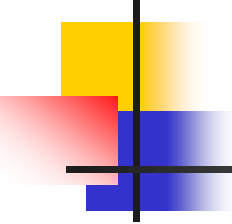


- 
-
- 使用栈可将表达式的中缀表示转换成它的前缀表示和后缀表示。
 - 为了实现这种转换，需要考虑各操作符的优先级。



优先级

操作符

1

单目-、!

2

*, /, %

3

+, -

4

<, <=, >, >=

5

==, !=

6

&&

7

||



中綴表示转后綴表示

- 如：

- 中綴表示：

- $A + B * (C - D) - E / F$

- 后綴表示：

- $ABCD-*+EF/-$



各个算术操作符的优先级

操作符 ch	;	(*, /, %	+, -)
isp (栈内)	0	1	5	3	6
icp (栈外)	0	6	4	2	1



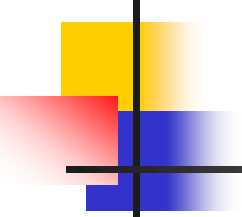
各个算术操作符的优先级

- isp叫做栈内(in stack priority)优先数
- icp叫做栈外(in coming priority)优先数
- 操作符优先数相等的情况：
 - 只出现在括号配对或栈底的“;”号与输入流最后的“;”号配对时。



利用栈将中缀表达式 转换为后缀表达式的算法

- 1) 操作符栈初始化;
- 2) 将结束符 ‘;’ 进栈;
- 3) 读入中缀表达式字符流的首字符 **ch**;
- 4) 重复执行以下步骤, 直到 **ch = ‘;’**, 同时栈顶的操作符也是 ‘;’, 停止循环。

- 
- ◆ 若 ch 是操作数直接输出，读入下一个字符 ch 。
 - ◆ 若 ch 是操作符，判断 ch 的优先级 icp 和位于栈顶的操作符 op 的优先级 isp 。
 - ◆ 若 $icp(ch) > isp(op)$ ，令 ch 进栈，读入下一个字符 ch 。
 - ◆ 若 $icp(ch) < isp(op)$ ，退栈并输出。
 - ◆ 若 $icp(ch) == isp(op)$ ，退栈但不输出，若退出的是“(”号读入下一个字符 ch 。

5) 算法结束，输出序列即为所需的后缀表达式。



步	输入	栈内容	语义	输出	动作
1		;			栈初始化
2	A	;		A	操作数A输出, 读字符
3	+	;	+ > ;		操作符+进栈, 读字符
4	B	;+		B	操作数B输出, 读字符
5	*	;+	* > +		操作符*进栈, 读字符
6	(;+*	(> *		操作符(进栈, 读字符
7	C	;+*(C	操作数C输出, 读字符
8	-	;+*(- > (操作符-进栈, 读字符
9	D	;+*(-		D	操作数D输出, 读字符
10)	;+*(-) < -	-	操作符-退栈输出
11		;+*() = ((退栈, 消括号, 读字符



步	输入	栈内容	语义	输出	动作
12	-	;+*	- < *	*	操作符*退栈输出
13		;+	- < +	+	操作符+退栈输出
14		;	- > ;		操作符-进栈, 读字符
15	E	; -		E	操作数E输出, 读字符
16	/	; -	/ > -		操作符/进栈, 读字符
17	F	; - /		F	操作数F输出, 读字符
18	;	; - /	; < /	/	操作符/退栈输出
19		; -	; < -	-	操作符-退栈输出
20		;	; = ;		;配对, 转换结束



思考题：

利用栈将中缀表达式转换为 前缀表达式的算法

计算表达式的值

$$a + b * (c - d) - e / f$$

Diagram illustrating the evaluation of the expression $a + b * (c - d) - e / f$ using temporary registers (rst1 to rst5) to store intermediate results:

- $c - d$ is stored in **rst1**.
- $b * (c - d)$ is stored in **rst2**.
- $a + b * (c - d)$ is stored in **rst3**.
- e / f is stored in **rst4**.
- $a + b * (c - d) - e / f$ is stored in **rst5**.

