



编译原理
第七章
运行存储分配

哈尔滨工业大学 陈鄞 单丽莉





提纲

7.1 存储组织

7.2 静态存储分配

7.3 栈式存储分配

7.4 非局部数据的访问

7.5 参数传递

7.6 符号表

7.1 存储组织

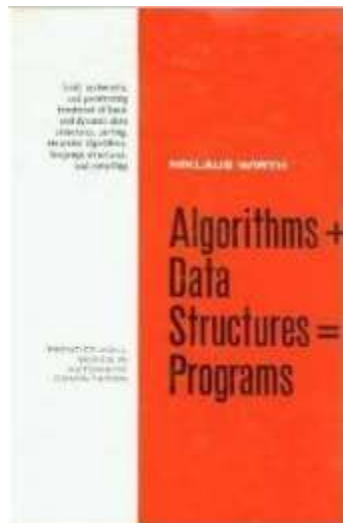
➤ 一个目标程序运行所需的存储空间包括

➤ 代码区

➤ 数据区



Niklaus Wirth



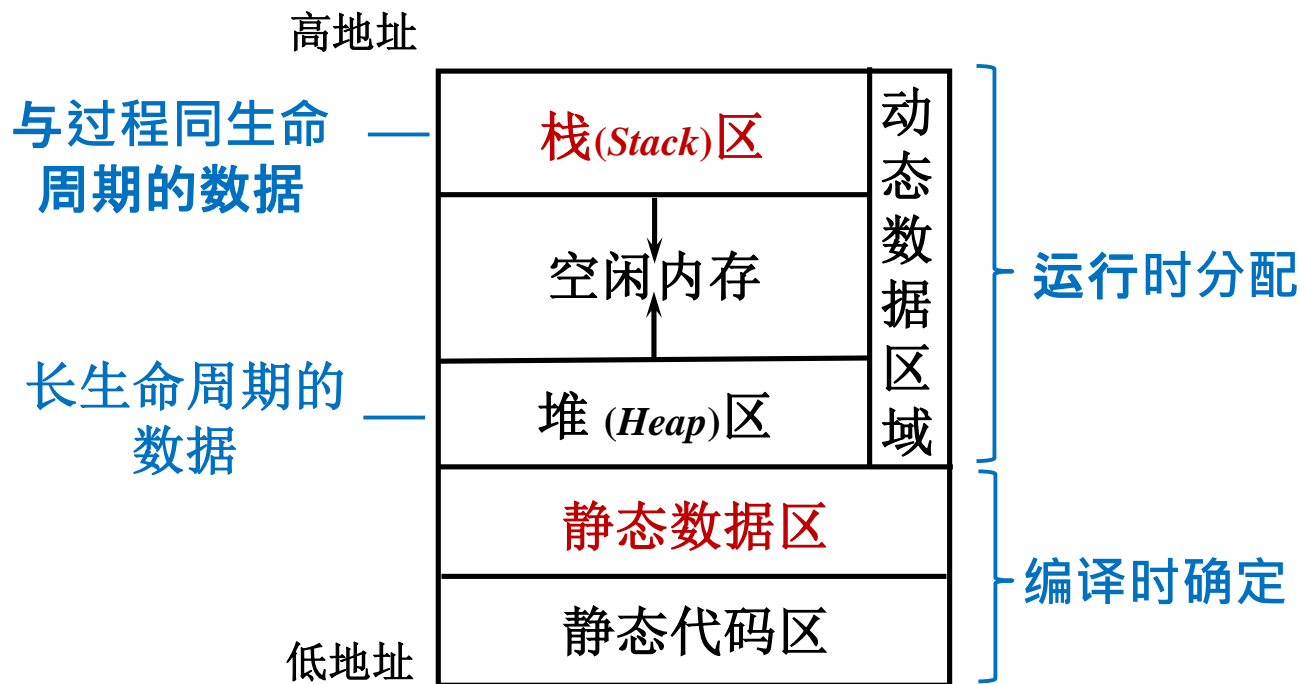
算法 + 数据结构
= 程序

存储分配策略

- 对于那些在**编译时刻**就可以确定大小的数据对象，可以在编译时刻就为它们分配存储空间，这样的分配策略称为**静态存储分配**
- 反之，如果不能在编译时完全确定数据对象的大小，就要采用**动态存储分配**的策略。即在编译时仅产生各种必要的信息，而在**运行时刻**，再动态地分配数据对象的存储空间
 - 栈式存储分配
 - 堆式存储分配

静态和**动态**分别对应**编译时刻**和**运行时刻**

运行时内存的划分



Linux操作系统编译器的可能内存管理

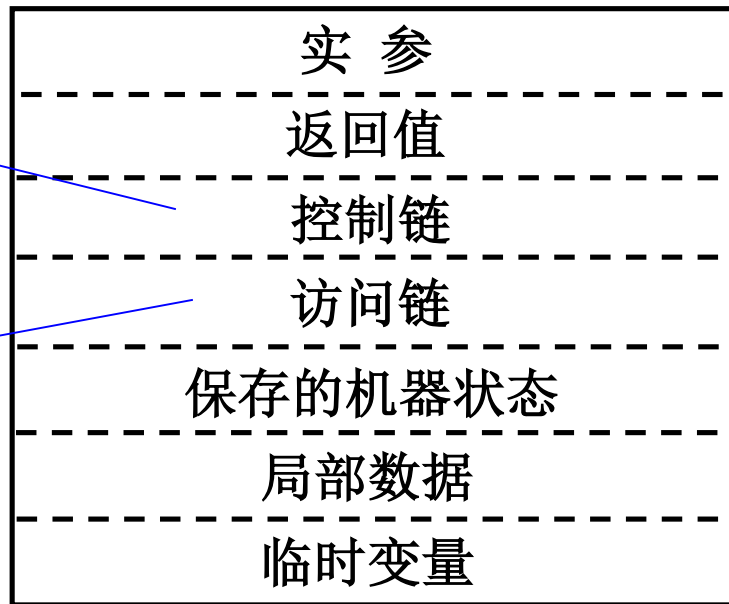
活动记录

- 使用过程(或函数、方法)作为用户自定义动作的单元的语言，其编译器通常以过程为单位分配存储空间
- 过程体的每次执行称为该过程的一个活动(activation)
- 过程每执行一次，就为它分配一块连续存储区，用来管理过程一次执行所需的信息，这块连续存储区称为活动记录(*activation record*)

活动记录的一般形式

指向调用者的活动记录

用来访问存放于其它活动记录中的非局部数据





提纲

7.1 存储组织

7.2 静态存储分配

7.3 栈式存储分配

7.4 非局部数据的访问

7.5 参数传递

7.6 符号表

7.2 静态存储分配

- 在静态存储分配中，编译器为每个过程确定其活动记录在目标程序中的位置
- 这样，过程中每个名字的存储位置就确定了
- 因此，这些名字的存储地址可以被编译到目标代码中
- 过程每次执行时，它的名字都绑定到同样的存储单元

静态存储分配的限制条件

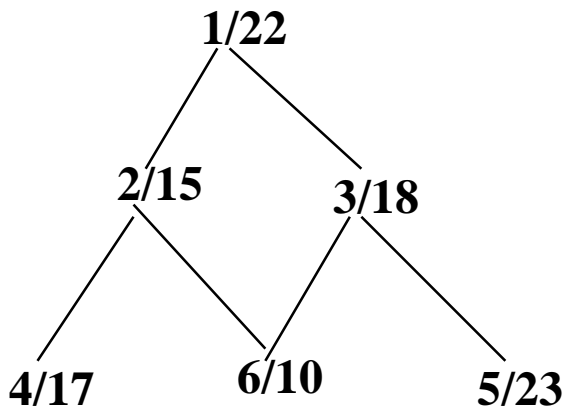
- 适合静态存储分配的语言必须满足以下条件
 - 数组上下界必须是常数
 - 不允许过程的递归调用
 - 不允许用户动态建立数据实体
- 满足这些条件的语言有BASIC和早期的FORTRAN等

常用的静态存储分配方法

- 顺序分配法
- 层次分配法

顺序分配法

- 按照过程出现的先后顺序逐段分配存储空间
- 各过程的活动记录占用互不相交的存储空间



过程编号/所需存储空间

能用更少的空间么？

过程	存储区域
1	0~21
2	22~36
3	37~54
4	55~71
5	72~94
6	95~104

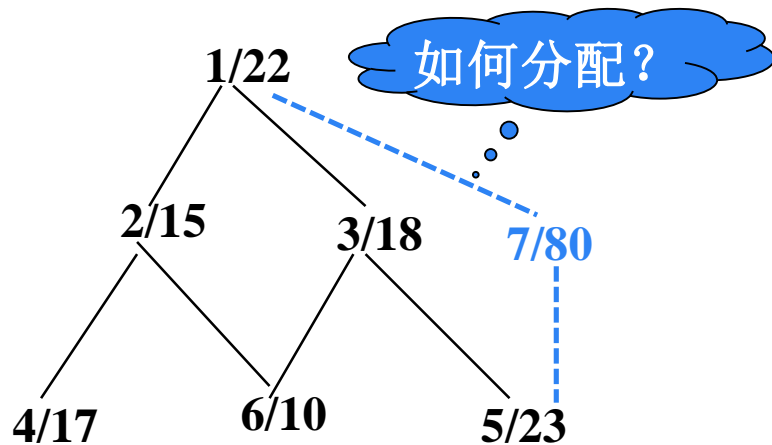
共需要105个存储单元

优点：处理上简单

缺点：对内存空间的使用不够经济合理

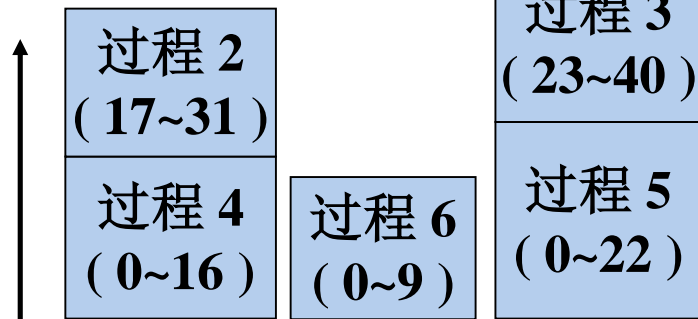
层次分配法

- 通过对过程间的调用关系进行分析，凡属无相互调用关系的并列过程，尽量使其局部数据**共享**存储空间



节点：过程编号/所需空间大小

共需要63个
存储单元

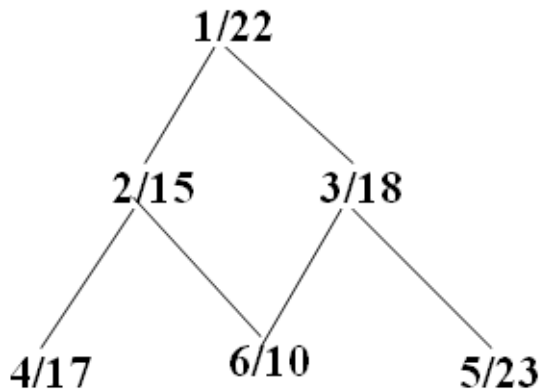


层次分配算法

➤ $B[n][n]$: 过程调用关系矩阵

➤ $B[i][j]=1$: 表示第 i 个过程调用第 j 个过程，且第 j 个过程未分配内存

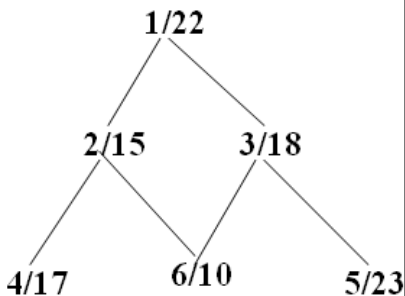
➤ $B[i][j]=0$: 表示第 i 个过程不调用第 j 个过程或第 j 个过程已分配内存



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	0	1	0	1
3	0	0	0	0	1	1
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0

总体思想：先为不调用其它过程的过程分配内存，再为调用的所有过程都已分配内存的过程分配内存。

层次分配算法



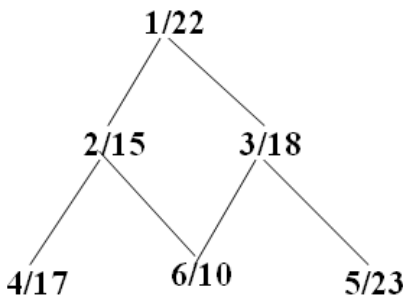
/*初始化所有过程的基地址和分配标记*/

```
for( 每个过程 i ) {  
    过程 i 的基地址初始化为 0 ;  
    过程 i 标记为未分配  
}  
for( 若还有过程未分配内存 )  
    /*找到可以分配的所有过程 */  
    for( 每个过程 i ) {  
        if( 矩阵  $B$  的第  $i$  行的和  $sum = 0$  ) { /*若过程  $i$  调用的  
            所有过程已分配*/  
            为  $i$  分配内存, 标记过程  $i$  已分配;  
            for( 每个调用  $i$  的过程  $k$  ) {  
                if (  $k$  的基地址在过程  $i$  的数据区 )  
                    设置  $k$  的基地址在  $i$  的数据区  
                之上  
                 $B[k][i] = 0$ ; /* 标记  $k$  调用的  $i$  已分配内存 */  
            }  
        }  
    }  
}
```

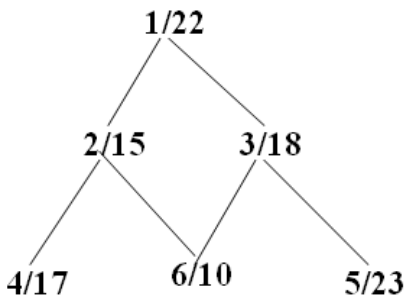
Units[n] : 过程所需内存数量

base[i] : 第 *i* 个过程局部数据区的基地址

allocated[i] : 第 *i* 个过程局部数据区是否分配的标志



```
void MakeFortranBlockAllocated(int *Units, int *base, int NoOfBlocks )
{ /* NoOfBlocks : 过程总数 */
    int i, j, k, Sum;
    int *allocated; /*过程是否已分配*/
    allocated = ( int* ) malloc (sizeof( int ) * NoOfBlocks );
    for( i = 0; i < NoOfBlocks; i ++ ) /*初始化 base和 allocated*/
    {
        base [ i ] = 0;
        allocated [ i ] = 0;
    }
    for( j = 0; j < NoOfBlocks; j ++ )
        for( i = 0; i < NoOfBlocks; i ++ )
        {
            Sum = 0;
            for ( k = 0; k < NoOfBlocks; k ++ )
                Sum += B [ i ] [ k ];          /*计算 j 调用的还未分配内存的过程数量*/
        }
}
```

```
if ( ! Sum && ! allocated [ i ] )
{ /* Sum=0 表示过程 i 不调用未分配内存的过程;
   allocated [ i ]=0; 表示过程 i 未分配内存, 接下来为 i 分配内存*/
   allocated [ i ] = 1;
   printf( " %d: %d -%d /n", i, base[ i ], base[ i ]+Units[ i ]-1 );
   for( k = 0; k < NoOfBlocks; k++ )
   if ( B[k][i] ) /* b[k][i]!=0 表示 过程 k 调用过程 i */
   { /* 由于过程 k 调用过程 i, 过程 k 必须在过程 i 之后分配内存且过程
      k 的起始地址必须大于过程 i 的起始地址*/
      if ( base[ k ] < base[ i ] + Units[ i ] )
      base[ k ] = base[ i ] + Units[ i ];
      B[k][i] = 0; /* 标记所有的 B[k][i] 为调用已分配内存过程 i */
   }
   }
}
free(allocated);
}
```



提纲

7.1 存储组织

7.2 静态存储分配

7.3 栈式存储分配

7.4 非局部数据的访问

7.5 参数传递

7.6 符号表

7.3 栈式存储分配

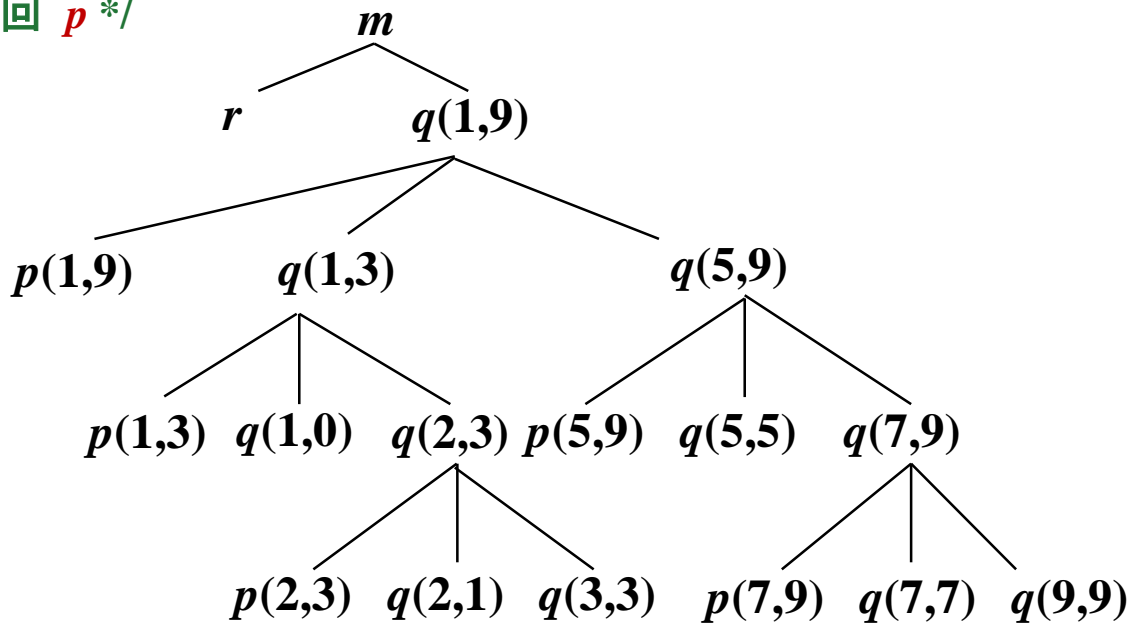
- 使用过程、函数或方法作为代码单元的语言，它们的编译器几乎都把(至少一部分的)运行时刻存储以**栈**的形式进行管理，称为**栈式存储分配**
- **分配策略**：当一个过程被**调用**时，该过程的活动记录被**压入**栈；当过程**结束**时，该活动记录被**弹出**栈，**活动记录也被称为栈帧**
- 这种安排允许**活跃时段不交叠**的多个过程调用之间**共享空间**，而且非局部变量的**相对地址总是固定的**，和过程调用序列无关

