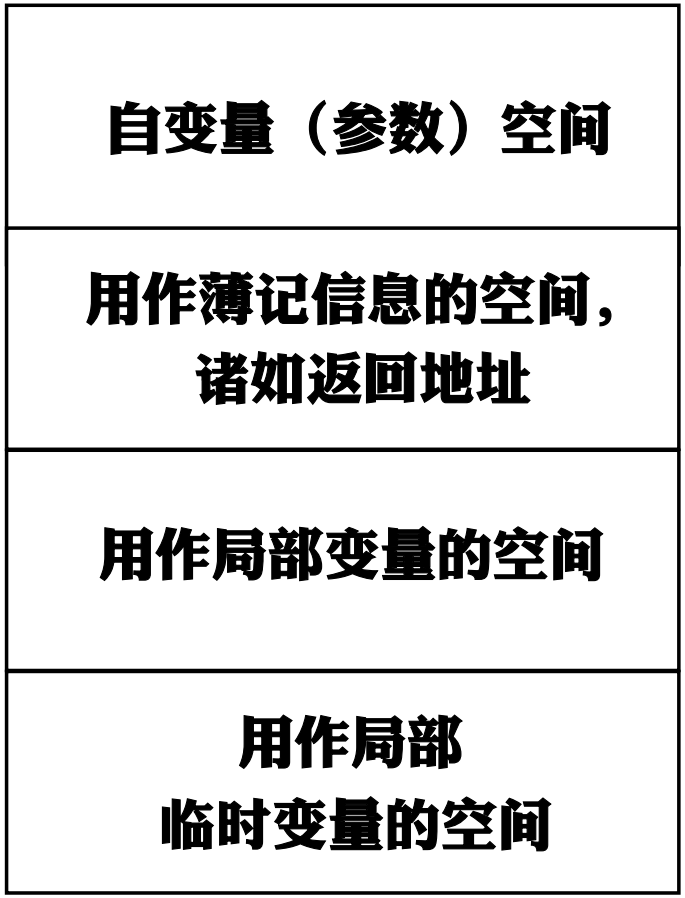


3.3 栈与递归

- **函数活动记录（activation record）**

是动态存储分配中一个重要的单元。

当调用或激活函数时，函数的活动记录包含了为其局部数据分配的存储空间。



3.3 栈与递归

· 运行栈的动态变化

每次调用时，执行进栈操作，把被调函数的活动信息压入栈中，即当进行一个新的函数调用时，都要在栈的顶部为新的活动记录分配空间。

在每次从函数返回时，执行出栈操作，释放本次的活动记录，恢复到上次调用所分配的数据区中。

被调函数中变量地址全部采用相对于栈顶的相对地址来表示。

3.3 栈与递归

一个函数在运行栈上可以有若干不同的活动记录，每个活动记录都代表了一次不同的调用

对于递归函数来说，递归的深度就决定了其在运行栈中活动记录的数目。