■ 使用两个栈:

■ 操作符栈 **OPTR** (operator)

■ 操作数栈 **OPND**(operand)

■ 为了实现这种计算, 需要考虑各操作符的优先级



中缀算术表达式求值

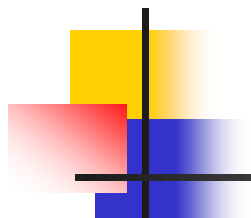
■ 对中缀表达式求值的一般规则：

1. 建立并初始化OPTR栈和OPND栈，然后在OPTR栈中压入一个“#”
2. 扫描中缀表达式，取一字符送入ch。
3. 当 $ch \neq \text{'\#'}$ 或OPTR栈的栈顶 $\neq \text{'\#'}$ 时，执行以下工作，否则结束算法。在OPND栈的栈顶得到运算结果。
 - ①若ch是操作数，进OPND栈，从中缀表达式取下一字符送入ch；

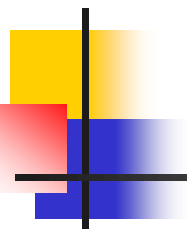


② 若ch是操作符，比较 $icp(ch)$ 的优先级和 $isp(OPTR)$ 的优先级：

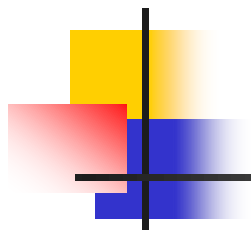
- ◆ 若 $icp(ch) > isp(OPTR)$ ，则ch进OPTR栈，从中缀表达式取下一字符送入ch；
- ◆ 若 $icp(ch) < isp(OPTR)$ ，则从OPND栈退出a2和a1，从OPTR栈退出 θ ，形成运算指令 $(a1)\theta(a2)$ ，结果进OPND栈；
- ◆ 若 $icp(ch) == isp(OPTR)$ 且 $ch == ')'$ ，则从OPTR栈退出'('，对消括号，然后从中缀表达式取下一字符送入ch；



步	输入	OPND	OPTR	语义	动作
1			#		栈初始化
2	A				操作数A进栈,读字符
3	+	A	#	+ > #	操作符+进栈,读字符
4	B	A	#+		操作数B进栈,读字符
5	*	AB	#+	* > +	操作符*进栈,读字符
6	(AB	#+*	(> *	操作符(进栈,读字符
7	C	AB	#+*(操作数C进栈,读字符
8	-	ABC	#+*(- > (操作符-进栈,读字符
9	D	ABC	#+*(-		操作数D进栈,读字符
10)	ABCD	#+*(-) < -	D、C、-退栈,计算C-D,结果r1进栈



步	输入	OPND	OPTR	语义	动作
11		ABr1	#+*() = ((退栈, 消括号, 读字符
12	-	ABr1	#+*	- < *	r1、B、*退栈, 计算 B*r1, 结果r2进栈
13		Ar2	#+	- < +	r2、A、+退栈, 计算 A+r2, 结果r3进栈
14		r3	#	- > #	操作符-进栈, 读字符
15	E	r3	#-		操作数E进栈, 读字符
16	/	r3E	#-	/ > -	操作符/进栈, 读字符
17	F	r3E	#-/		操作数F进栈, 读字符
18	#	r3EF	#-/	# < /	F、E、/退栈, 计算E/F, 结果r4进栈

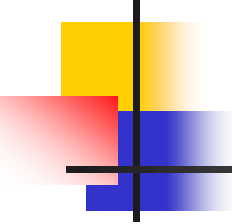


步	输入	OPND	OPTR	语义	动作
19		r3r4	#-	# < -	r4、r3、-退栈，计算 r3-r4，结果 r5 进栈
20		r5	#		算法结束，结果r5



```
void InFixRun()
```