活动树

- 用来描述程序运行期间**控制**进入和离开各个**活动**的情况的**树**称为**活动树**
- 树中的每个**结点**对应于一个**活动**。**根结点**是启动程序执行的**main**过程的活动
- 在表示过程 p 的某个活动的结点上，其**子结点**对应于被 p 的这次活动**调用的各个过程的活动**。按照这些**活动被调用的顺序**，**自左向右**地显示它们。一个子结点必须在其**右兄弟**结点的活动**开始之前结束**

例：一个快速排序程序的概要

```
int a[11]; /*全局数组*/
void readArray() /*将9个整数读入到a[1],...,a[9]中 */
{
    int i;
    ...
}
int partition(int m, int n)
{
    /*选择一个分割值v, 划分a[m...n], 使得a[m...p-1]小于v, a[p]=v,
    a[p+1...n]大于等于v。返回 p */
    ...
}
void quicksort(int m, int n)
{
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main()
{
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1, 9);
}
```





活动树

m

r

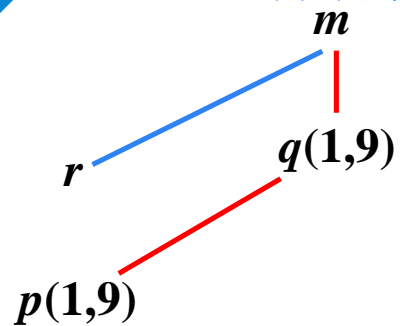


控制栈向下增长

int <i>a</i> [11]
<i>main</i>
<i>r</i>
int <i>i</i>



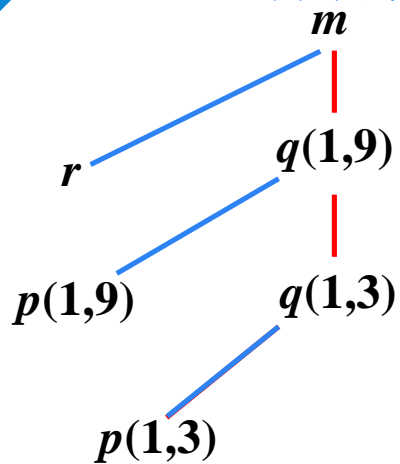
活动树



控制栈向下增长

int $a[11]$
<i>main</i>
<i>q(1,9)</i>
int i
<i>p(1,9)</i>
int i

活动树



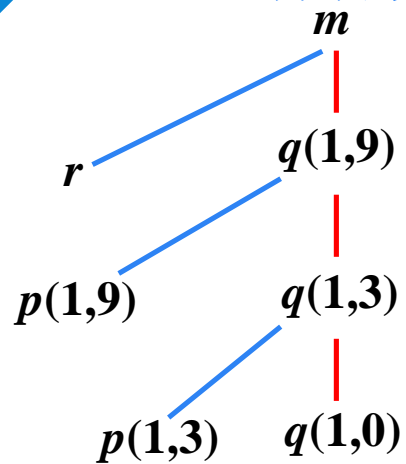
控制栈向下增长

<code>int a[11]</code>
<code>main</code>
<code>q(1,9)</code>
<code>int i</code>
<code>q(1,3)</code>
<code>int i</code>
<code>p(1,3)</code>
<code>int i</code>

当一个过程是递归的时候，常常会有该过程的多个活动记录同时出现在栈中



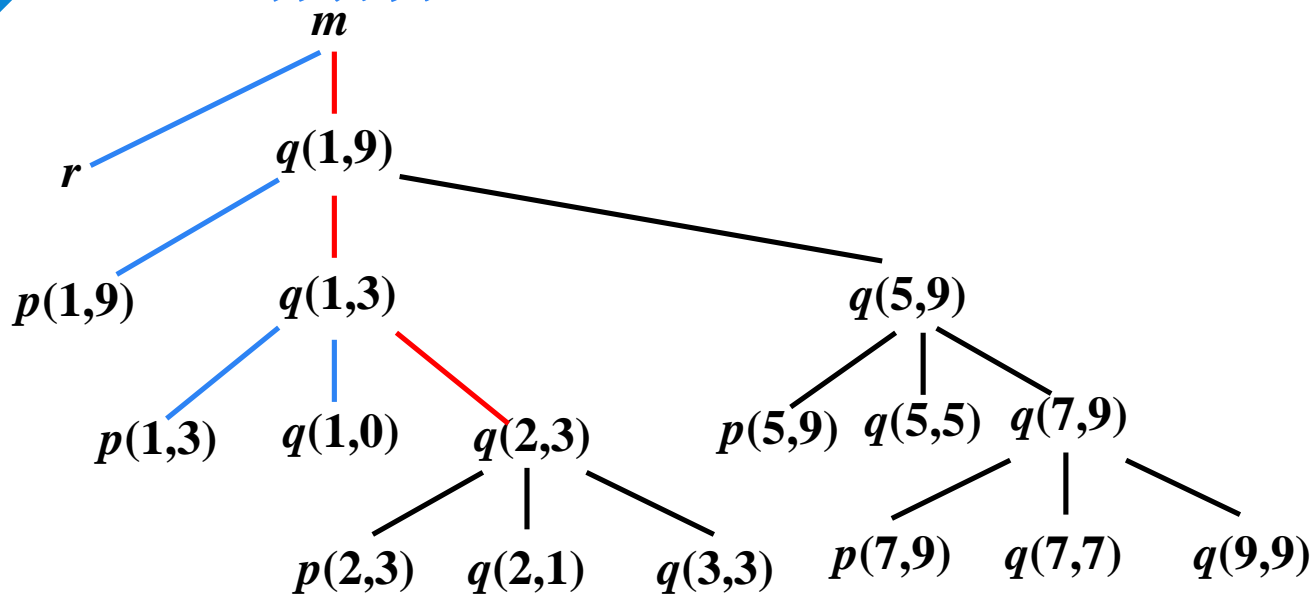
活动树



控制栈向下增长

<code>int a[11]</code>
<code>main</code>
<code>q(1,9)</code>
<code>int i</code>
<code>q(1,3)</code>
<code>int i</code>
<code>q(1,0)</code>
<code>int i</code>

活动树



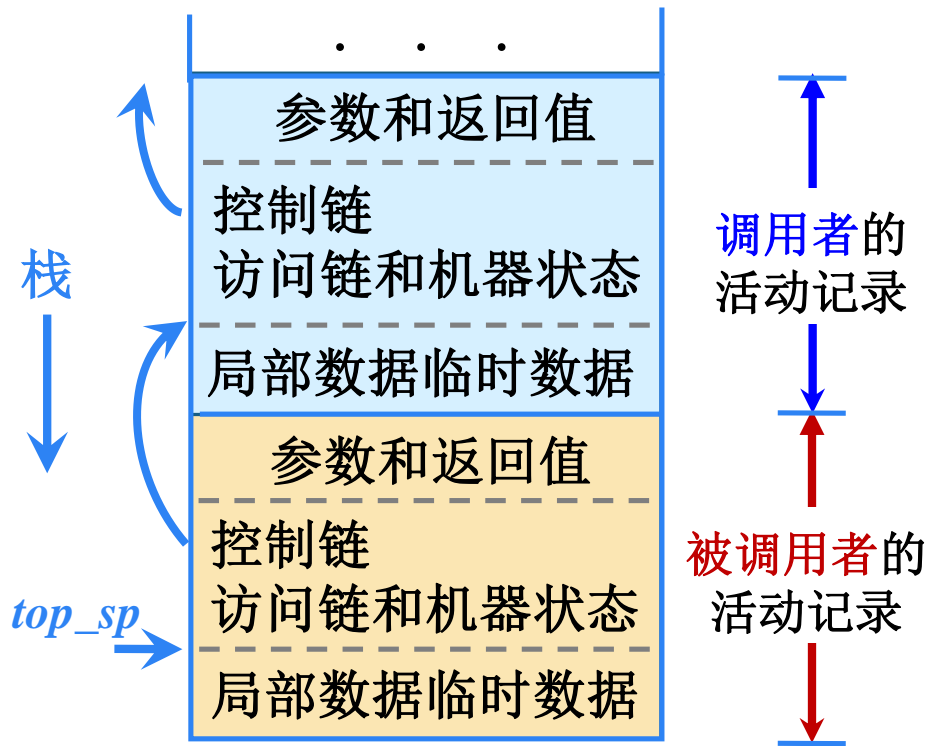
控制栈向下增长

int a[11]
main
q(1,9)
int i
q(1,3)
int i
q(2,3)
int i

- 每个活跃的活动都有一个位于控制栈中的活动记录
- 活动树的根的活动记录位于栈底
- 程序控制所在的活动的记录位于栈顶
- 栈中全部活动记录的序列对应于在活动树中到达当前控制所在的活动结点的路径

设计活动记录的一些原则

- 在调用者和被调用者之间传递的值一般被放在被调用者的活动记录的开始位置，这样它们可以尽可能地靠近调用者的活动记录
- 固定长度的项被放置在中间位置
 - 控制链、访问链、机器状态字
- 在早期不知道大小的项被放置在活动记录的尾部
- 栈顶指针寄存器 top_sp 指向活动记录中机器状态字段的末端即局部数据开始的位置，以该位置作为基地址访问活动记录中的数据

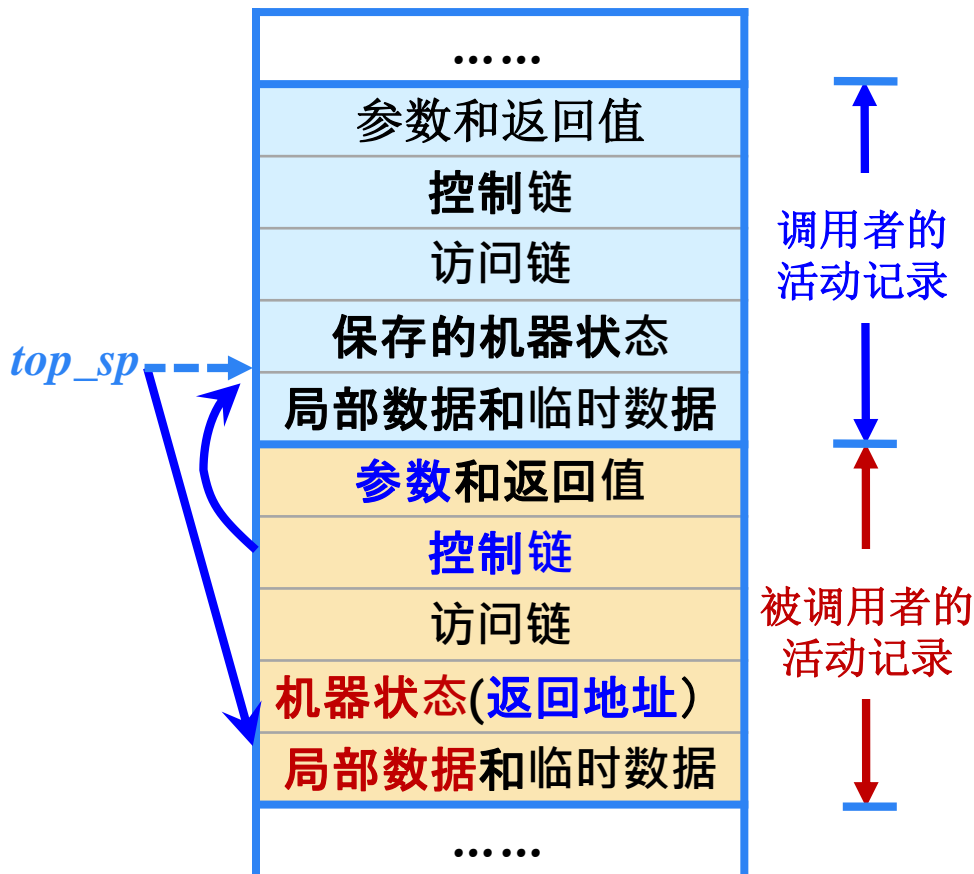


调用序列和返回序列

- 过程调用和过程返回都需要执行一些代码来管理活动记录栈，保存或恢复机器状态等
 - 调用序列
 - 实现过程调用的代码段。为一个活动记录在栈中分配空间，并在此记录的字段中填写信息并保存机器状态
 - 返回序列
 - 恢复机器状态，使得调用过程能够在调用结束之后继续执行
- 调用序列和返回序列中的代码通常被分割到调用过程（调用者）和被调用过程（被调用者）中
 - 分割原则：尽量分割给被调用者，因只被生成一次

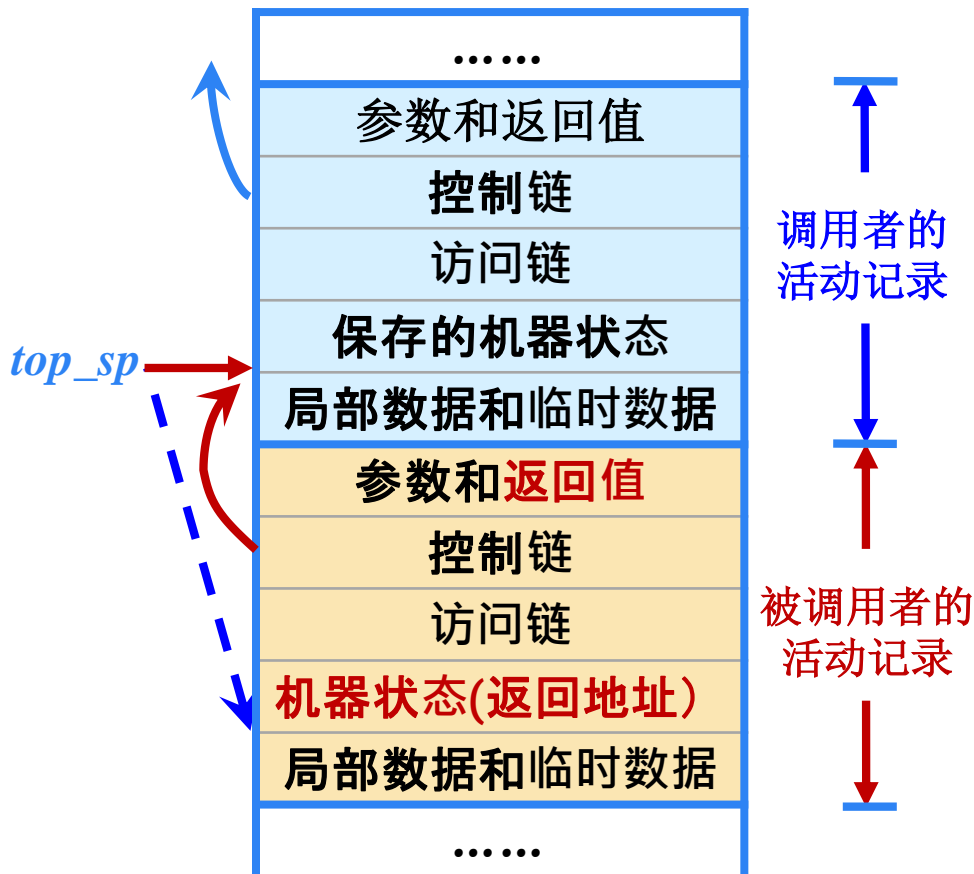
调用序列

- **调用者**计算**实际参数**的值
- **调用者**将**返回地址**（程序计数器的值）放到被调用者的**机器状态字段**中。将自己的 top_sp 值放到被调用者的**控制链**中。然后，**增加 top_sp** 的值，使其指向被调用者**局部数据开始的位置**
- **被调用者**保存寄存器值和其它状态信息
- **被调用者**初始化其局部数据并开始执行

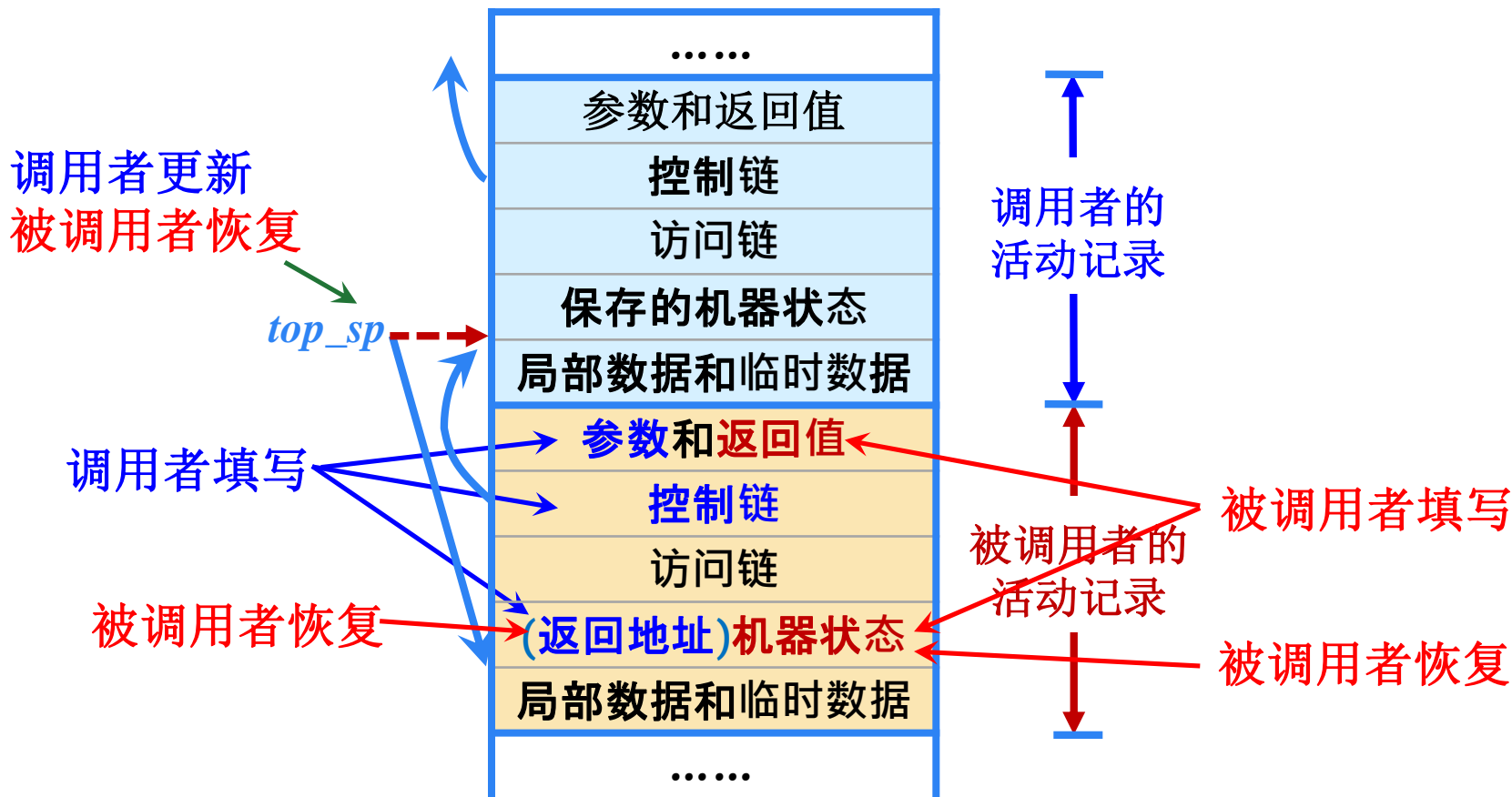


返回序列

- **被调用者**将**返回值**放到与参数相邻的位置
- **被调用者**根据**控制链**恢复调用者的 top_sp ，恢复寄存器信息，然后**跳转到**机器状态字段中的**返回地址**
- 调用者知道**返回值**相对于自己的 top_sp 值的位置。因此，调用者可以使用**返回值**