当函数递归调用时，函数体的同一个局部变量，在不同的递归层次要分配不同的存储空间，放在内部栈的不同位置。

3.3 栈与递归

函数调用时的三个步骤

调用函数发送调用信息。调用信息包括调用方要传送给被调方的信息，诸如实参、返回地址等。

为被调函数分配需要的局部数据区，用来存放被调方定义的局部变量、形参变量（存放实参）、返回地址等，并接受调用方传送来的调用信息。

调用方暂停，把计算控制转到被调方，即自动转移到被调用的函数的程序入口。

3.3 栈与递归

当被调方结束运行，返回到调用方时，其返回处理也分解为三步进行

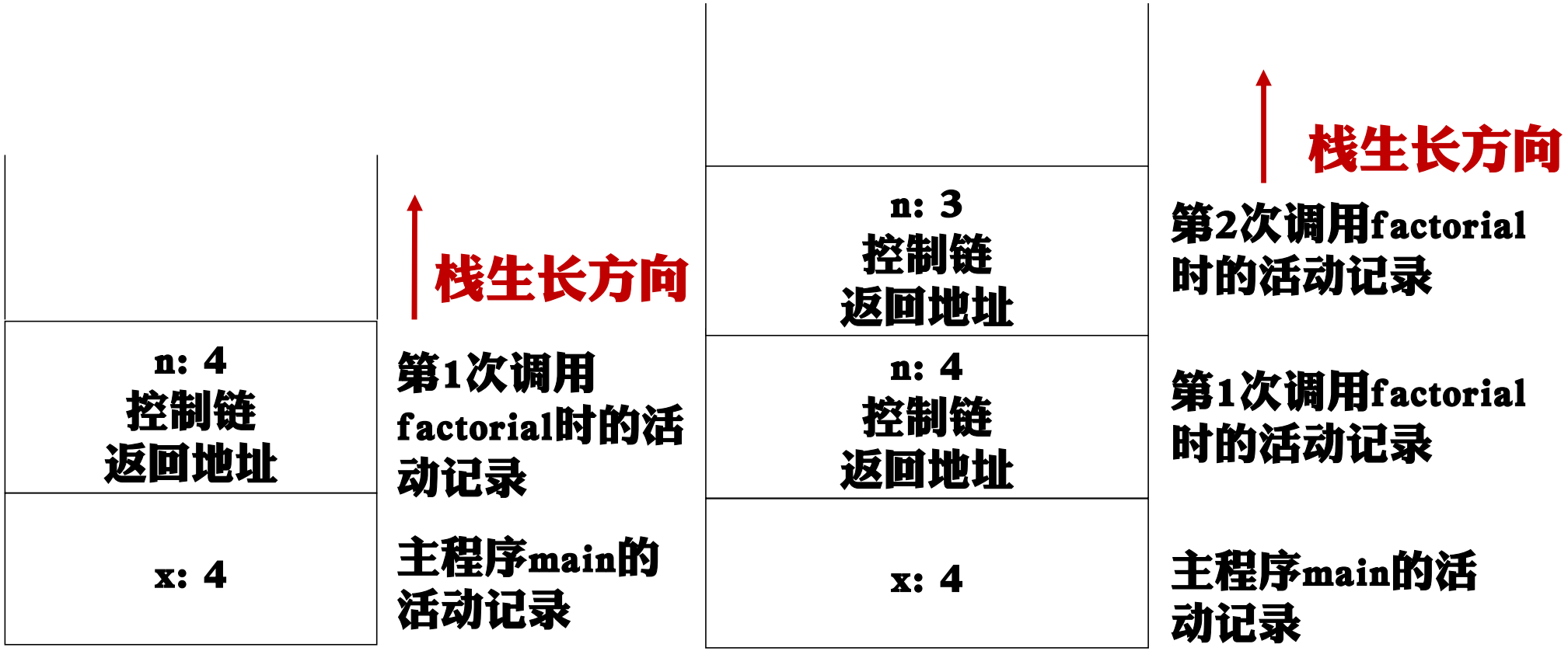
1. **传送返回信息，包括被调方要传回给调用方的信息，诸如计算结果等。**
2. **释放分配给被调方的数据区。**
3. **按返回地址把控制转回调用方。**