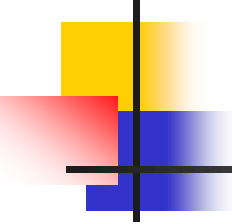
```
{  
    stack<char> OPTR, OPND;  
    char ch, op;  
    OPTR.makeEmpty(); OPND.makeEmpty();  
    OPTR.Push('#');           //两个栈初始化  
    getchar(ch);              //读入一个字符  
    op = '#';  
    while (ch != '#' || op != '#') {  
        if (isdigit(ch))      //是操作数, 进栈  
            { OPND.Push(ch); getchar(ch); }  
        else {                 //是操作符, 比较优先级  
            OPTR.GetTop(op);    //读一个操作符
```



```

        if ( icp(ch) > isp(op) )    //栈顶优先级低
        { OPTR.Push (ch); getchar(ch); }
    else if ( icp(ch) < isp(op) )
    {
        OPTR.Pop(op);                //栈顶优先级高
        OPND.DoOperator(op); // (a1)θ(a2)
    }
    else if ( ch == ')' )            //优先级相等
    { OPTR.Pop(op); getchar(ch); }
} /*end of if...else*/
OPTR.GetTop(op);
} /*end of while*/
}

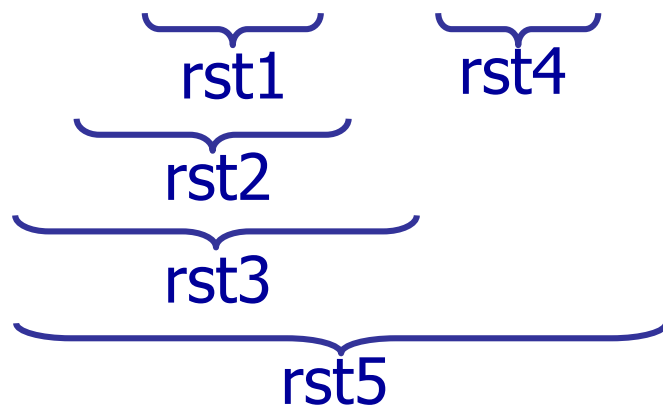
```



思考题：
应用后缀表示计算表达式的值

应用后缀表示计算表达式的值

- 从左向右顺序地扫描表达式，并用一个栈暂存扫描到的操作数或计算结果。
- 在后缀表达式的计算顺序中已隐含了加括号的优先次序，括号在后缀表达式中不出现。
- 计算例 $a\ b\ c\ d\ -\ *\ +\ e\ f\ /\ -$





栈的应用：递归



递归(Recursion)的定义

- 若一个对象部分地包含它自己，或，用它自己给自己定义，则称这个对象是递归的
- 若一个过程直接地或间接地调用自己，则称这个过程是递归的过程
- 以下三种情况常常用到递归方法
 - ◆ 定义是递归的
 - ◆ 数据结构是递归的
 - ◆ 问题的解法是递归的



定义是递归的

例如，阶乘函数

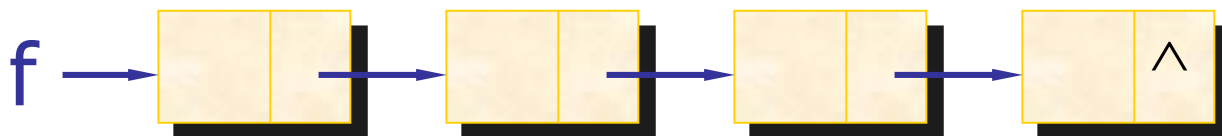
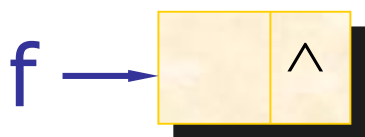
$$n! = \begin{cases} 1, & \text{当 } n = 0 \text{ 时} \\ n * (n-1)!, & \text{当 } n \geq 1 \text{ 时} \end{cases}$$

求解阶乘函数的递归算法

```
long Factorial(long n)
{
    if (n == 0) return 1;
    else return n*Factorial(n-1);
}
```