调用者和被调用者之间的任务划分

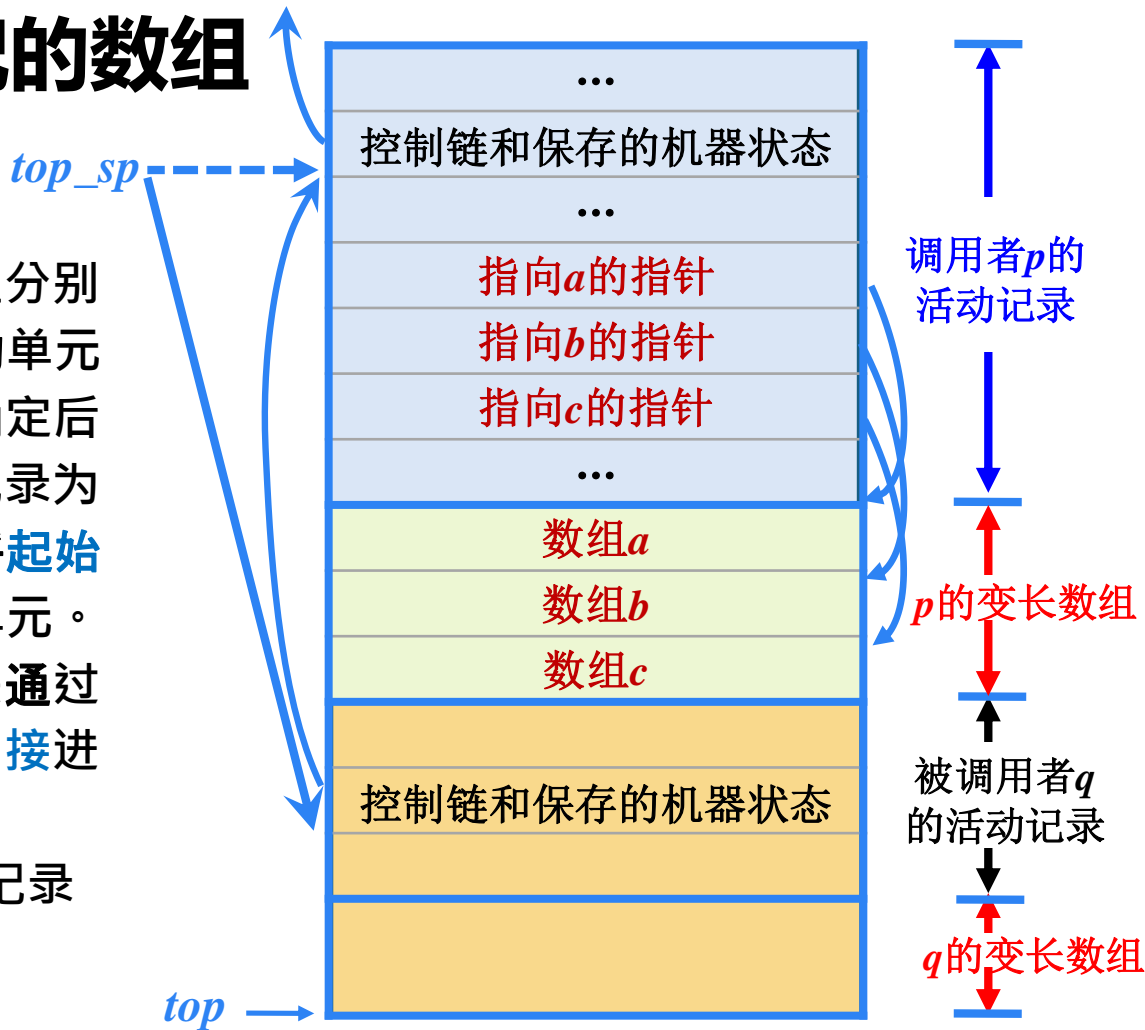


变长数据的存储分配

- 在现代程序设计语言中，在编译时刻不能确定大小的对象通常被分配在堆区。但是，如果它们是过程的局部对象，也可以将它们分配在运行时刻栈中。
 - 尽量将对象放置在栈区的原因：可以避免对它们的空间进行垃圾回收，也就减少了相应的开销
- 只有一个数据对象局部于某个过程，且当此过程结束时它变得不可访问，才可以使用栈为这个对象分配空间

访问动态分配的数组

- 在活动记录中为变长数组分别分配一个存放数组指针的单元
- 运行时，数组的大小能确定后，在栈顶挨着过程活动记录为数组分配存储空间，并将起始地址存放于数组指针的单元。
- 对这些数组元素的引用是通过活动记录中的数组指针间接进行的。
- 当过程 p 结束后，其活动记录连同数组空间一同出栈。





提纲

7.1 存储组织

7.2 静态存储分配

7.3 栈式存储分配

7.4 非局部数据的访问

7.5 参数传递

7.6 符号表

7.4 非局部数据的访问

- 一个过程除了可以使用**过程自身**声明的**局部数据**以外，还可以使用**过程外**声明的**非局部数据**
 - 全局数据
 - **外围过程**定义的数据（支持过程嵌套声明的语言）←块结构语言的静态作用域规则
 - 例：*Pascal*语言、*Python*语言、ML(Meta Language)

例：

➤ (Pascal语言)

红框表示此过程直接定义的名字的作用域

```
program sort ( input, output );  
  var a: array[0..10] of integer;  
      x: integer;  
  procedure readarray;  
    var i: integer;  
    begin ... a ... end. {readarray} ;  
  procedure exchange( i, j : integer);  
    begin x=a[i];a[i]=a[j];a[j]=x; end. {exchange} ;  
  procedure quicksort(m, n:integer);  
    var k, v : integer;  
    function partition( y, z : integer):integer;  
      var i, j : integer;  
      begin ... a ... v ... exchange(i,j) ... end. {partition};  
    begin ... a ... v ... partition ... quicksort ... end. {quicksort} ;  
  begin ... a ... readarray ... quicksort ... end. {sort};
```

静态作用域
规则

过程

嵌套深度

sort

1

readarray

2

exchange

2

quicksort

2

partition

3

名字的嵌套深度等于声明
名字的过程的嵌套深度

非局部数据的访问

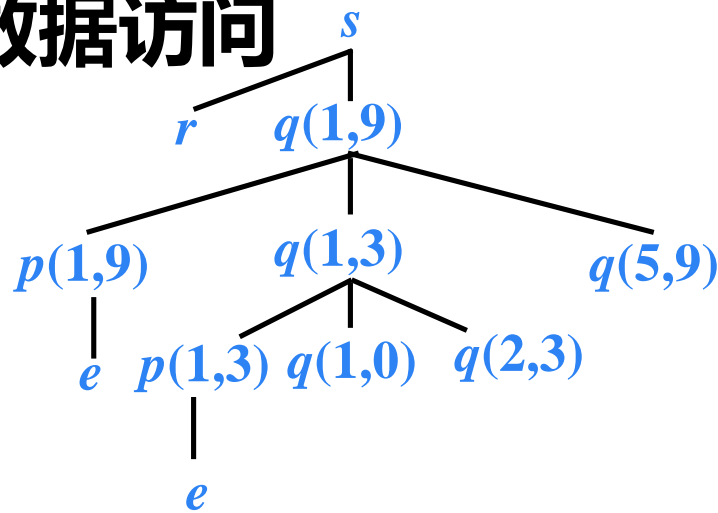
- 一个过程除了可以使用过程自身声明的局部数据以外，还可以使用过程外声明的非局部数据
 - 全局数据
 - 外围过程定义的数据（支持过程嵌套声明的语言）
 - 例：*Pascal*语言 C语言的程序块机制
- 如何访问非局部数据？
 - 核心问题：如何找到被访问的非局部数据所在的活动记录
 - 方法
 - 访问链（静态链）
 - display表（嵌套层次显示表）

访问链 (*Access Links*)

- **静态作用域规则**：只要过程 b 的声明嵌套在过程 a 的声明中，过程 b 就可以访问过程 a 中声明的对象
- **访问链**：可以在相互嵌套的过程的**活动记录之间**建立一种称为**访问链**(*Access link*)的指针，使得**内嵌**的过程可以**通过访问链**访问**外层**过程中声明的对象
- **访问链的指向**：如果过程 b 在源代码中**直接嵌套**在过程 a 中(即 b 的嵌套深度= a 的嵌套深度+1)，那么 b 的任何活动中的**访问链都指向最近的 a 的**活动记录****

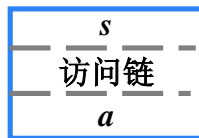
例：基于访问链的非局部数据访问

```
program sort ( input, output );  
  var a: array[0..10] of integer;  
  x: integer;  
  procedure readarray;  
    var i: integer;  
    begin ... a ... end { readarray } ;  
  procedure exchange( i, j: integer);  
    begin x=a[i];a[i]=a[j];a[j]=x; end { exchange } ;  
  procedure quicksort( m, n: integer );  
    var k, v : integer;  
    function partition( y, z: integer): integer;  
      var i, j : integer;  
      begin ... a ... v ... exchange(i,j) ... end { partition } ;  
    begin ... a ... v ... partition ... quicksort ... end { quicksort } ;  
  begin ... a ... readarray ... quicksort ... end { sort } ;
```



基于访问链的 非局部数据访问

控制栈



活动树

s

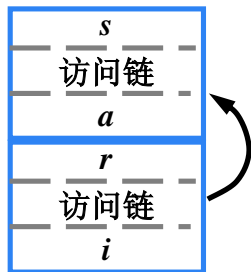
过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

基于访问链的非局部数据访问

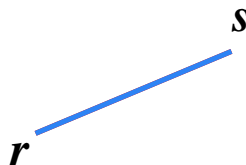
r 中非局部数据 a 的地址表示：
($n_r - n_a$, a 在活动记录中的偏移量)

r 访问非局部数据 a ：
沿 r 访问链经过 $n_r - n_a$ 即1步就能到达 s 的活动记录

控制栈



活动树



过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

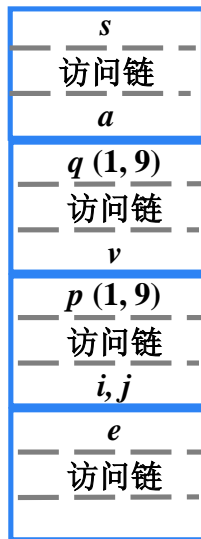
基于访问链的非局部数据访问

p 中非局部数据 a 的地址表示：
($n_p - n_a$, a 在活动记录中的偏移量)

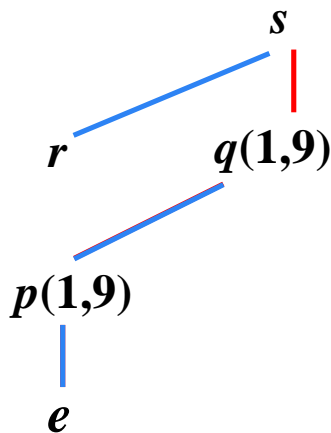
p 访问非局部数据 a ：
沿 p 访问链经过 $n_p - n_a$ 即 2 步就能
到达 s 的活动记录

过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

控制栈



活动树



沿着栈顶活动记录的访问链就可以找到它可以访问的所有活动的数据。

基于访问链的非局部数据访问

e 中 a 的地址表示：

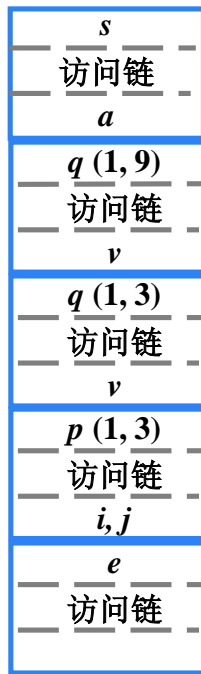
($n_e - n_a$, a 在活动记录中的偏移量)

e 访问非局部数据 a ：

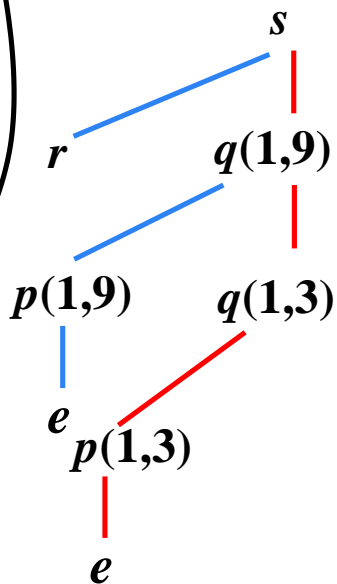
沿 e 访问链经过 $n_e - n_a$ 即1步就能到达 s 的活动记录

过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

控制栈

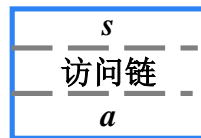


活动树



访问链的建立

控制栈



活动树

s

过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

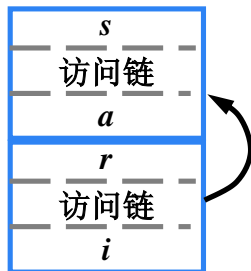
访问链的建立

情况1：嵌套深度 $n_s < n_r$

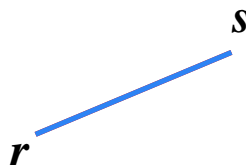
s 一定是直接定义 r

因此， r 的访问链直接指向 s 的活动记录

控制栈



活动树



过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

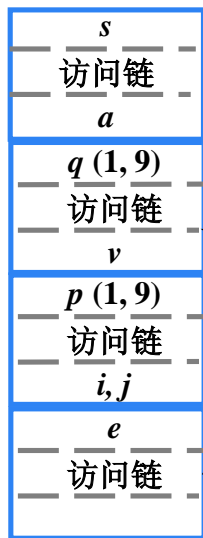
访问链的建立

情况2：嵌套深度 $n_p > n_e$

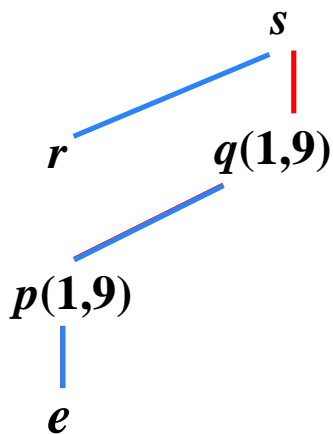
e 的访问链要指向最近的 s 的活动记录， s 也是 p 的外围过程，从 p 沿访问链经过 $n_p - n_e + 1$ 步即到达 s 的活动记录。

过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

控制栈



活动树



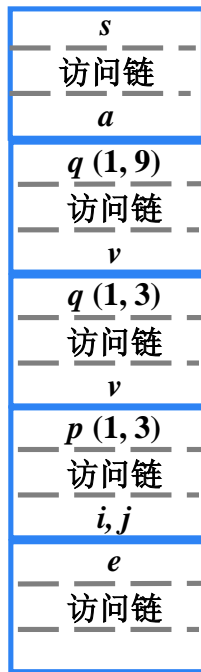
访问链的建立

情况3：嵌套深度 $n_q = n_q$

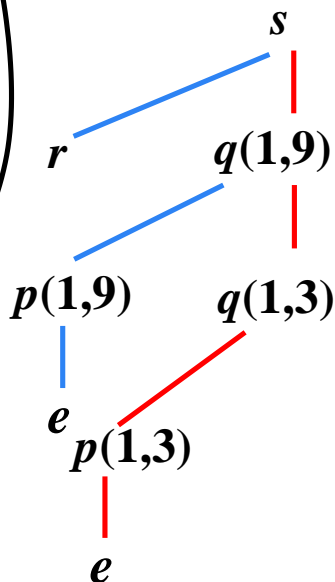
$q(1, 3)$ 的访问链与 $q(1, 9)$ 的访问链指向相同的活动记录。

过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

控制栈

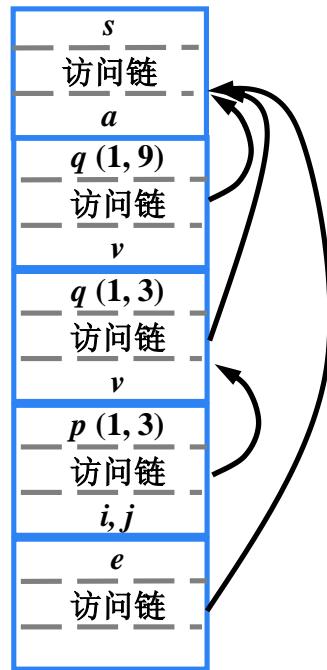


活动树



访问链的建立

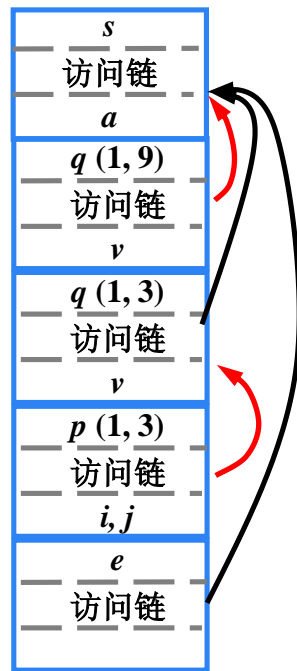
- 建立访问链的代码属于调用序列的一部分
- 假设嵌套深度为 n_x 的过程 x 调用嵌套深度为 n_y 的过程 y ($x \rightarrow y$)
 - $n_x < n_y$ 的情况(外层调用内层)
 - y 一定是直接在 x 中定义的 (例如: $s \rightarrow q$, $q \rightarrow p$), 因此, $n_y = n_x + 1$
 - y 的访问链设置为指向 x 的活动记录的指针