3.3 栈与递归

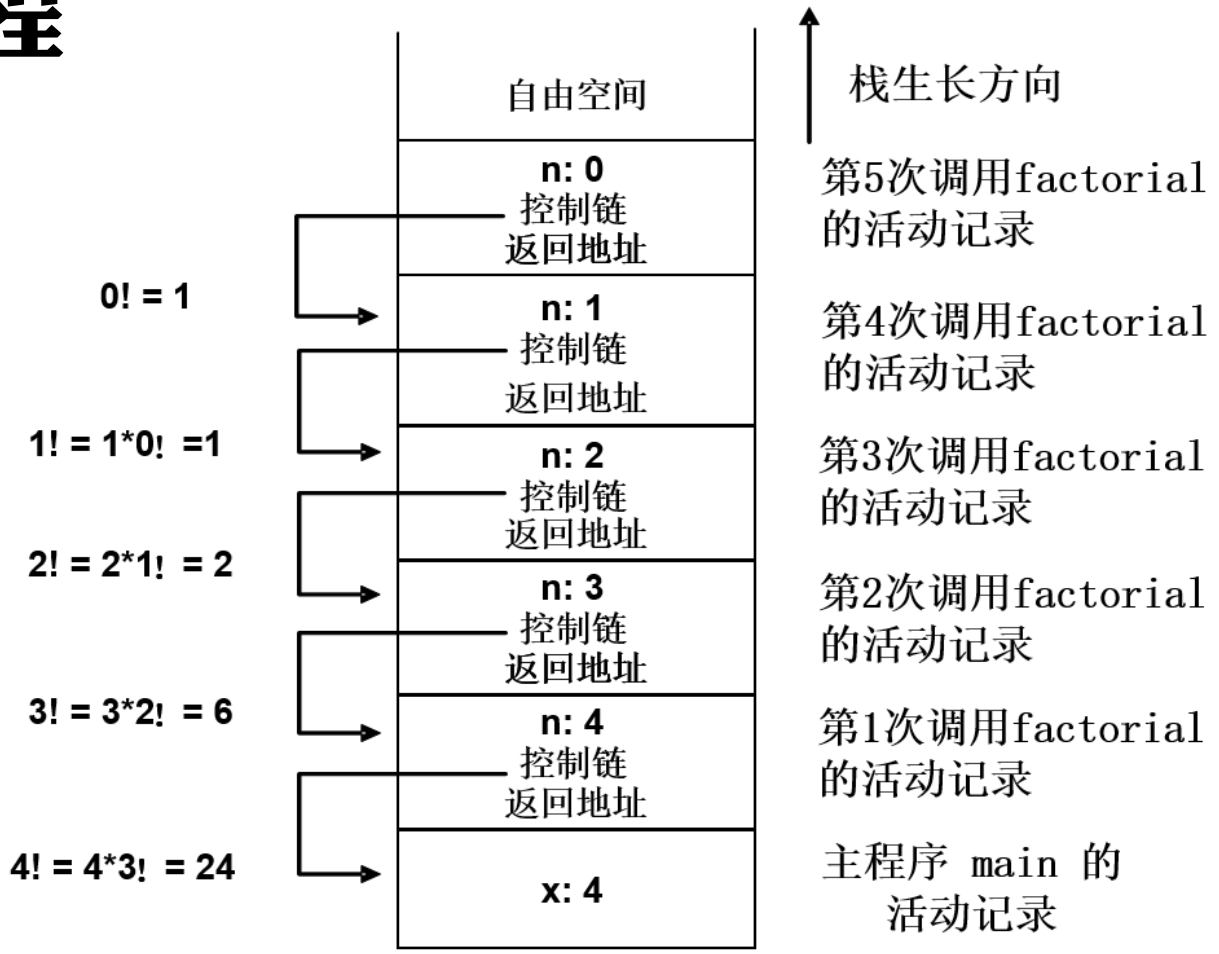
以阶乘为例:

```
#include <stdio.h>
main( ) {   int x;
     scanf(" %d" , &x);
     printf(" %d\n" , factorial(4));
}
long factorial( long n ) {
    if ( n == 0 )
        return 1;
     else
        return n * factorial( n-1);  // 递归调用
}
```

递归函数调用过程


栈生长方向
n: 4
控制链
返回地址
x: 4
第1次调用
factorial时的活
动记录
主程序main的
活动记录
栈生长方向
n: 3
控制链
返回地址
n: 4
控制链
返回地址
x: 4
第2次调用factorial
时的活动记录
第1次调用factorial
时的活动记录
主程序main的活
动记录

递归函数调用过程


自由空间
n: 0
控制链
返回地址
n: 1
控制链
返回地址
n: 2
控制链
返回地址
n: 3
控制链
返回地址
n: 4
控制链
返回地址
x: 4
0! = 1
1! = 1*0! =1
2! = 2*1! = 2
3! = 3*2! = 6
4! = 4*3! = 24
栈生长方向
第5次调用factorial的活动记录
第4次调用factorial的活动记录
第3次调用factorial的活动记录
第2次调用factorial的活动记录
第1次调用factorial的活动记录
主程序 main 的活动记录

栈的应用举例

假定元素进栈的顺序是1234，那么出栈的顺序可否为3142;

若用S表示进栈，X表示出栈，假定元素入栈的顺序是1234，为了得到1342的出栈顺序，请给出对应的SX操作序列。

一个栈的进栈序列为123…N，可以得到出栈序列p1p2…pN，若p1为3，则p2的可能取值的个数有多少个?