数据结构是递归的

- 例如，单链表结构

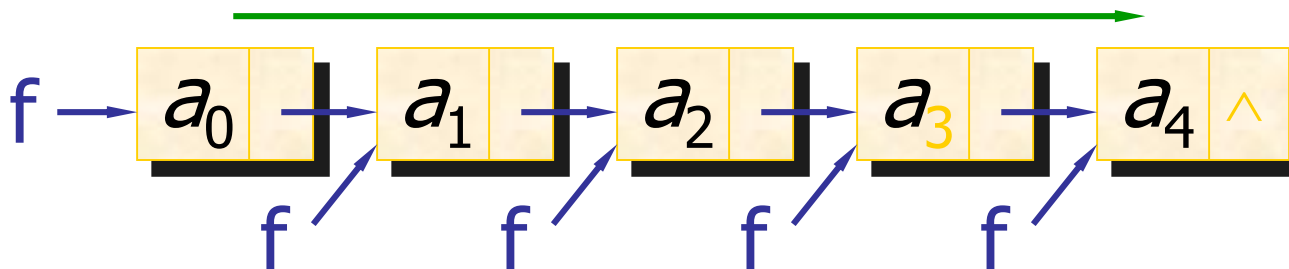


- 一个结点，它的指针域为NULL，是一个单链表；
- 一个结点，它的指针域指向单链表，仍是一个单链表。

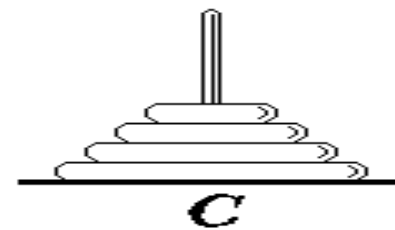
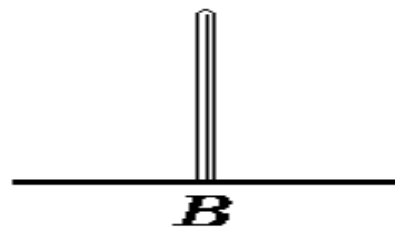
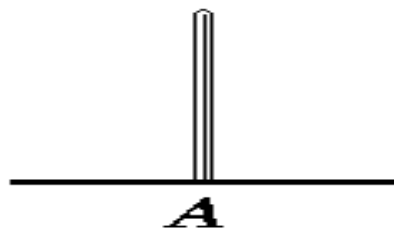
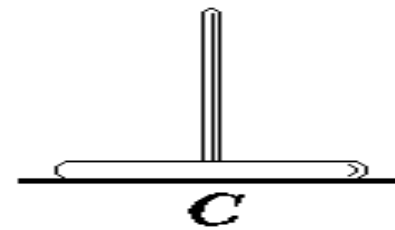
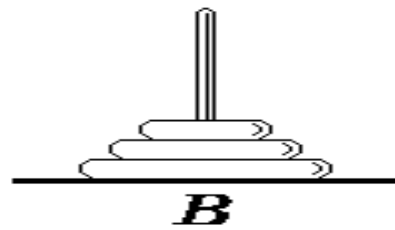
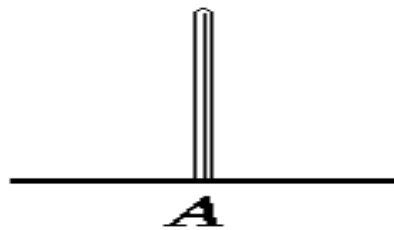
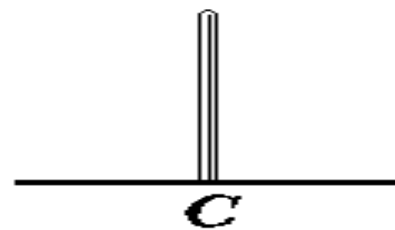
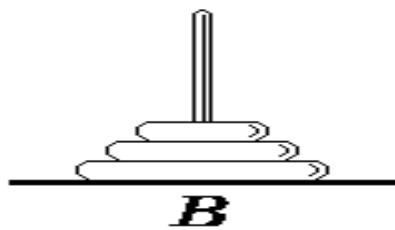
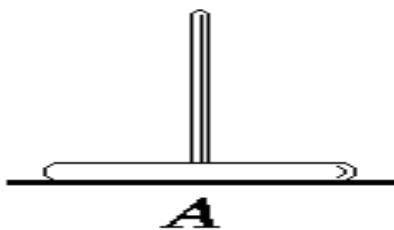
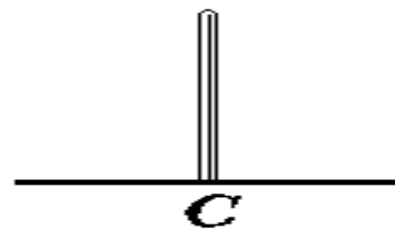
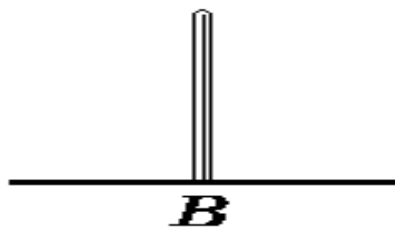
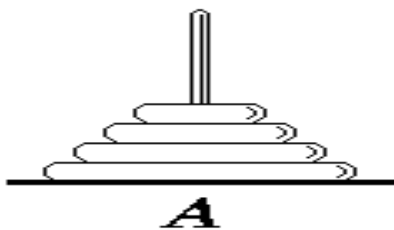
搜索链表最后一个结点并打印其数值