过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

访问链的建立

- 建立访问链的代码属于调用序列的一部分
- 假设嵌套深度为 n_x 的过程 x 调用嵌套深度为 n_y 的过程 y ($x \rightarrow y$)
 - $n_x < n_y$ 的情况(外层调用内层)
 - $n_x = n_y$ 的情况(本层调用本层)
 - 例如：递归调用 (如： $q \rightarrow q$)
 - 被调用者的活动记录的访问链与调用者的活动记录的访问链是相同的，可以直接复制

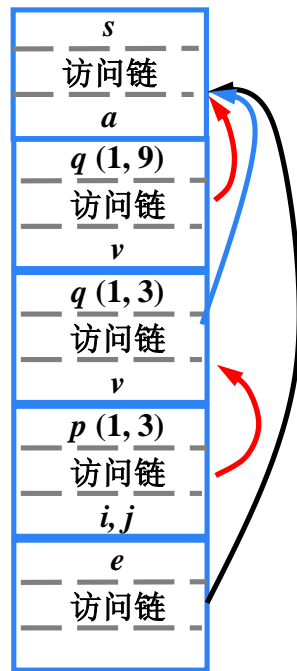


过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

访问链的建立

- 建立访问链的代码属于调用序列的一部分
- 假设嵌套深度为 n_x 的过程 x 调用嵌套深度为 n_y 的过程 y ($x \rightarrow y$)
 - $n_x < n_y$ 的情况(外层调用内层)
 - $n_x = n_y$ 的情况(本层调用本层)
 - $n_x > n_y$ 的情况(内层调用外层，如： $p \rightarrow e$)
 - 调用者 x (如： p)必定嵌套在某个过程 z (如： s)中，而 z (如： s)中直接定义了被调用者 y (如： p)
 - 从 x 的活动记录开始，沿着访问链经过 $n_x - n_y + 1$ 步就可以找到离栈顶最近的 z 的活动记录。 y 的访问链必须指向 z 的这个活动记录

嵌套深度是在编译阶段通过静态分析就能确定的



过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

display表（嵌套层次显示表）

➤ 访问链方法存在的问题

- 访问外层过程名字的效率比较低，需要沿访问链寻找

➤ display表

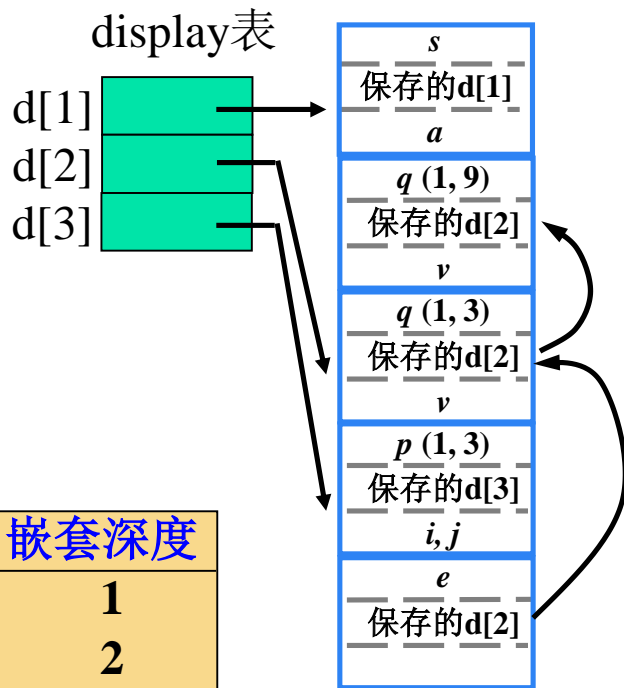
- 是一个指针数组 d ，长度为程序中所有过程嵌套深度(层号)的最大值
- 在任何时刻， $d[i]$ 均指向运行栈中最新建立的嵌套深度为 i 的过程的活动记录（在运行栈中可能存在多个嵌套深度为 i 的过程的活动记录）

➤ 基于display表的非局部数据访问

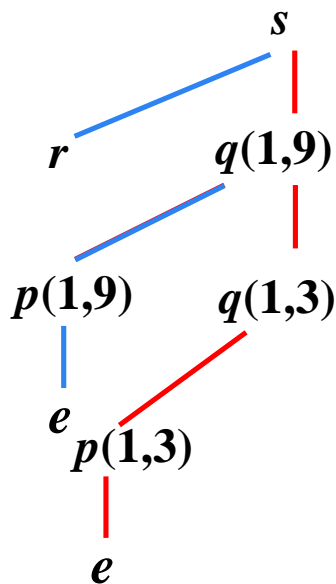
- 如果要访问某个嵌套深度为 i 的非局部名字 x ，只要沿着指针 $d[i]$ 找到 x 所属过程的活动记录，再根据已知的偏移量就可以在活动记录中找到 x ，此时 x 的地址使用如下二元组表示：（嵌套深度，偏移量）

例

控制栈



活动树



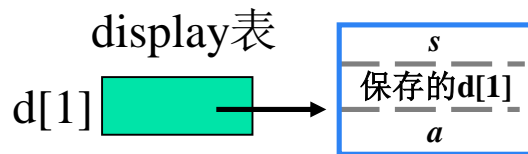
过程	嵌套深度
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

display表的维护

- 每当**开始**一个新活动时都要**修改**display表。
 - 如果嵌套深度为 n_p 的过程 p **被调用**，并且栈中已存在嵌套深度为 n_p 的活动记录，则 p 的活动记录要**保存** $d[n_p]$ **原来的值**，同时置 $d[n_p]$ **指向** p 的这个活动记录
- 而当控制从新活动中**返回**时，又必须**恢复**display表的状态
 - 当 p **返回**且它的活动记录出栈时，再将 $d[n_p]$ **恢复**为调用 p 之前的旧值即 p 活动记录中**保存的** $d[n_p]$
- 如果运行栈中存在**多个嵌套深度为** i **的过程的活动记录**，则通过这些**保存的** $d[i]$ 就将它们**链接**在了一起，但是 $d[i]$ 始终指向**最接近栈顶的**且嵌套深度为 i 的活动记录



控制栈



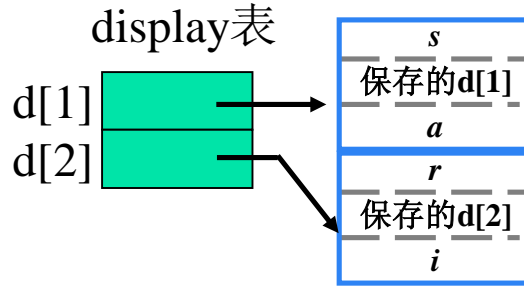
活动树

s

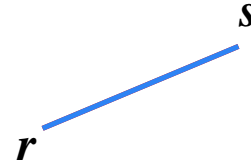
过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3



控制栈

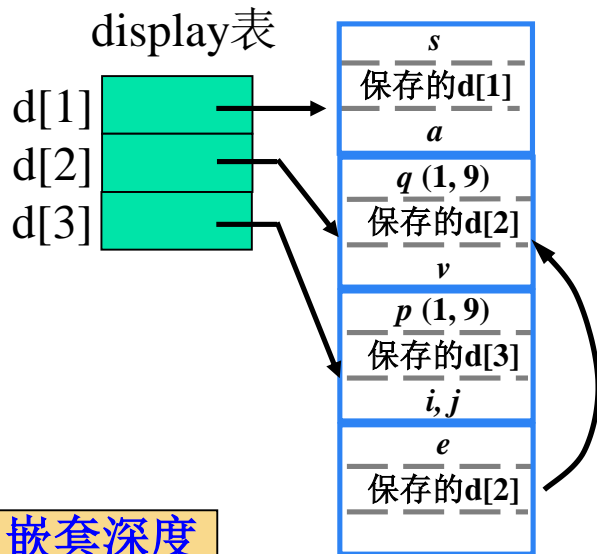


活动树

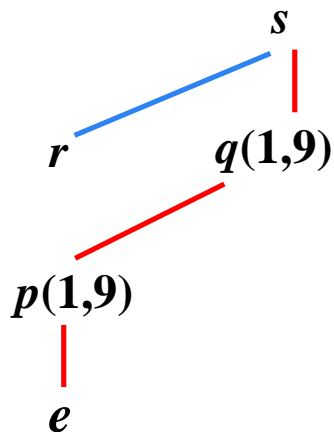


过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

控制栈

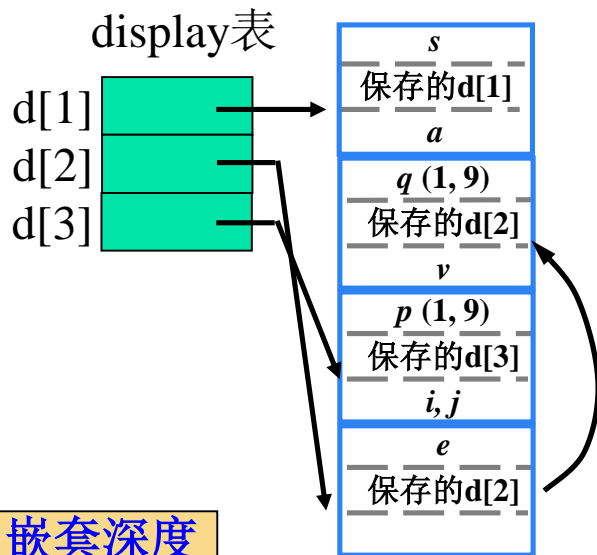


活动树

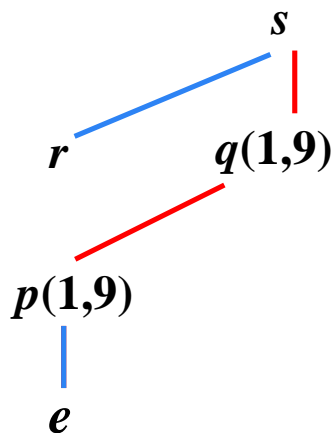


过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

控制栈

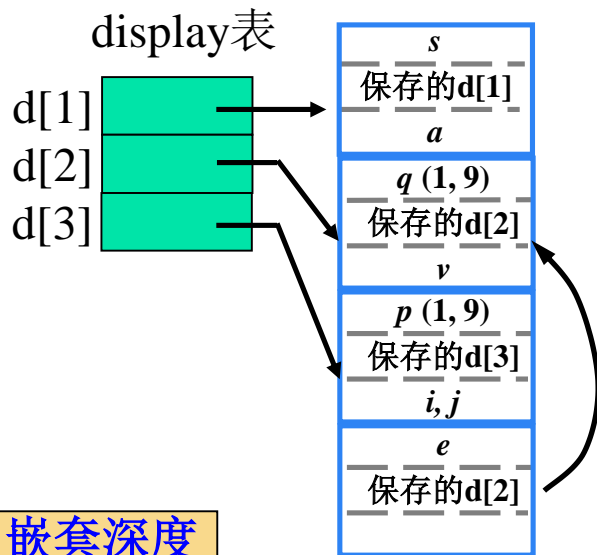


活动树

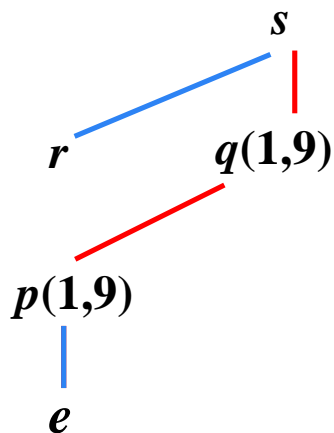


过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

控制栈

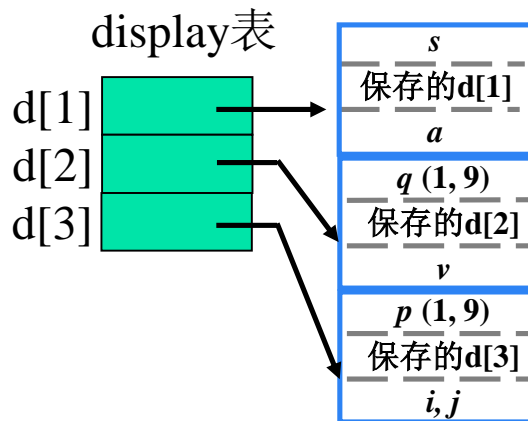


活动树

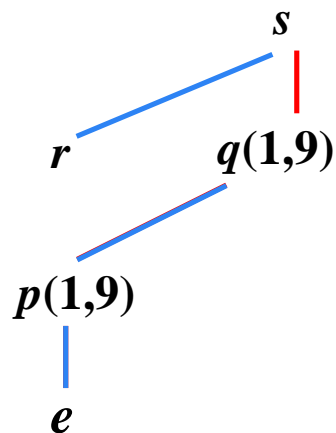


过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

控制栈

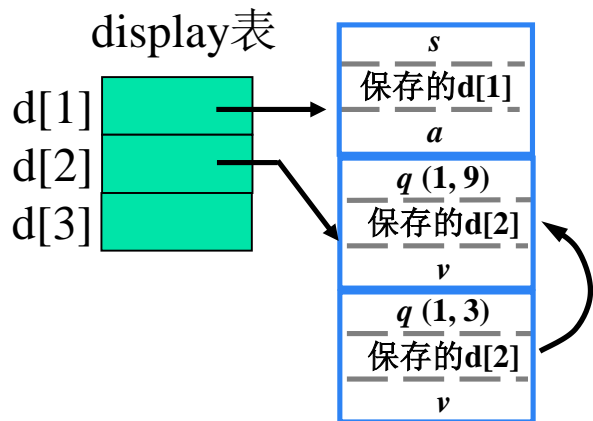


活动树

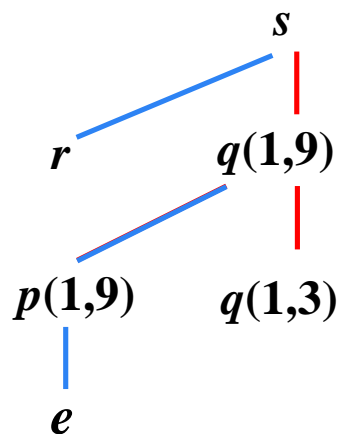


过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

控制栈

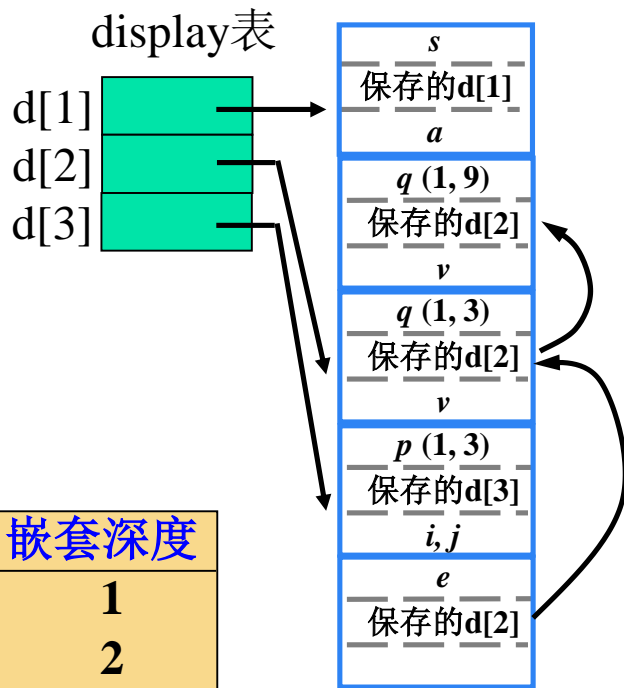


活动树

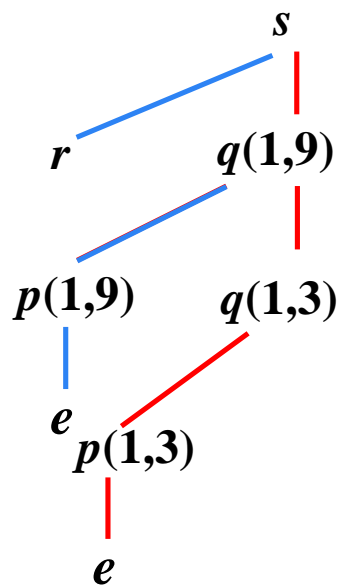


过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

控制栈

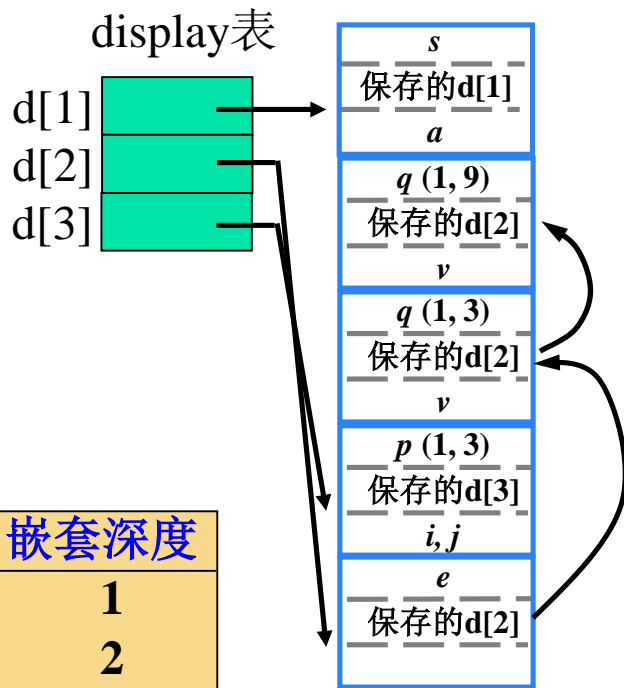


活动树

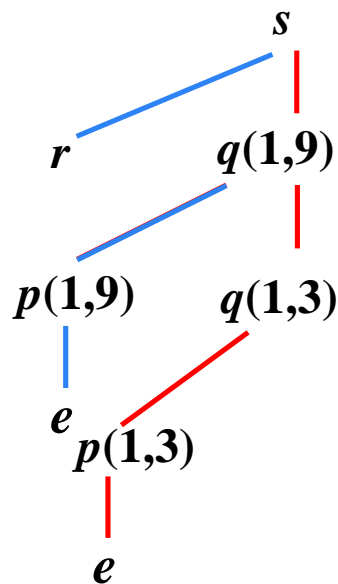


过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

控制栈



活动树



过程	嵌套深度
<i>sort</i>	1
<i>readarray</i>	2
<i>exchange</i>	2
<i>quicksort</i>	2
<i>partition</i>	3