一个栈的进栈序列为123…N，则所有可能的出栈的序列有多少个?

栈的应用举例

我们把n个元素的出栈个数的记为f(n), 那么对于1,2,3, 我们很容易得出:

f(1) = 1　　//即 1

f(2) = 2　　//即 12、21

f(3) = 5　　//即 123、132、213、321、231

来考虑f(4), 我们给4个出栈元素编号为a,b,c,d, 那么考虑：元素a只可能出现在1号位置，2号位置，3号位置和4号位置(很容易理解，一共就4个位置，比如abcd,元素a就在1号位置)。

栈的应用举例

1) 如果元素a在1号位置，那么只可能a进栈，马上出栈，此时还剩元素b、c、d等待操作，就是子问题f(3)；

2) 如果元素a在2号位置，那么一定有一个元素比a先出栈，即有f(1)种可能顺序（只能是b），还剩c、d，即f(2)，根据乘法原理，一共的顺序个数为f(1) * f(2)；

3) 如果元素a在3号位置，那么一定有两个元素比1先出栈，即有f(2)种可能顺序（只能是b、c），还剩d，即f(1)，根据乘法原理，一共的顺序个数为f(2) * f(1)；

4) 如果元素a在4号位置，那么一定是a先进栈，最后出栈，那么元素b、c、d的出栈顺序即是此小问题的解，即f(3)；

栈的应用举例

结合所有情况，即f(4) = f(3) + f(2) * f(1) + f(1) * f(2) + f(3);

我们定义f(0) = 1；于是f(4)可以重新写为：

f(4) = f(0)*f(3) + f(1)*f(2) + f(2) * f(1) + f(3)*f(0)

推广到n：f(n) = f(0)*f(n-1) + f(1)*f(n-2) + f(2)*f(n-3)+…… + f(n-1)*f(0)

$$f(n) = \sum_{i=0}^{n-1} f(i)f(n-1-i)$$

栈的应用举例

对于每一个数来说，必须进栈一次、出栈一次。我们把进栈设为状态‘1’，出栈设为状态‘0’。

n个数的所有状态对应n个1和n个0组成的2n位二进制数。

假定等待入栈的操作数按照1..n的顺序排列，

在出栈的过程中，入栈的操作数b大于等于出栈的操作数a(a≤b)，

输出序列的总数目=由左而右扫描由n个1和n个0组成的2n位二进制数，其中：任何一个位置上之前出现的1的累计数不小于0的累计数的方案种数。

栈的应用举例

不符合要求的数的特征：

由左而右扫描时，必然在某一奇数位2m+1位上首先出现m+1个0的累计数和m个1的累计数，此后的2(n-m)-1位上有n-m个 1和n-m-1个0。如若把后面这2(n-m)-1位上的0和1互换，使之成为n-m个0和n-m-1个1，结果得1个由n+1个0和n-1个1组成的2n位数，即一个不合要求的数对应于一个由n+1个0和n-1个1组成的排列。

同理：

由左而右扫描时，必然在某一偶数位2m任何一个由m+1个0和m-1个1组成的2m位二进制数，由于0的个数多2个，2m为偶数，故必在某一个奇数位上出现0的累计数超过1的累计数。

因此，不符合要求的2n位数与n+1个0，n－1个1组成的排列一一对应。

栈的应用举例

输出序列的总数目：

c(2n,n)-c(2n,n+1)=c(2n,n)/(n+1)，

卡特兰数

$C_{2n}^{n}/(n+1)$ **=(2n)!/(n!*n!)/(n+1)**

栈的应用举例

(1)买票找零

有2n个人排成一行进入剧场。入场费5元。其中只有n个人有一张5元钞票，另外n人只有10元钞票，剧院无其它钞票，问有多少种方法使得只要有10元的人买票，售票处就有5元的钞票找零？(将持5元者到达视作将5元入栈，持10元者到达视作使栈中某5元出栈)

(2) 洗碗

饭后，姐姐洗碗，妹妹把姐姐洗过的碗一个一个地放进碗橱摞成一摞。一共有n个不同的碗，洗前也是摞成一摞的，也许因为小妹贪玩而使碗拿进碗橱不及时，姐姐则把洗过的碗摞在旁边，问：小妹摞起的碗有多少种可能的方式?