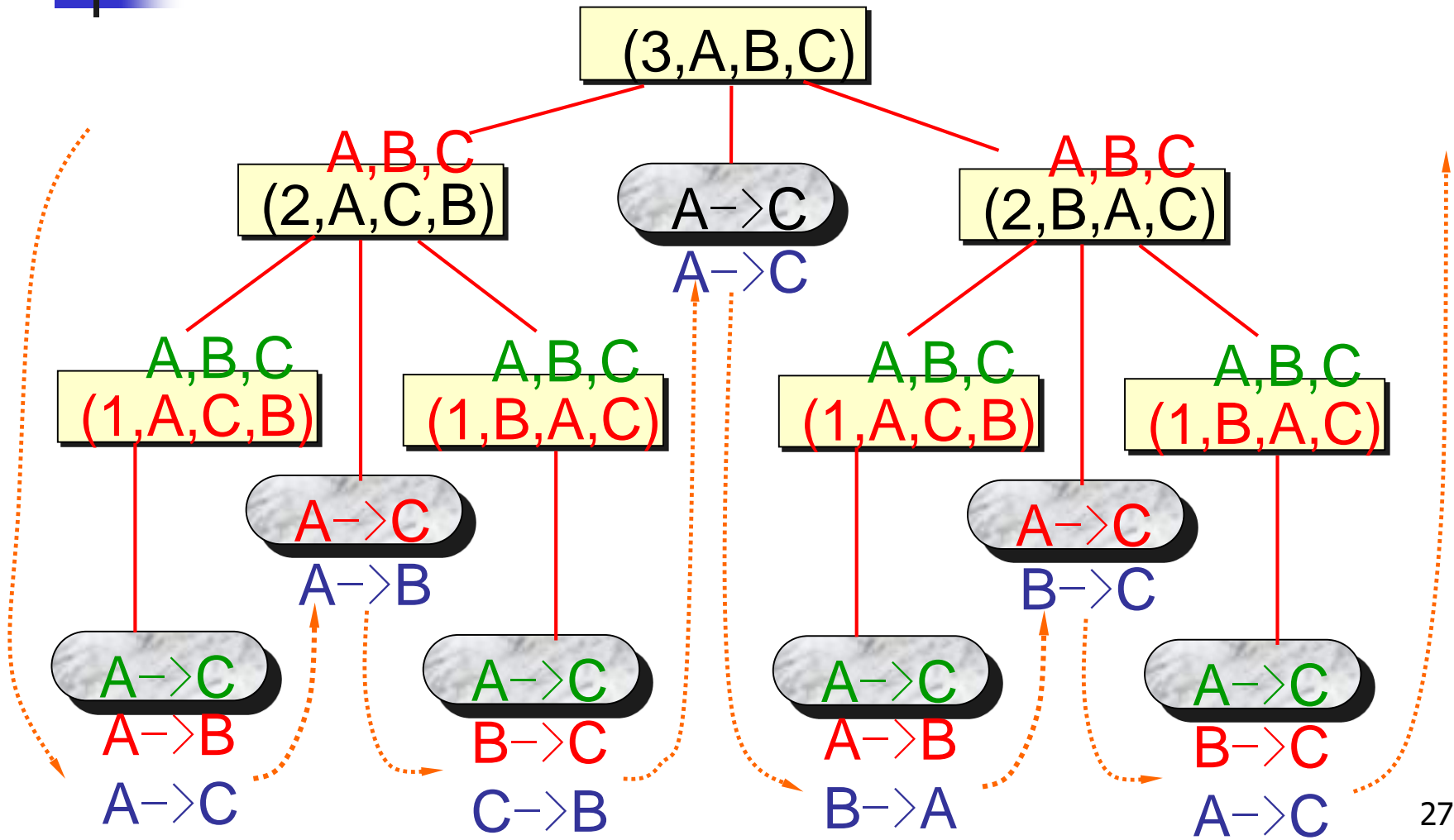
```
template <class E>
void Print(ListNode<E> *f)
{
    if (f ->link == NULL)
        cout << f ->data << endl;
    else Print(f ->link);
}
```