练习

设有如下的 Pascal 程序：

```
program main( input, output) ;  
    var k: integer;  
    function f(n: integer): integer;  
        begin if n <=0 then f:=1 else f:=n* f(n-1) end;  
    begin  
        k:=f(10);  
        writeln(k);  
    end
```

- (1) 若采用访问链存取非局部名字，当第3次(递归地)进入后，试绘制运行栈中的活动记录示意图(至少标示活动名称，画出访问链和控制链及其指向的活动记录)。
- (2) 若采用 display 表方式实现非局部数据的访问，参考教材图7-14重做(1)。



提纲

7.1 存储组织

7.2 静态存储分配

7.3 栈式存储分配

7.4 非局部数据的访问

7.5 参数传递

7.6 符号表

7.5 参数传递（自学）

- 形式参数（formal parameter）
 - 在过程**定义**中使用的参数
- 实际参数（actual parameter）
 - 在**调用**过程时使用的参数
- 形参和实参相关联的几种方法
 - 传值（Call- by-Value）
 - 传地址（Call- by-Reference）
 - 传值结果（Call- by-Value-Result）
 - 传名（Call- by-Name）

7.5.1 传值 (Call- by-Value)

- 把实参的**值**传递给相应的形参
- 实现方法
 - 调用过程把**实参**的**值**计算出来，并传递到**被调用过程**相应的**形式单元**中
 - **被调用过程**中，像引用**局部数据**一样引用**形式参数**，**直接访问**对应的**形式单元**
- 例：Pascal的**值**参数、C/C++的缺省参数传递

例

...

procedure P(w,x,y,z);

begin

y:=y*w;

z:=z+x;

end

begin

a:=5;

b:=3;

P(a+b,a-b,a,a);

write(a);

end

w 8

x 2

y 5

z 5

a 5

b 3

t₁ 8

t₂ 2

例

...

procedure P(w,x,y,z);

begin

y:=y*w;

z:=z+x;

end

begin

a:=5;

b:=3;

P(a+b,a-b,a,a);

write(a);

end

w 8

x 2

y 40

z 5

a 5

b 3

t₁ 8

t₂ 2

例

...

procedure P(w,x,y,z);

begin

y:=y*w;

z:=z+x;

end

begin

a:=5;

b:=3;

P(a+b,a-b,a,a);

write(a);

end

w

8

x

2

y

40

z

7

a

5

b

3

t₁

8

t₂

2

7.5.2 传地址 (Call- by-Reference)

- 把实参的**地址**传递给相应的形参
- 实现方法
 - 调用过程把**实参的地址**传递到**被调用过程**相应的**形式单元**中
 - **被调用过程**中，对**形参**的引用或赋值被处理成对**形式单元**的间接访问
- 例：Pascal的**var**参数、C/C++的传**引用**

例

...

procedure P(w,x,y,z);

begin

 y:=y*w;

 z:=z+x;

end

begin

 a:=5;

 b:=3;

 P(a+b, a-b, a, a);

 write(a);

end

w t₁的地址

x t₂的地址

y a的地址

z a的地址

a 5

b 3

t₁ 8

t₂ 2

例

...

procedure P(w,x,y,z);

begin

 y:=y*w;

 z:=z+x;

end

begin

 a:=5;

 b:=3;

 P(a+b, a-b, a, a);

 write(a);

end

w t₁的地址

x t₂的地址

y a的地址

z a的地址

a 40

b 3

t₁ 8

t₂ 2

例

...

procedure P(w,x,y,z);

begin

 y:=y*w;

 z:=z+x;

end

begin

 a:=5;

 b:=3;

 P(a+b, a-b, a, a);

 write(a);

end

w t₁的地址

x t₂的地址

y a的地址

z a的地址

a 42

b 3

t₁ 8

t₂ 2

7.5.3 传值结果 (Call- by-Value-Result)

- 传地址的一种变形
- 实现方法
 - 每个形参对应两个形式单元。第一个形式单元存放实参的地址，第二个形式单元存放实参的值
 - 在过程体中，对形参的引用或赋值看作对它的第二个形式单元的直接访问
 - 过程完成返回前，把第二个单元的内容存放到第一个单元所指的实参单元中
- 例：Fortune

例

...

procedure P(w,x,y,z);

begin

 y:=y*w;

 z:=z+x;

end

begin

 a:=5;

 b:=3;

 P(a+b,a-b,a,a);

 write(a);

end

w	t ₁ 的地址	8
x	t ₂ 的地址	2
y	a的地址	5
z	a的地址	5
a	5	
b	3	
t ₁	8	
t ₂	2	

例

...

procedure P(w,x,y,z);

begin

 y:=y*w;

 z:=z+x;

end

begin

 a:=5;

 b:=3;

 P(a+b,a-b,a,a);

 write(a);

end

w	t ₁ 的地址	8
x	t ₂ 的地址	2
y	a的地址	40
z	a的地址	5
a	5	
b	3	
t ₁	8	
t ₂	2	

例

...

procedure P(w,x,y,z);

begin

 y:=y*w;

 z:=z+x;

end

begin

 a:=5;

 b:=3;

 P(a+b,a-b,a,a);

 write(a);

end

w	t ₁ 的地址	8
x	t ₂ 的地址	2
y	a的地址	40
z	a的地址	7
a	5	
b	3	
t ₁	8	
t ₂	2	

例

...

procedure P(w,x,y,z);

begin

 y:=y*w;

 z:=z+x;

end

begin

 a:=5;

 b:=3;

 P(a+b,a-b,a,a);

 write(a);

end

w	t ₁ 的地址	8
x	t ₂ 的地址	2
y	a的地址	40
z	a的地址	7
a	5	
b	3	
t ₁	8	
t ₂	2	

例

...

procedure P(w,x,y,z);

begin

 y:=y*w;

 z:=z+x;

end

begin

 a:=5;

 b:=3;

 P(a+b,a-b,a,a);

 write(a);

end

w	t ₁ 的地址	8
x	t ₂ 的地址	2
y	a的地址	40
z	a的地址	7
a	5	
b	3	
t ₁	8	
t ₂	2	

例

...

procedure P(w,x,y,z);

begin

 y:=y*w;

 z:=z+x;

end

begin

 a:=5;

 b:=3;

 P(a+b,a-b,a,a);

 write(a);

end

w	t ₁ 的地址	8
x	t ₂ 的地址	2
y	a的地址	40
z	a的地址	7
a	40	
b	3	
t ₁	8	
t ₂	2	

例

...

procedure P(w,x,y,z);

begin

 y:=y*w;

 z:=z+x;

end

begin

 a:=5;

 b:=3;

 P(a+b,a-b,a,a);

 write(a);

end

w	t ₁ 的地址	8
x	t ₂ 的地址	2
y	a的地址	40
z	a的地址	7
a	7	
b	3	
t ₁	8	
t ₂	2	

7.5.4 传名 (Call- by-Name)

- 相当于把被调用过程的过程体抄到调用出现的地方，但把其中出现的形参都替换成相应的实参
- 实现方法
 - 在进入被调用过程之前不对实参预先进行计值，而是让过程体中每当使用到相应的实参时才逐次对它实行计值（或计算地址）
 - 通常把实参处理成一个子程序（称为参数子程序），每当过程体中使用到相应的实参时就调用这个子程序
- 例：ALGOL60

例

...

procedure P(w,x,y,z);

begin

y:=y*w;

z:=z+x;

end

begin

a:=5;

b:=3;

P(a+b,a-b,a,a);

write(a);

end

a

5

b

3

例

...

procedure P(w,x,y,z);

begin

 y:=y*w;

 z:=z+x;

end

begin

 a:=5;

 b:=3;

 P(a+b,a-b,a,a);

 write(a);

end



begin

 a:=5;

 b:=3;

 a:=a*(a+b);

 a:=a+(a-b);

 write(a);

end

a	5
b	3

例

...

procedure P(w,x,y,z);

begin

y:=y*w;

z:=z+x;

end

begin

a:=5;

b:=3;

P(a+b,a-b,a,a);

write(a);

end



begin

a:=5;

b:=3;

a:=a*(a+b);

a:=a+(a-b);

write(a);

end

a	40
b	3

例

...

procedure P(w,x,y,z);

begin

y:=y*w;

z:=z+x;

end

begin

a:=5;

b:=3;

P(a+b,a-b,a,a);

write(a);

end



begin

a:=5;

b:=3;

a:=a*(a+b);

a:=a+(a-b);

write(a);

end

a	77
b	3

传名方式中的重命名问题

PROGRAM EX

...

var **A**: integer;

PROCEDURE P(**B**: integer)

...

var **A**: integer;

BEGIN

A:=0;

B:=**B**+1;

A:=**A**+**B**;

END

BEGIN

A:=2;

P(**A**);

write(**A**);

END



BEGIN

A:=2;

A:=0;

B:=**B**+1;

A:=**A**+**B**;

write(**A**);

END



BEGIN

A:=2;

A:=0;

A:=**A**+1;

A:=**A**+**A**;

write(**A**);

END



BEGIN

A:=2;

TA:=0;

A:=**A**+1;

TA:=**TA**+**A**;

write(**A**);

END

参数传递方式对比

...

procedure P(w,x,y,z);

begin

y:=y*w;

z:=z+x;

end

begin

a:=5;

b:=3;

P(a+b,a-b,a,a);

write(a);

end

- 传值 : 5
- 传地址 : 42
- 传值结果 : 7
- 传名 : 77



提纲

7.1 存储组织

7.2 静态存储分配

7.3 栈式存储分配

7.4 非局部数据的访问

7.5 参数传递

7.6 符号表

7.6 符号表

➤ 符号表是用于存放标识符的属性信息的数据结构

➤ 种属 (Kind)

➤ 类型 (Type)

➤ 存储位置、长度

➤ 作用域

➤ 参数和返回值信息

➤ 符号表的作用

➤ 辅助代码生成

➤ 一致性检查

NAME	TYPE	KIND	VAL	ADDR
SIMPLE	整	简变		
SYMBLE	实	数组		
TABLE	字符	常数		
⋮	⋮	⋮	⋮	⋮

符号表上的主要操作

- 声明语句的翻译（定义性出现）
 - 填、查
- 可执行语句的翻译（使用性出现）
 - 查

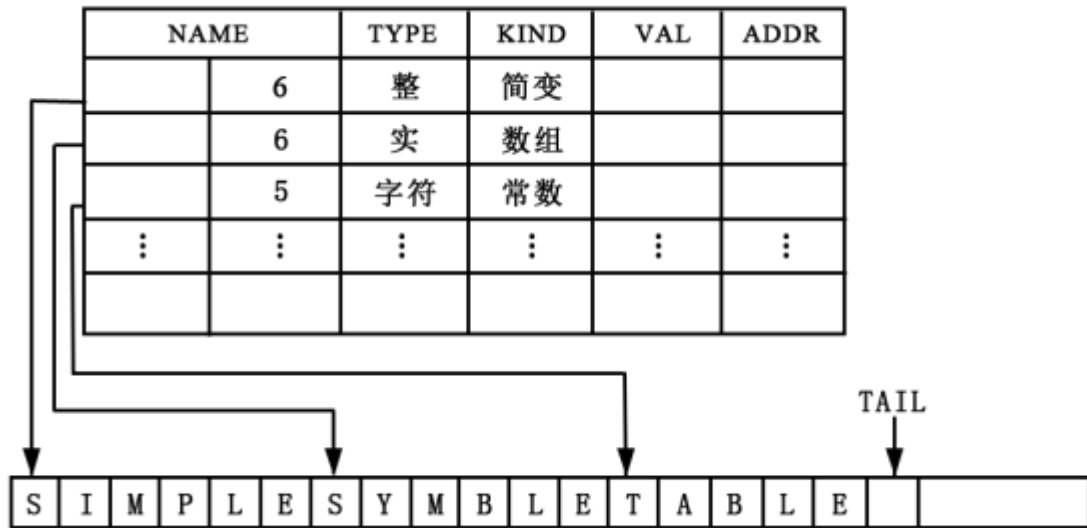
单个过程符号表的组织

➤ 方法一：一张大表

➤ 问题：不同**种属**的名字
所需存放的属性信息在
数量上的差异会造成符号表**空间的浪费**

➤ 数组名：数组的维数、
各维的长度

➤ 过程名：参数个数、参
数类型、返回值类型



单个过程符号表的组织

➤ 方法一：一张大表

➤ 问题：不同**种属**的名字所需存放的**属性信息**在**数量上的差异**会造成符号表**空间的浪费**

➤ 方法二：多张子表（按**种属**分）

➤ 变量表、数组表、过程表、...

➤ 问题：为避免**重名**问题，**插入或查找**某个符号时需要查看所有的符号表，从而造成**时间上的浪费**

➤ 解决办法

➤ **基本属性**（直接存放在符号表中）+ **扩展属性**（动态申请内存）

例

基本属性（直接存放在符号表中）+扩展属性（动态申请内存）

```
int abc;
```

```
int i;
```

```
char myarray[3][4];
```

	名字		基本属性		扩展属性				
	符号种类	类型	地址	扩展属性指针	维数	各维维长			
符号表表项 1	abc	变量	int	0	NULL				
符号表表项 2	i	变量	int	4	NULL				
符号表表项 3	myarray	数组	int	8		<table><tr><td>2</td><td>3</td><td>4</td></tr></table>	2	3	4
2	3	4							
...				...					

各维的width

多个过程符号表的组织

➤ 需要考虑的问题

假设过程 p 要访问过程 q 中的数据对象 x ,
 x 的地址 = q 的活动记录基地址 + x 在 q 的活动记录中的偏移地址

➤ 刻画过程之间的嵌套关系（作用域信息）

➤ 重名问题：不同作用域可以重名

➤ 常用的组织方式

➤ 每个过程建立一个符号表，同时需要建立起这些符号表之间的联系，用来刻画过程之间的嵌套关系

例

```
program sort ( input, output );  
  var a: array[0..10] of integer;
```

```
  x: integer;
```

```
  procedure readarray;
```

```
    var i: integer;
```

```
    begin ... a ... end {readarray} ;
```

```
  procedure exchange(i,j:integer);
```

```
    begin x=a[i];a[i]=a[j];a[j]=x; end {exchange} ;
```

```
  procedure quicksort(m, n:integer);
```

```
    var k, v : integer;
```

```
    function partition(y, z:integer):integer;
```

```
      var i, j : integer;
```

```
      begin ... a ... v ... exchange(i,j) ... end {partition};
```

```
      begin ... a ... v ... partition ... quicksort ... end {quicksort};
```

```
  begin ... a ... readarray ... quicksort ... end {sort};
```

sort			
	header		48
a	array		0
x	int		44
readarray			
exchange			
quicksort			

readarray

	header	4
i	int	0

exchange

	header	0
--	--------	---

quicksort

	header	8
k	int	0
v	int	4
partition		

partition

	header	8
i	int	0
j	int	4

符号表的建立

- 嵌套过程声明语句的文法

$$P \rightarrow D$$
$$D \rightarrow D \ D \mid \text{proc id} ; D \ S \mid \text{id} : T ;$$

- 在允许嵌套过程声明的语言中，局部于每个过程的名字可以使用第6章介绍的方法分配相对地址；
- 当看到嵌套的过程 p 时，应暂时挂起对外围过程 q 声明语句的翻译，转而翻译过程 p 的声明语句。 p 处理完后，紧接着翻译过程 q 的剩余代码。
- 过程定义亦满足嵌套性，因此可以使用栈来管理不同过程的符号表指针

嵌套过程声明语句的SDT

*tblptr*栈: 未处理完过程的符号表的指针

*offset*栈: 未处理完过程的可用偏移地址

$P \rightarrow M D \{ \text{addwidth}(\text{top}(\text{tblptr}), \text{top}(\text{offset}));$

$\text{pop}(\text{tblptr});$

$\text{pop}(\text{offset}); \}$

$M \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{nil});$

$\text{push}(t, \text{tblptr});$

$\text{push}(0, \text{offset}); \}$

mktable(*previous*)

创建一个新符号表，其父过程指针为*previous*，并返回指向新表的指针。

$D \rightarrow D_1 D_2$

$D_p \rightarrow \text{proc id} ; N D_1 S \{ t = \text{top}(\text{tblptr});$

$\text{addwidth}(t, \text{top}(\text{offset}));$

$\text{pop}(\text{tblptr});$

$\text{pop}(\text{offset});$

$\text{enterproc}(\text{top}(\text{tblptr}), \text{id.lexeme}, t); \}$

addwidth(*table*, *width*)

将符号表中所有局部数据宽度之和*width*记录在*table*指向的符号表表头中

enterproc(*table*, *name*, *newtable*)

在*table*指向的符号表中为过程*name*创建一条过程记录，*newtable*指向过程*name*的符号表

$D_v \rightarrow \text{id} : T ; \{ \text{enter}(\text{top}(\text{tblptr}), \text{id.lexeme}, T.\text{type}, \text{top}(\text{offset}));$

$\text{top}(\text{offset}) = \text{top}(\text{offset}) + T.\text{width}; \}$

$N \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{top}(\text{tblptr}));$

$\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$

enter(*table*, *name*, *type*, *offset*)