通用的递归转非递归方法

尾递归

```
long  fact(int n){
    if (n<0) return 1;
    return n*fact(n-1);
}
```

```
int fun(int n){
        if (n==1) return 1;
        else return n+fun(n-1);
}
```

通用的递归转非递归方法

尾递归?

```
int FibonacciRecur(int n) {
  if (0==n) return 0;
  if (1==n) return 1;
  return FibonacciRecur(n-1)+FibonacciRecur(n-2);
}
int FibonacciTailRecur(int n, int acc1, int acc2) {
  if (0==n) return acc1;
  return FibonacciTailRecur(n-1, acc2, acc1+acc2);
}
```

通用的递归转非递归方法

设有一个背包，可以放入的物品重量为s，现有n件商品，重量分别为w_1，w_2.....w_n,问能否从这些商品中选择若干个放入到背包中，使得其重量之和正好为s。如果存在这样的一组商品，则问题有解（true），否则问题无解（false）。

knap(s,n)

- true，当s=0
- false，当s<0 或 s>0 且 n<1
- knap(s-w_{n-1},n-1) || knap(s,n-1)

通用的递归转非递归方法

设有一个背包，可以放入的物品重量为s，现有n件商品，重量分别为w_1，w_2.....w_n,问能否从这些商品中选择若干个放入到背包中，使得其重量之和正好为s。如果存在这样的一组商品，则问题有解（true），否则问题无解（false）。

```
bool knap(int s,int n){
    if (0 == s) return true;
    if ( (s<0) || ( s>0 && n<1) )  return false;
    if (knap(s-w_{n-1},n-1)) {
            cout << w[n-1];
          return true;
    } else return knap(s,n-1);
}
```

通用的递归转非递归方法

设有一个背包，可以放入的物品重量为s，现有n件商品，重量分别为w_1，w_2.....w_n,问能否从这些商品中选择若干个放入到背包中，使得其重量之和正好为s。如果存在这样的一组商品，则问题有解（true），否则问题无解（false）。

```
bool knap(int s,int n){
    if (0 == s) return true;
    if ( (s<0) || ( s>0 && n<1) )  return false;
    if (knap(s-w_{n-1},n-1)) {
            cout << w[n-1];
          return true;
    } else return knap(s,n-1);
}
```

通用的递归转非递归方法

```
bool knap(int s,int n){
    if (0 == s) return true;
    if ( (s<0) || ( s>0 && n<1) )  return false;
    if (knap(s-w_{n-1},n-1)) {
            cout << w[n-1];
          return true;
    } else return knap(s,n-1);
    ;
}
```

```
enum rdType {0,1,2};
public class knapNode {
    int s,n;
    rdType rd;
    bool   k;
}
```

通用的递归转非递归方法

```
bool knap(int s,int n){
    if (0 == s) return true;
    if ( (s<0) || ( s>0 && n<1) )  return false;
    if (knap(s-w_{n-1},n-1)) {
            cout << w[n-1];
          return true;
    } else return knap(s,n-1);
    ;
}
```

```
enum rdType {0,1,2};
public class knapNode {
    int s,n;
    rdType rd;
    bool   k;
}
```

```
Stack<knapNode> stack;
KnapNode temp,x;
bool knap(int s,int n){

    //整个函数的入口处
                        temp.s = s; temp.n = n; temp.rd = 0;
                        stack.push(temp);

                        lable0:
                          stack.pop(&temp);
                                                  if (0 == temp.s ) {
                                                    temp.k = true;
                                                    stack.push(temp);
                                                    goto label3;
                                                  }
    if ((temp.s<0) ⁞ ( temp.s>0 && temp.n<1) ) {
                                                    temp.k = false;
                                                    stack.push(temp);
                                                    goto label3;
                                                  }
    stack.push(temp);
                                                  // 第1个入口
    x.s = temp.s - w[temp.n - 1];
                                                  x.n = temp.n - 1;
                                                  x.rd = 1;
                                                  stack.push(x);
                                                  goto Lable0;
                        label1:
                          stack.pop(&x);
                                                  if (temp.k == true) {
                                                     x.k =true;
                                                                          stack.push(x);
                                                                          cout << w[x.n-1] << endl;
                                                                          goto label3;
                                                  }
    stack.push(x);
    temp.s=x.s
                                                  temp.n=x.n-1;
                                                  temp.rd = 2;
                                                  stack.push(temp);
                                                  goto label0;
                        label2:
    stack.pop(&x);
    x.k = temp.k;
                                                  stack.push(x);
                        label3:
    stack.push(&temp);
    switch(temp.rd) {
                                                    case 0: return temp.k;
                                                    case 1: goto label1;
                                                    case 2: goto label2;
                                                  }
}
```