递归找链尾



问题的解法是递归的







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

```
void Hanoi (int n, char A, char B, char C)
```

```
{//解决汉诺塔问题的算法
```

```
    if (n == 1) cout << " move " << A << " to " << C << endl;
```

```
    else { Hanoi(n-1, A, C, B);
```

```
        cout << " move " << A << " to " << C << endl;
```

```
        Hanoi(n-1, B, A, C);
```

```
    }
```

```
}
```



自顶向下、逐步分解的策略

- 子问题应与原问题做同样的事情，且更为简单；
- 解决递归问题的策略是把一个规模比较大的问题分解为一个或若干规模比较小的问题，分别对这些比较小的问题求解，再综合它们的结果，从而得到原问题的解。

— 分而治之策略（分治法）

- 这些比较小的问题的求解方法与原来问题的求解方法一样。



构成递归的条件

- 不能无限制地调用本身，必须有一个出口，化简为非递归状况直接处理

Procedure <name> (<parameter list>)

{

if (< initial condition>) //递归结束条件

 return (initial value);

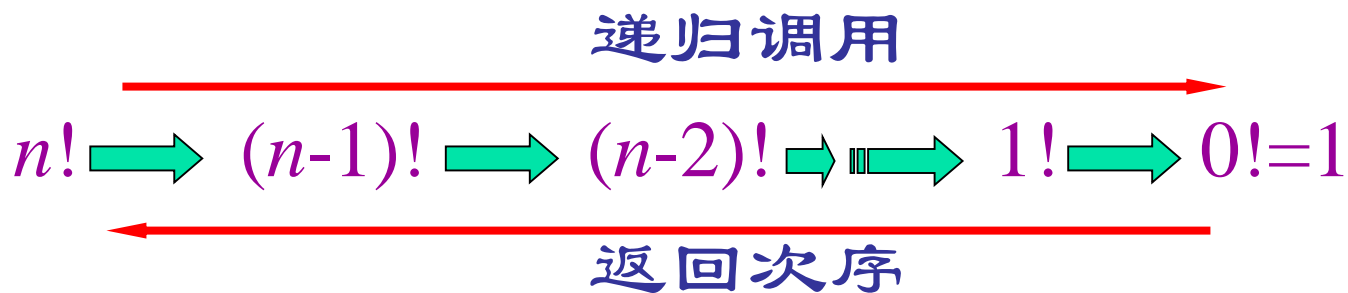
else //递归

 return (<name> (parameter exchange));