在*table*指向的符号表中为名字*name*建立一个新表项，类型为*type*，相对地址为*offset*

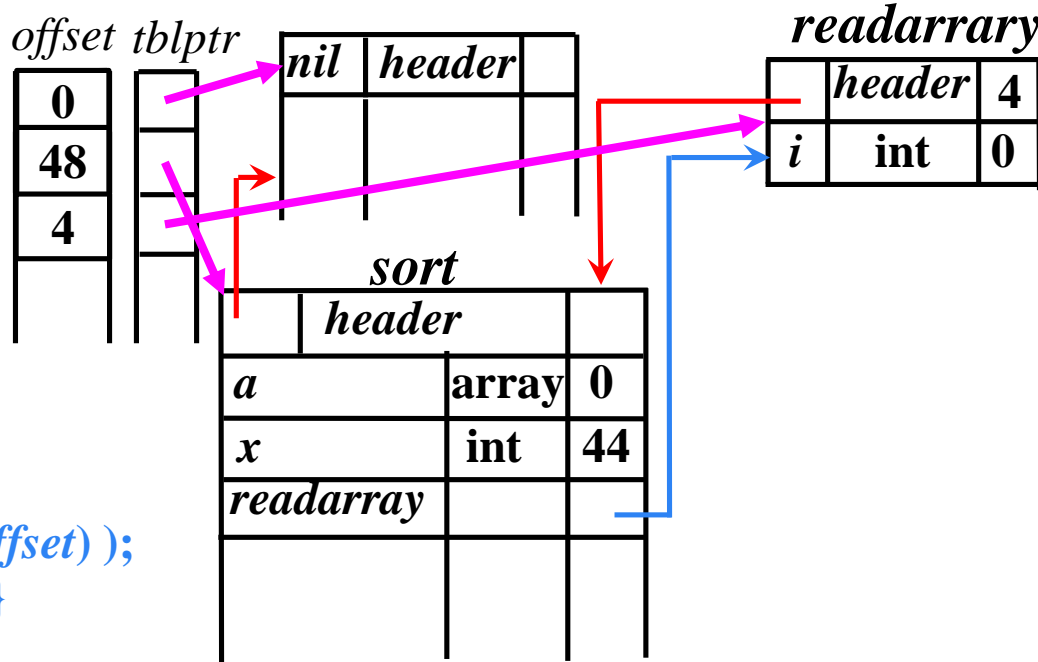
例

```
program sort ( input, output );  
  var a: array[0..10] of integer;  
      x: integer;  
  procedure readarray;  
    var i: integer;  
    begin ... a ... end {readarray} ;  
  procedure exchange(i,j:integer);  
    begin x=a[i];a[i]=a[j];a[j]=x; end {exchange} ;  
  procedure quicksort(m, n:integer);  
    var k, v : integer;  
    function partition(y, z:integer):integer;  
      var i, j : integer;  
      begin ... a ... v ... exchange(i,j) ... end {partition};  
    begin ... a ... v ... partition ... quicksort ... end {quicksort} ;  
  begin ... a ... readarray ... quicksort ... end {sort};
```

```
program sort;  
  var a:int[11]; x:int;  
  proc readarray;  
    var i:int; {...}  
  proc exchange; {...}  
  proc quicksort;  
    var k, v:int;  
    func partition;  
      var i, j:int; {...}  
    {...}  
    {...}
```

例

```
program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange; {...}
  proc quicksort;
    var k,v:int;
    func partition;
      var i,j:int; {...}
    {...}
  {...}
```



$P \rightarrow M D \{ \text{addwidth}(\text{top}(\text{tblptr}), \text{top}(\text{offset}));$
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \}$

$M \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{nil});$
 $\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$

$D \rightarrow D_1 D_2$

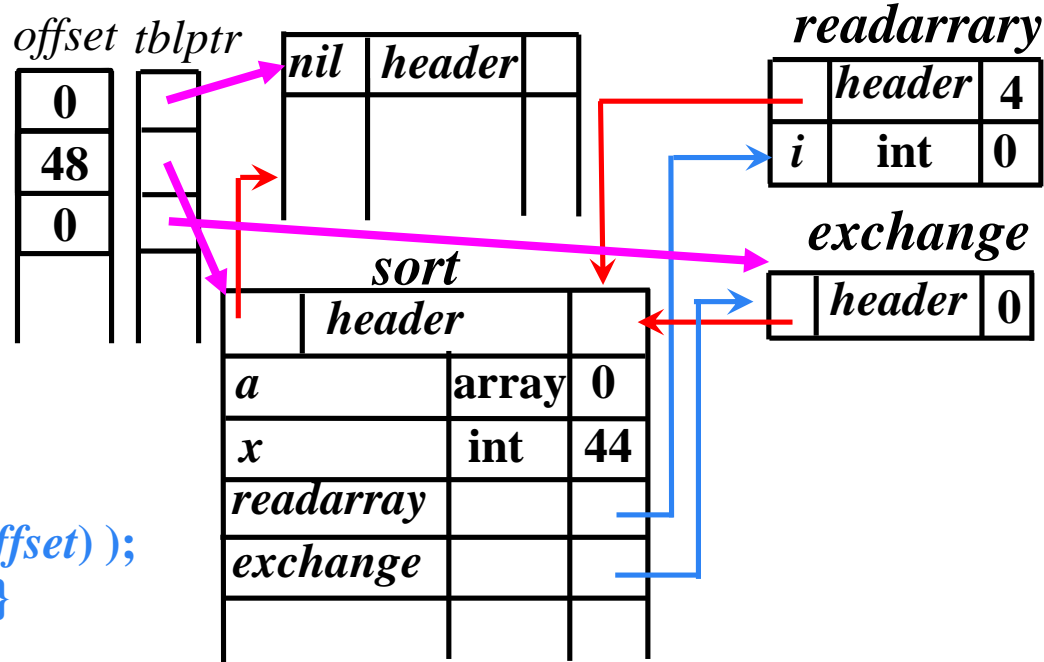
$D_p \rightarrow \text{proc id} ; N D_1 S \{ t = \text{top}(\text{tblptr}); \text{addwidth}(t, \text{top}(\text{offset}));$
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \text{enterproc}(\text{top}(\text{tblptr}), \text{id.lexeme}, t); \}$ //将过程记入符号表

$D_v \rightarrow \text{id}: T ; \{ \text{enter}(\text{top}(\text{tblptr}), \text{id.lexeme}, T.\text{type}, \text{top}(\text{offset})); \}$ //向栈顶的符号表添加表项
 $\text{top}(\text{offset}) = \text{top}(\text{offset}) + T.\text{width}; \}$

$N \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{top}(\text{tblptr})); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$ //新建符号表

例

```
program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange; {...}
  proc quicksort;
    var k,v:int;
    func partition;
      var i,j:int; {...}
    {...}
  {...}
```



$P \rightarrow M D \{ \text{addwidth}(\text{top}(\text{tblptr}), \text{top}(\text{offset}));$
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \}$

$M \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{nil});$
 $\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$

$D \rightarrow D_1 D_2$

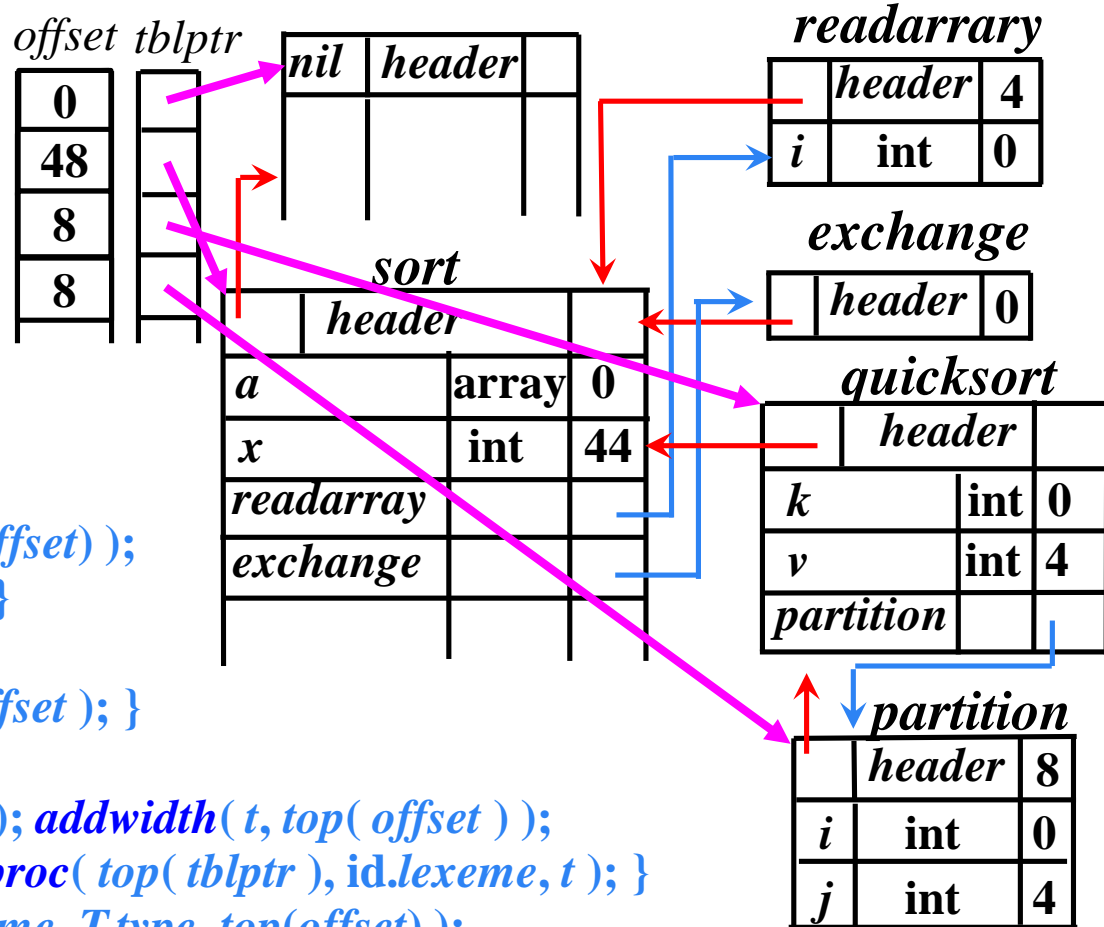
$D_p \rightarrow \text{proc id}; N D_1 S \{ t = \text{top}(\text{tblptr}); \text{addwidth}(t, \text{top}(\text{offset}));$
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \text{enterproc}(\text{top}(\text{tblptr}), \text{id.lexeme}, t); \}$ //将过程记入符号表

$D_v \rightarrow \text{id}: T; \{ \text{enter}(\text{top}(\text{tblptr}), \text{id.lexeme}, T.\text{type}, \text{top}(\text{offset})); \}$ //向栈顶的符号表添加表项
 $\text{top}(\text{offset}) = \text{top}(\text{offset}) + T.\text{width}; \}$

$N \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{top}(\text{tblptr})); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$ //新建符号表

例

```
program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange; {...}
  proc quicksort;
    var k,v:int;
    func partition;
      var i,j:int; {...}
    {...}
  {...}
```



$P \rightarrow M D \{ \text{addwidth}(\text{top}(\text{tblptr}), \text{top}(\text{offset}));$
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \}$

$M \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{nil});$
 $\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$

$D \rightarrow D_1 D_2$

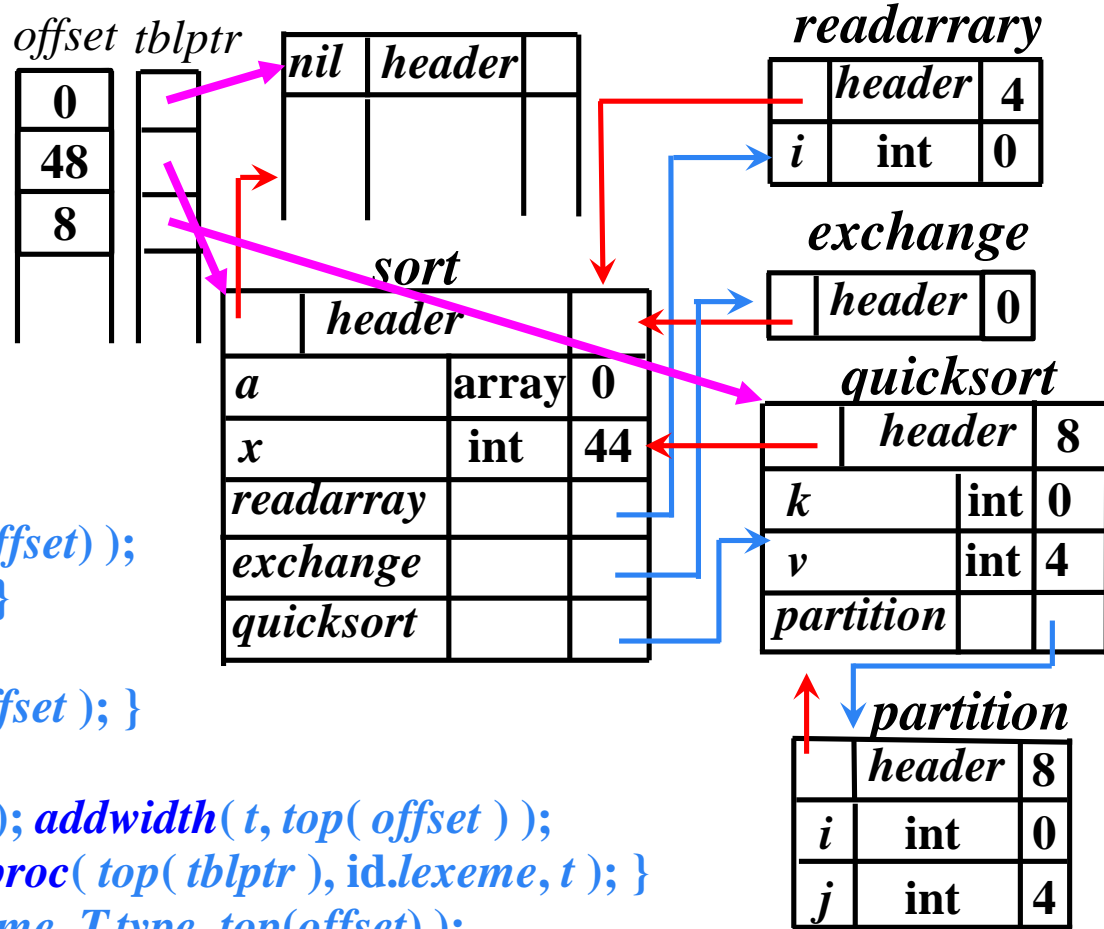
$D_p \rightarrow \text{proc id}; N D_1 S \{ t = \text{top}(\text{tblptr}); \text{addwidth}(t, \text{top}(\text{offset}));$
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \text{enterproc}(\text{top}(\text{tblptr}), \text{id.lexeme}, t); \}$

$D_v \rightarrow \text{id}: T; \{ \text{enter}(\text{top}(\text{tblptr}), \text{id.lexeme}, T.\text{type}, \text{top}(\text{offset}));$
 $\text{top}(\text{offset}) = \text{top}(\text{offset}) + T.\text{width}; \}$

$N \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{top}(\text{tblptr})); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$

例

```
program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange; {...}
  proc quicksort;
    var k,v:int;
    func partition;
      var i,j:int; {...}
    {...}
  {...}
```



$P \rightarrow M D \{ \text{addwidth}(\text{top}(\text{tblptr}), \text{top}(\text{offset}));$
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \}$

$M \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{nil});$
 $\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$

$D \rightarrow D_1 D_2$

$D_p \rightarrow \text{proc id}; N D_1 S \{ t = \text{top}(\text{tblptr}); \text{addwidth}(t, \text{top}(\text{offset}));$
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \text{enterproc}(\text{top}(\text{tblptr}), \text{id.lexeme}, t); \}$

$D_v \rightarrow \text{id}: T; \{ \text{enter}(\text{top}(\text{tblptr}), \text{id.lexeme}, T.\text{type}, \text{top}(\text{offset}));$
 $\text{top}(\text{offset}) = \text{top}(\text{offset}) + T.\text{width}; \}$

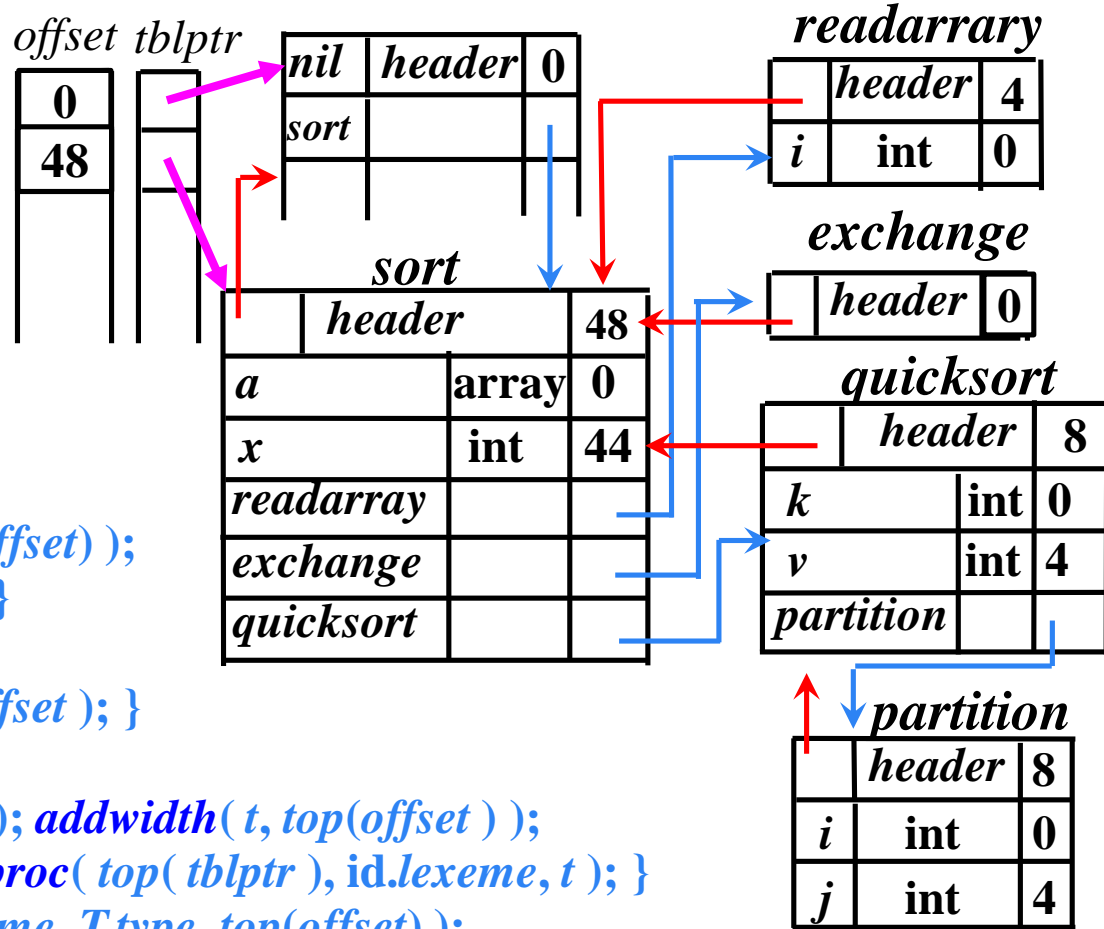
$N \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{top}(\text{tblptr})); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange; {...}
  proc quicksort;
    var k,v:int;
    func partition;
      var i,j:int; {...}
      {...}
      {...}

```



$P \rightarrow M D \{ \text{addwidth}(\text{top}(\text{tblptr}), \text{top}(\text{offset}));$
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \}$

$M \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{nil});$
 $\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$

$D \rightarrow D_1 D_2$

$D_p \rightarrow \text{proc id}; N D_1 S \{ t = \text{top}(\text{tblptr}); \text{addwidth}(t, \text{top}(\text{offset}));$
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \text{enterproc}(\text{top}(\text{tblptr}), \text{id.lexeme}, t); \}$

$D_v \rightarrow \text{id}: T; \{ \text{enter}(\text{top}(\text{tblptr}), \text{id.lexeme}, T.\text{type}, \text{top}(\text{offset}));$
 $\text{top}(\text{offset}) = \text{top}(\text{offset}) + T.\text{width}; \}$

$N \rightarrow \varepsilon \quad \{ t = \text{mktable}(\text{top}(\text{tblptr})); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$

标识符的基本处理方法

- 当在某一层的**声明语句**中识别出一个标识符(id的**定义性出现**)时，以此标识符查找相应于**本层的符号表**
 - 如果查到，则报错并发出诊断信息“id**重复声明**”
 - 否则，在符号表中加入新登记项，将标识符及有关信息填入
- 当在**可执行语句**部分扫视到标识符时(id的**应用性出现**)
 - 首先在**该层符号表**中查找该id，如果找不到，则到**直接外层符号表**中去查，如此等等，一旦找到，则在表中取出有关信息并作相应处理 将层号(或嵌套深度之差)作为中间代码中地址的一部分
 - 如果查遍所有外层符号表均未找到该id，则报错并发出诊断信息“id**未声明**”

例