通用的递归转非递归方法

设计一个工作栈，保存当前记录，包含函数中出现的所有参数，局部变量，返回语句标号，返回值等信息，类似操作系统所作的工作；

设置t+2个语句标号；label0标示入口第一个可执行语句；lablet+1 则放置在函数结尾出口处；labeli为第i个递归返回的语句；

增加非递归入口

替换第i个规则：push（）；goto label0；labeli：pop（x）...可能需要继续传递结果；

增加出口标记；

增加出口语句：switch

优化代码

递归转非递归的另一个例子

```
int exmp(int n) {
   if(n<2) return n+1;
   else return exmp(n/2) *
exmp(n/4);
 }
```

```
void exmp(int n,int& f)
{
    int u1,u2;
    if(n<2) f = n+1;
    else {
        exmp(n/2,u1);
        exmp(n/4,u2);
        f = u1*u2;
    }
}
```

```
typedef struct elem  //存储递归函数的现场信息
{
    int rd;          //返回语句的标号，rd=0~t+1
    int pn,pf;       //函数形参,pn表示参数n,pf表示参数f
    int q1,q2;       //局部变量,q1表示u1,q2表示u2
}ELEM;
class nonrec
{
    private:
        std::stack<ELEM> S;
    public:
        nonrec(void){}
        void replace1(int n,int& f);
        void replace2(int n,int& f);
};
void nonrec::replace1(int n,int& f)
{
    ELEM x,tmp;
    x.rd=3;                      //因为exmp内共调用2次递归子函数，t=2，所以td=t+1=3,压到栈底作监视哨
    x.pn=n;
    S.push(x);                   //调用最开始函数,递归的总入口。相当于调用exmp(7,f)
label0:
    if( (x = S.top()).pn < 2)    //处理递归出口,所有递归出口处需要增加语句goto label t+1。这也是递归语句第一条可执行语句
    {
        S.pop();
        x.pf = x.pn + 1;         //获得函数的解pf
        S.push(x);
        goto label3;             //因为递归出口语句执行完后需要处理函数返回,而lable t+1是用来处理函数返回需要做的工作的，所以需要goto lable3
      }

        x.rd = 1;                //调用第一个递归函数,位于label0的后面，所以如果不满足递归出口会不断调用这里，直到满足递归出口
        x.pn = (int)(x.pn/2);
        S.push(x);               //一次调用使用一个堆栈数据
        goto label0;             //调用后开始进入函数内部，由于函数的第一条执行语句位于lable0，所以需要goto label0


label1: tmp = S.top();           //label1处理第1个递归函数返回时需要进行的处理，通常是pop自己的数据，然后把计算结果放到调用者对应的数据内
        S.pop();
        x = S.top();
        S.pop();
        x.q1 = tmp.pf;           //获取第1个递归函数计算的结果，并回传给上层函数的q1
        S.push(x);

        x.rd = 2;                //调用第二个递归函数
        x.pn = (int)(x.pn/4);
        S.push(x);
        goto label0;

label2: tmp = S.top();           //从第二个递归函数中返回
        S.pop();
        x = S.top();
        S.pop();
        x.q2 = tmp.pf;
        x.pf = x.q1 * x.q2;
        S.push(x);
label3:                          //递归出口（label0)结束后会调用这里
        switch((x=S.top()).rd)
        {
            case 1:
            goto label1;
            break;
            case 2:
            goto label2;
            break;
            case 3:              //t+1处的label：表示整个函数结束
            tmp = S.top();
            S.pop();
            f = tmp.pf;          //最终的计算结果
            break;
            default:
            break;
}
```