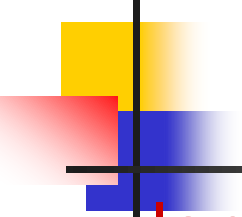
}

递归过程与递归工作栈

- 递归过程在实现时，需要自己调用自己。
- 层层向下递归，退出时的次序正好相反：



- 主程序第一次调用递归过程为外部调用；
- 递归过程每次递归调用自己为内部调用。
- 它们返回调用它的过程的地址不同。

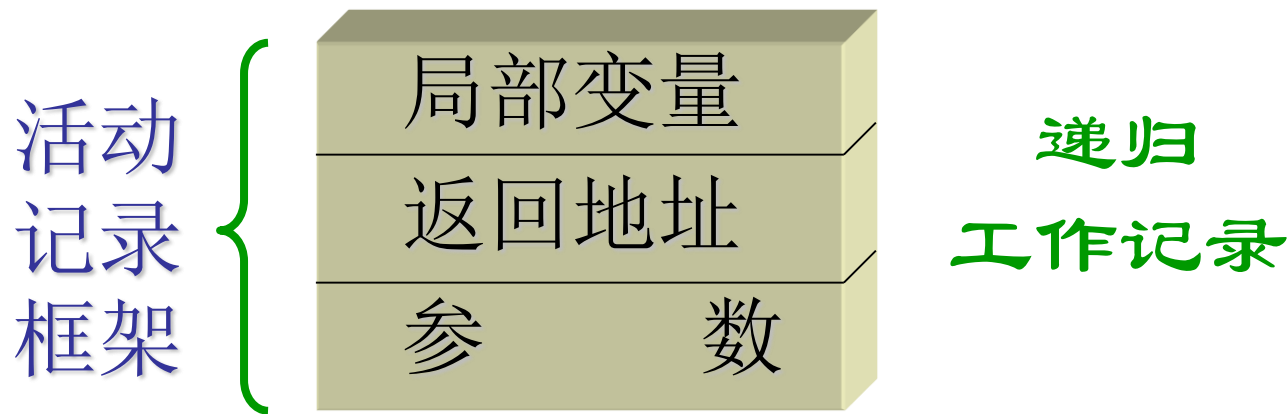


```
long Factorial(long n) {  
    int temp;  
    if (n == 0) return 1;  
    else temp = n * Factorial(n-1);  
RetLoc2 ————— ↑  
    return temp;  
}
```

```
void main() {  
    int n;  
    n = Factorial(4);  
RetLoc1 ————— ↑  
}
```


递归工作栈

- 每一次递归调用时，需要为过程中使用的参数、局部变量等另外分配存储空间。
- 每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。



函数递归时的活动记录

调用块

.....
<下一条指令>

函数块

Function(<参数表>)
.....
<return>

返回地址(下一条指令)

局部变量

参数

计算Fact时活动记录的内容

递归调用序列

参数 返回值 返回地址

返回时的指令

4	24	RetLoc1
---	----	---------

return 4*6 //返回 24

3	6	RetLoc2
---	---	---------

return 3*2 //返回 6

2	2	RetLoc2
---	---	---------

return 2*1 //返回 2

1	1	RetLoc2
---	---	---------

return 1*1 //返回 1

0	1	RetLoc2
---	---	---------

return 1 //返回 1



递归过程改为非递归过程

- 递归过程简洁、易编、易懂
- 递归过程效率低，重复计算多
- 改为非递归过程的目的是提高效率
- 单向递归和尾递归可**直接**用迭代实现其非递归过程
- 其他情形必须**借助**栈实现非递归过程