```
program sort ( input, output );
  var a: array[0..10] of integer;
```

```
  x: integer;
```

```
  procedure readarray;
```

```
    var i: integer;
```

```
    begin ... a ... end {readarray} ;
```

```
  procedure exchange(i,j:integer);
```

```
    begin x=a[i];a[i]=a[j];a[j]=x; end {exchange} ;
```

```
  procedure quicksort(m, n:integer);
```

```
    var k, v : integer;
```

```
    function partition(y, z:integer):integer;
```

```
      var i, j : integer;
```

```
      begin ... a ... v ... exchange(i,j) ... end {partition};
```

```
      begin ... a ... v ... partition ... quicksort ... end {quicksort} ;
```

```
  begin ... a ... readarray ... quicksort ... end {sort};
```

sort			
	header		48
a	array		0
x	int		44
readarray			
exchange			
quicksort			

readarray			
	header		4
i	int		0

exchange			
	header		0

quicksort			
	header		8
k	int		0
v	int		4
partition			

partition			
	header		8
i	int		0
j	int		4

[a(1,0)]

[v(2,4)]

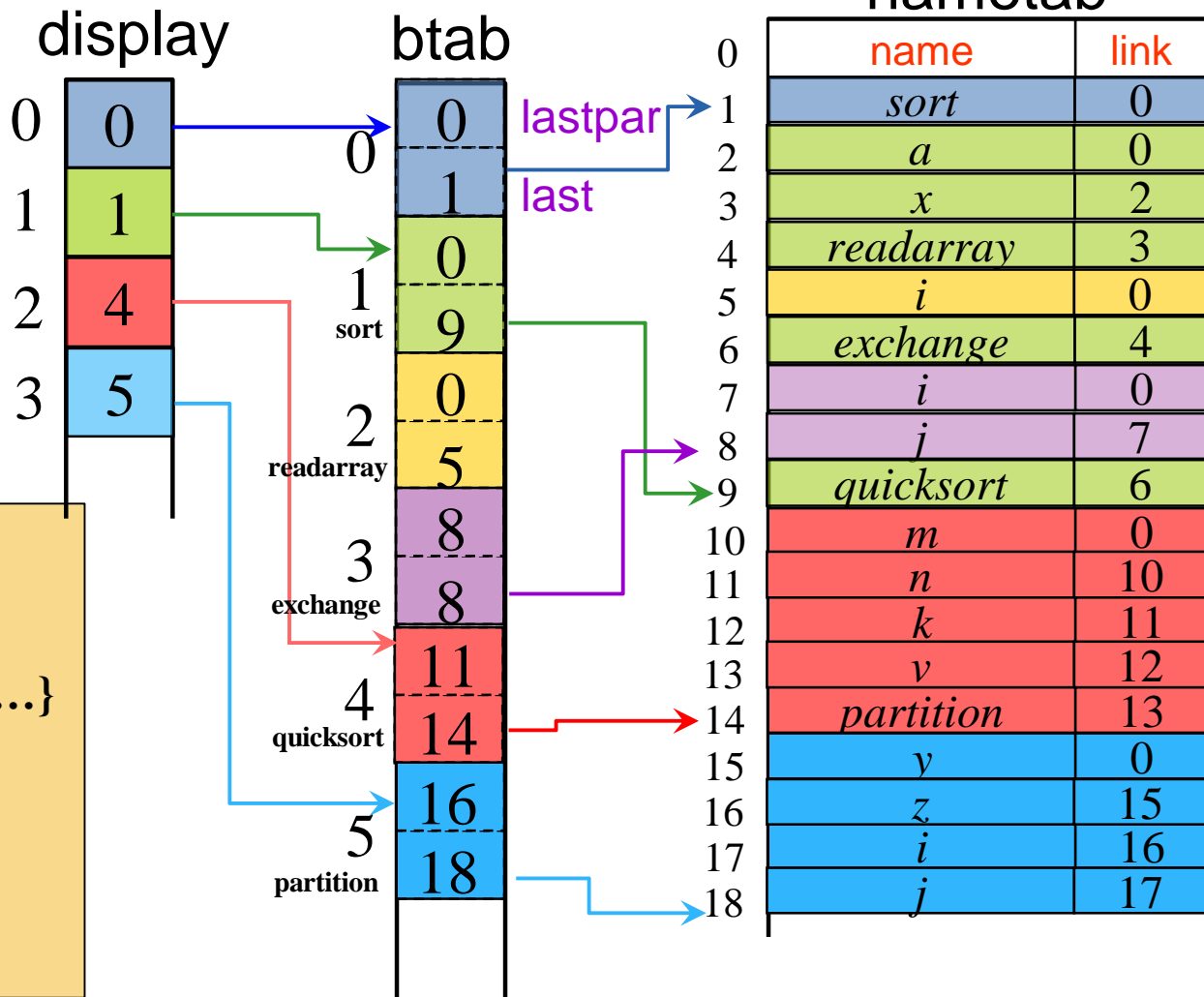
符号表的另一种组织方式*

- 将所有块^①的符号表放在一个大数组^②中，然后再引入一个块表^③来描述各块^④的符号表在大数组^⑤中的位置及其相互关系
- 一个过程^⑥可以看作是一个块^⑦

例

- **lastpar** 指向本过程体中最后一个形参在 **nametab** 中的位置
- **last** 指向本过程体中最后一个名字在 **nametab** 中的位置

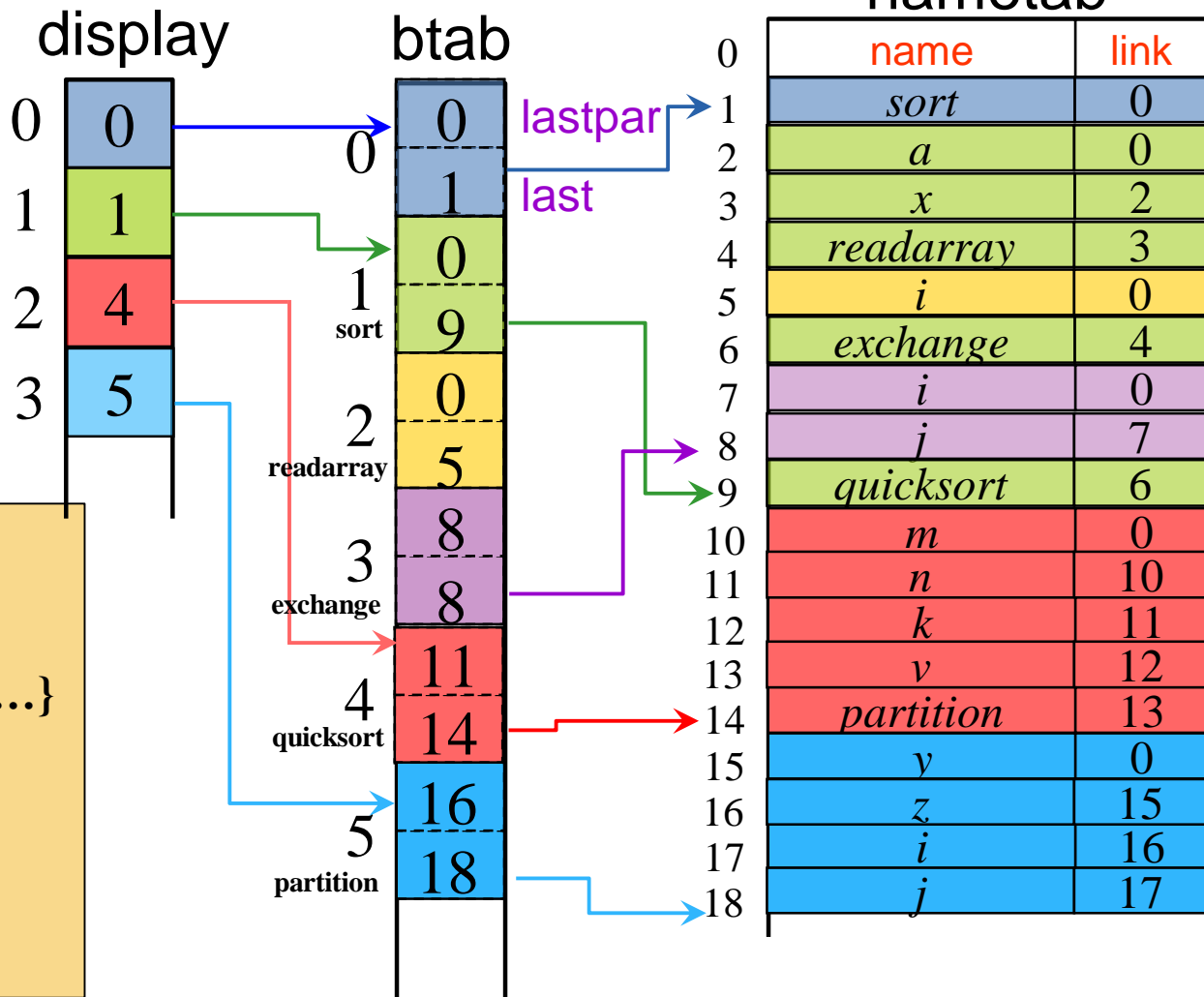
```
program sort;  
  var a:int[11]; x:int;  
  proc readarray;  
    var i:int; {...}  
  proc exchange (i,j:integer); {...}  
  proc quicksort(m, n:integer);  
    var k, v:int;  
    func partition(y, z:integer);  
      var i, j:int; {...}  
      {...}  
      {...}
```



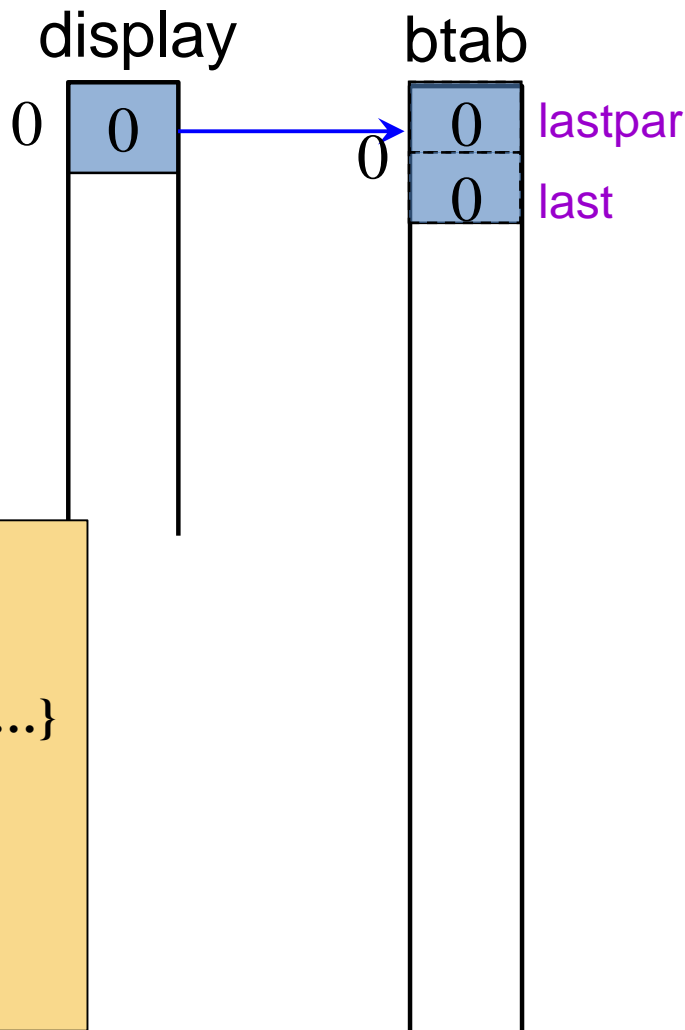
例

- **link**指向同一过程体中定义的**上一个名字**在nametab中的位置，每个过程体在nametab中登记的**第一个名字**的link为0

```
program sort;  
  var a:int[11]; x:int;  
  proc readarray;  
    var i:int; {...}  
  proc exchange (i,j:integer); {...}  
  proc quicksort(m, n:integer);  
    var k, v:int;  
    func partition(y, z:integer);  
      var i, j:int; {...}  
      {...}  
      {...}
```



例



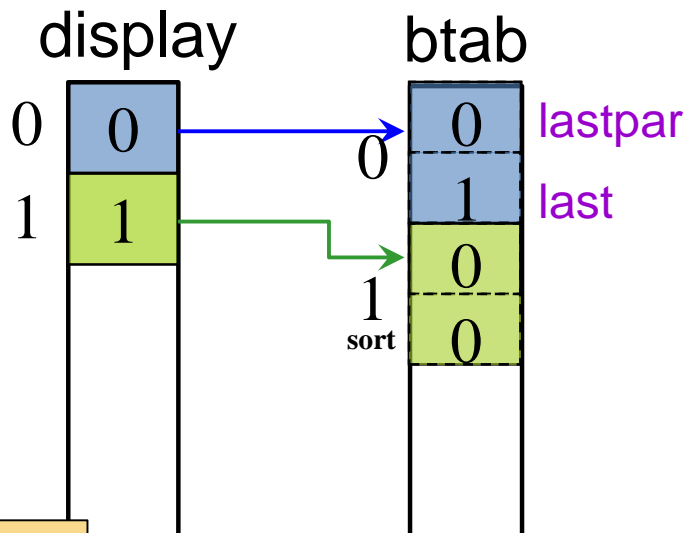
nametab	
0	<div>name</div> <div>link</div>
1	<div>sort</div> <div>0</div>

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
    {...}
  {...}

```

例



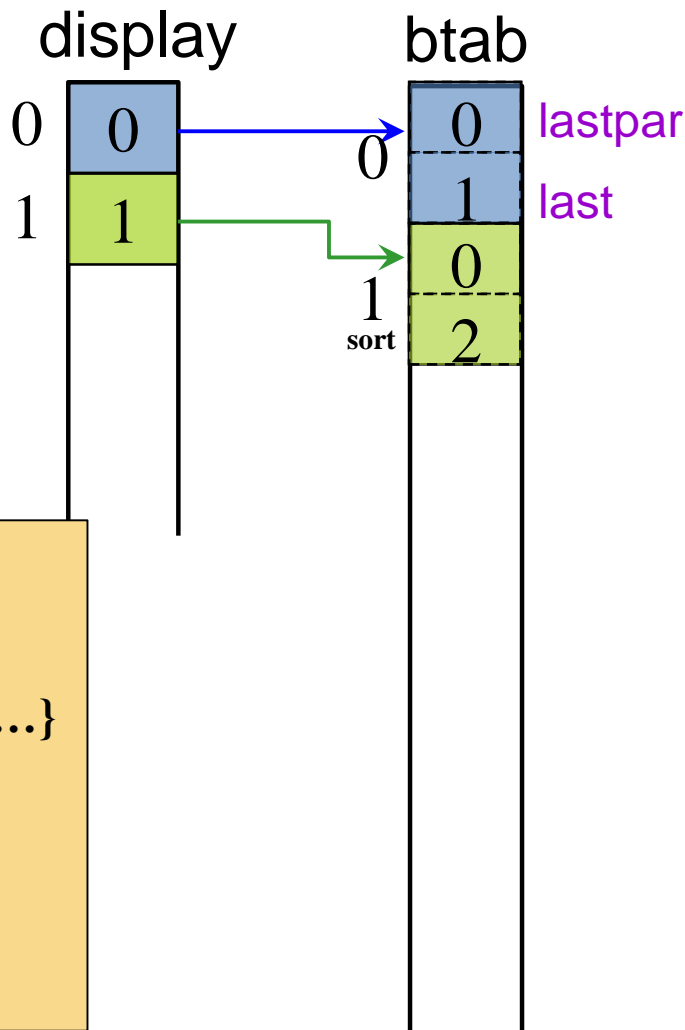
nametab	
0	name link
1	sort 0
2	a 0

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
    {...}
  {...}

```

例



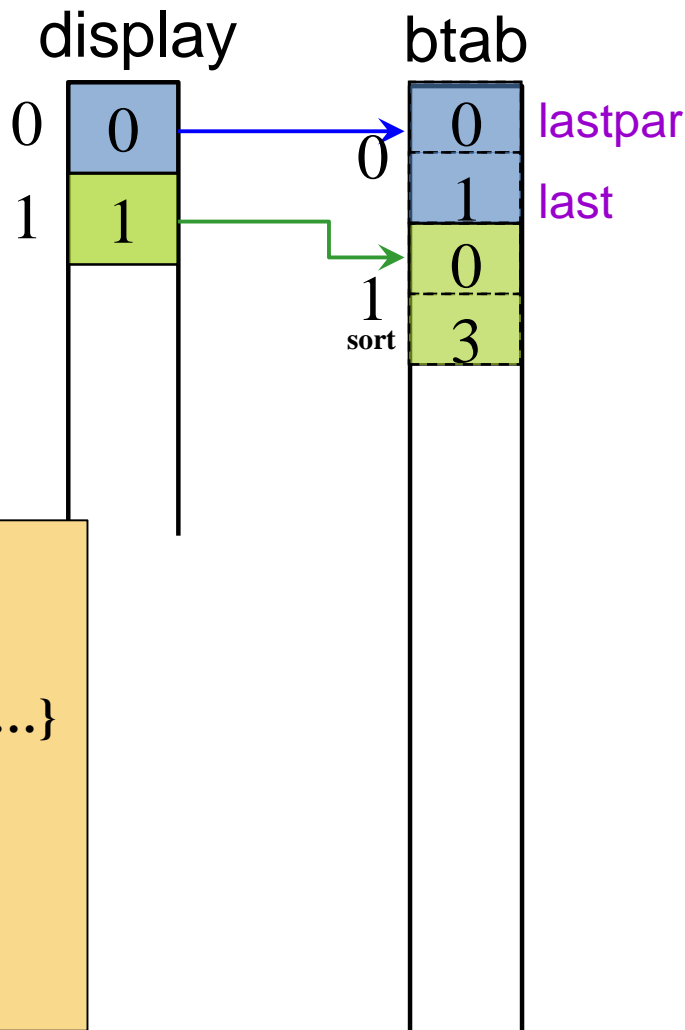
nametab	
0	name link
1	sort 0
2	a 0
3	x 2

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
    {...}
  {...}