单向递归



计算斐波那契数列的函数Fib(n)的定义

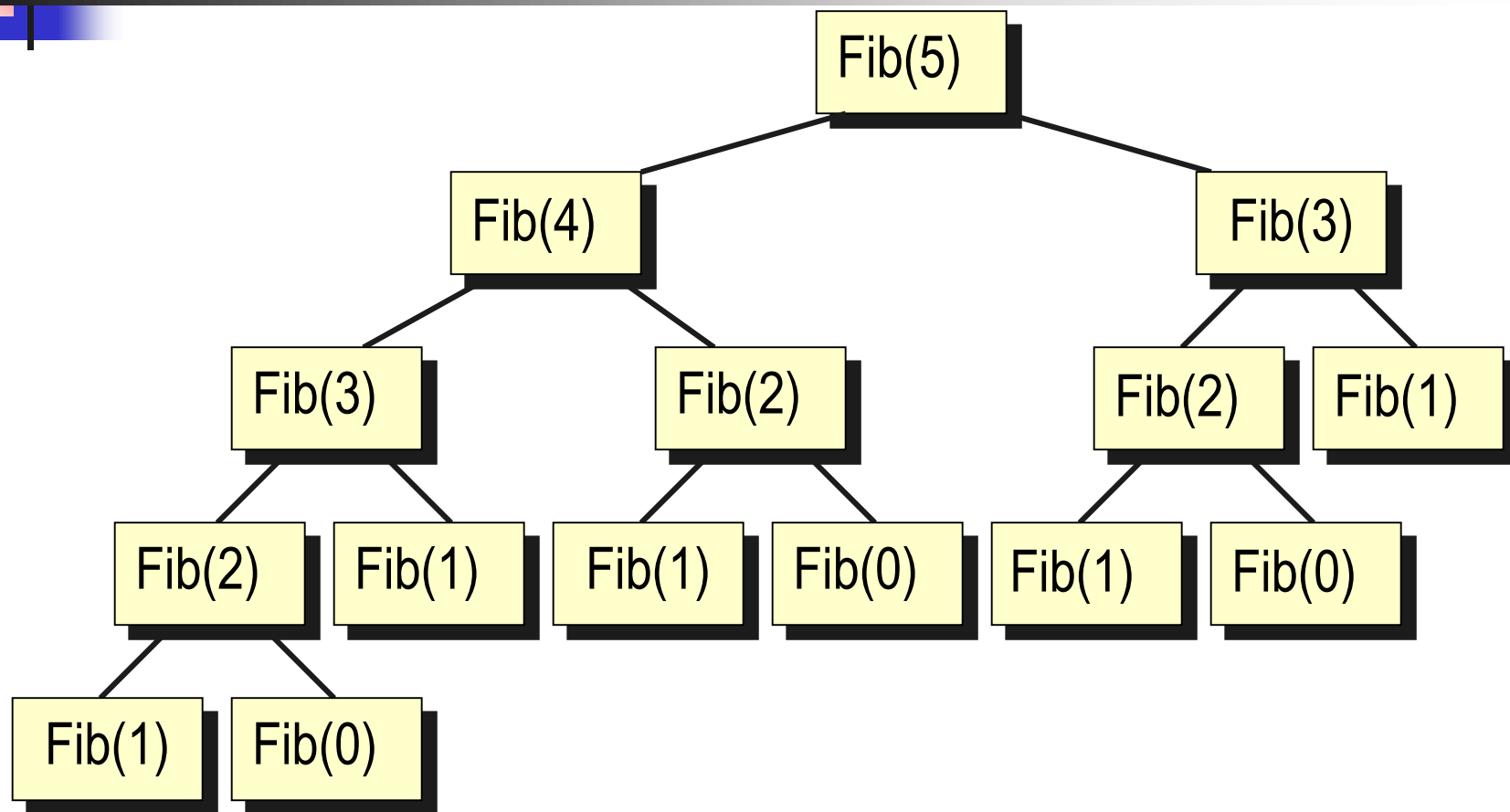
$$\text{Fib}(n) = \begin{cases} n, & n = 0, 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2), & n > 1 \end{cases}$$

如 $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5$

求解斐波那契数列的递归算法

```
long Fib(long n) {  
    if (n <= 1) return n;  
    else return Fib(n-1)+Fib(n-2);  
}
```

斐波那契数列的递归调用树



调用次数 $\text{NumCall}(k) = 2 * \text{Fib}(k+1) - 1$



单向递归用迭代法实现

```
long FibIter(long n)
{
    if (n <= 1) return n;
    long twoback = 0, oneback = 1, Current;
    for (int i = 2; i <= n; i++) {
        Current = twoback + oneback;
        twoback = oneback;
        oneback = Current;
    }
    return Current;
}
```




尾递归

逆向输出

25	36	72	18	99	49	54	63
----	----	----	----	----	----	----	----

```

void recfunc(int A[ ], int n) {
    if (n >= 0) {
        cout << A[n] << " ";
        n--;
        recfunc(A, n);
    }
}

```

尾递归用迭代法实现

```

void sterfunc(int A[ ], int n) {
    //消除了尾递归的非递归函数
    while (n >= 0) {
        cout << "value  " << A[n] << endl;
        n--;
    }
}

```