```


例



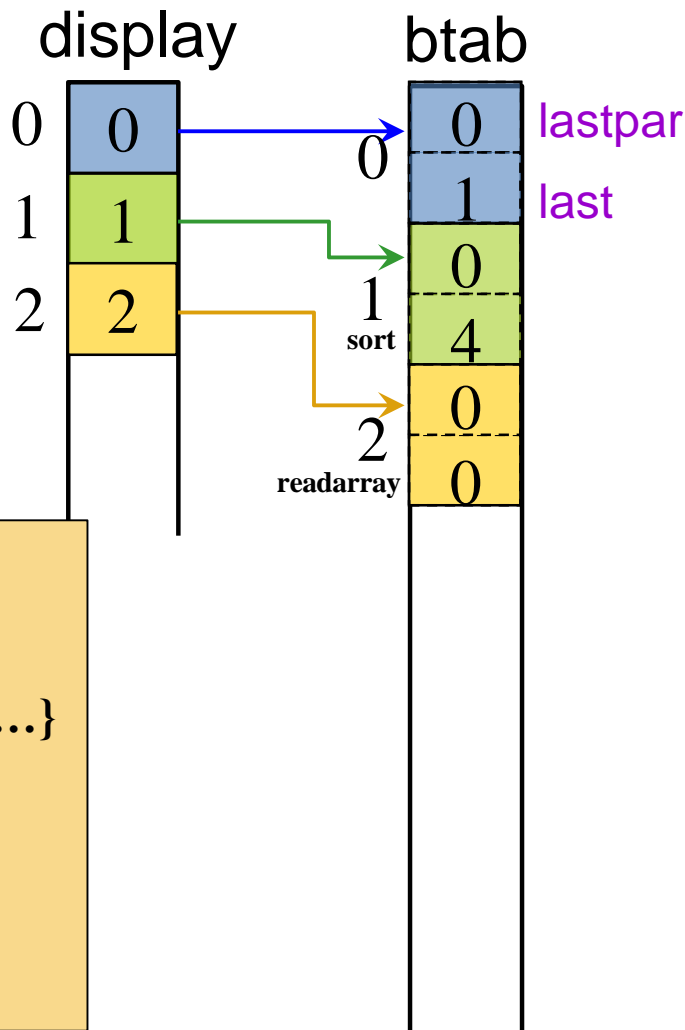
nametab	
0	name link
1	sort 0
2	a 0
3	x 2
4	readarray 3

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
    {...}
  {...}

```

例



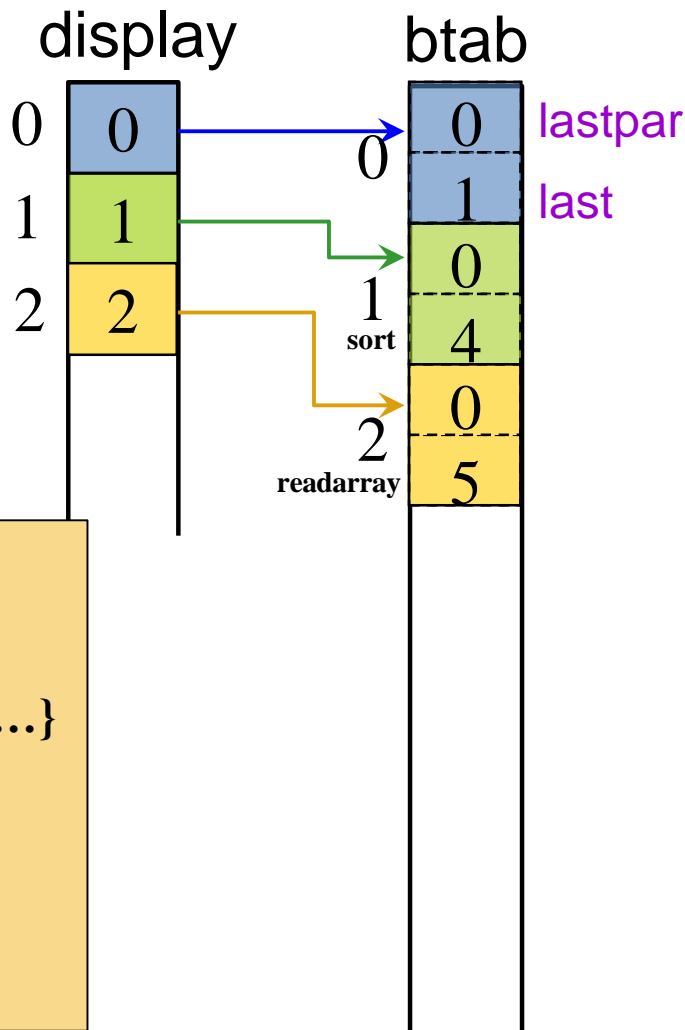
nametab	
0	name link
1	sort 0
2	a 0
3	x 2
4	readarray 3
5	i 0

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
      {...}

```

例



nametab	
0	name link
1	sort 0
2	a 0
3	x 2
4	readarray 3
5	i 0

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
      {...}

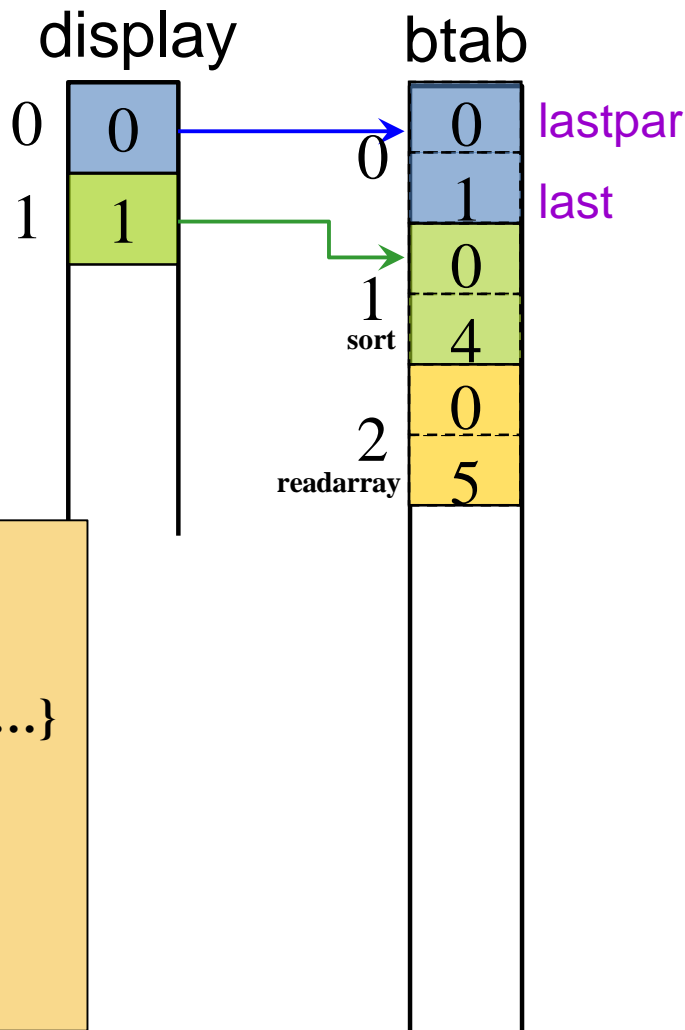
```

例

```

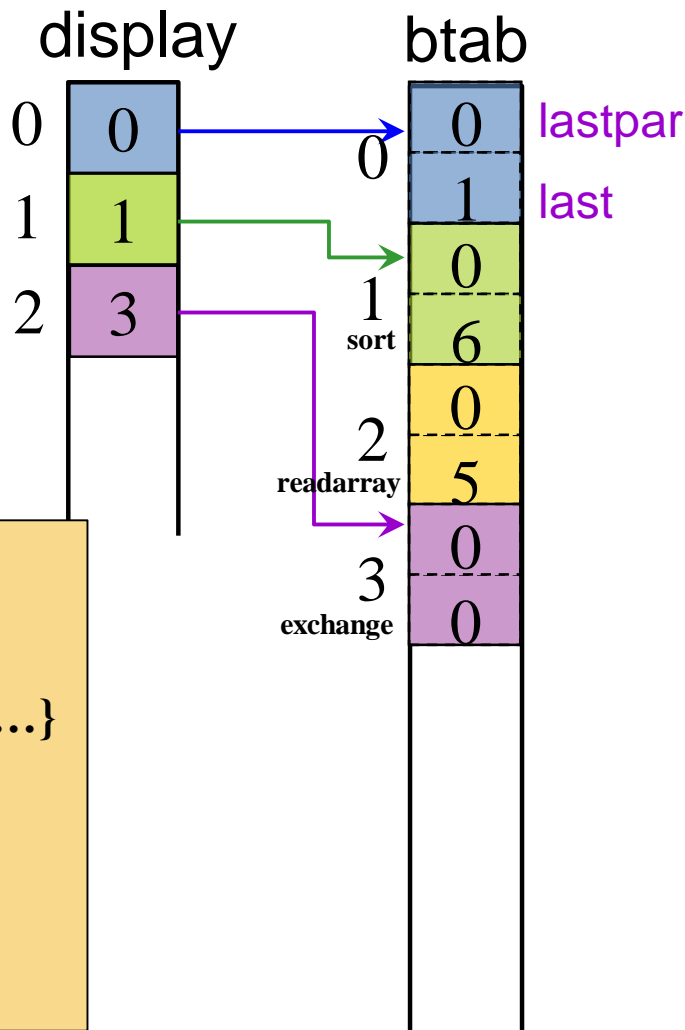
program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
    {...}
  {...}

```



nametab	
0	name link
1	<i>sort</i> 0
2	<i>a</i> 0
3	<i>x</i> 2
4	<i>readarray</i> 3
5	<i>i</i> 0
6	<i>exchange</i> 4

例

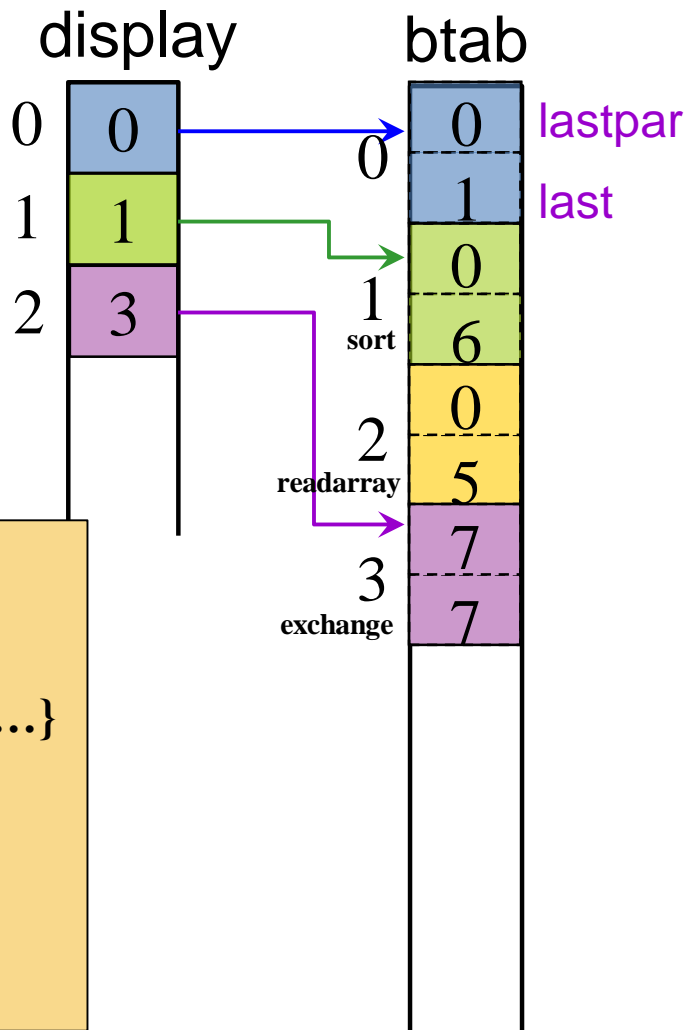


nametab	
0	name link
1	sort 0
2	a 0
3	x 2
4	readarray 3
5	i 0
6	exchange 4
7	i 0

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
      {...}
    end func;
  end proc;
end program;
  
```

例



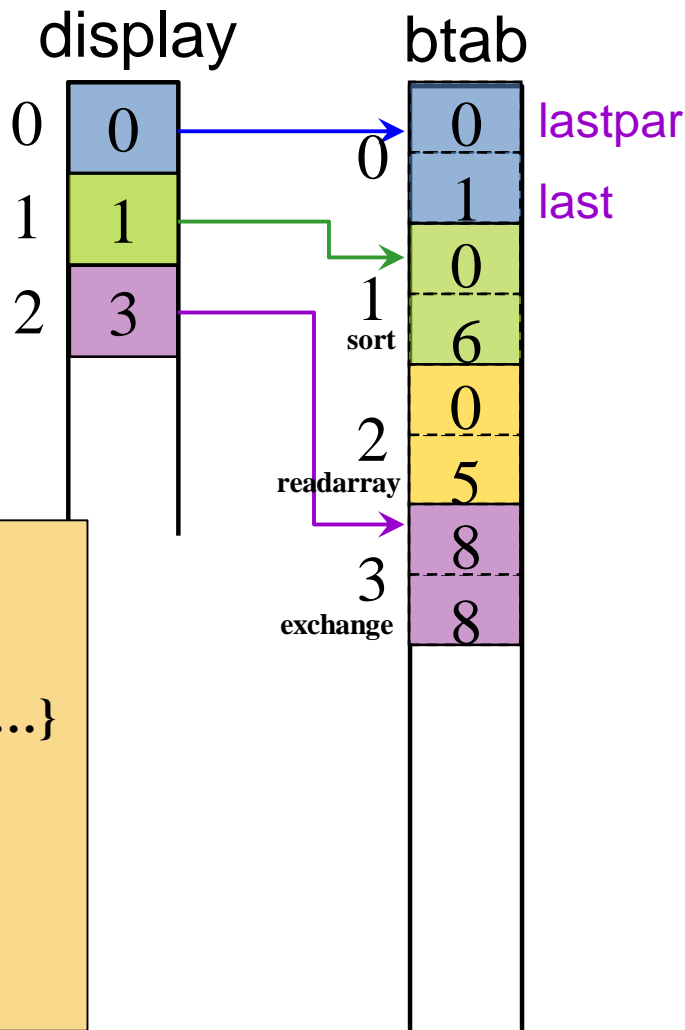
nametab		
	name	link
0		
1	sort	0
2	a	0
3	x	2
4	readarray	3
5	i	0
6	exchange	4
7	i	0
8	j	7

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i,j:int; {...}
    {...}
  {...}

```

例



nametab		
	name	link
0		
1	sort	0
2	a	0
3	x	2
4	readarray	3
5	i	0
6	exchange	4
7	i	0
8	j	7

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
    {...}
  {...}

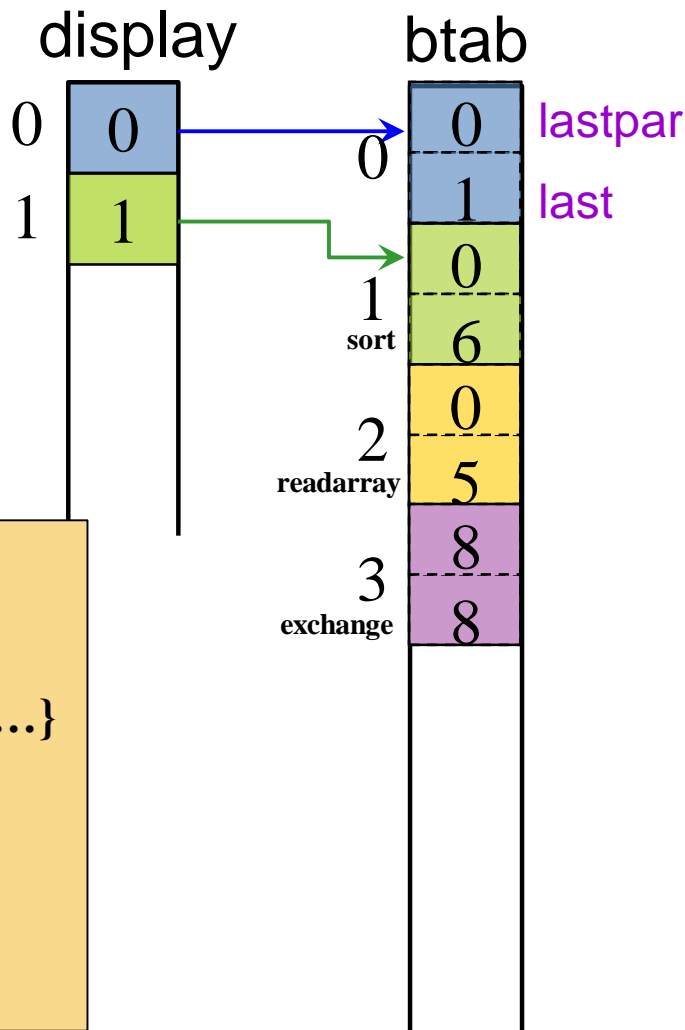
```

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
      {...}

```



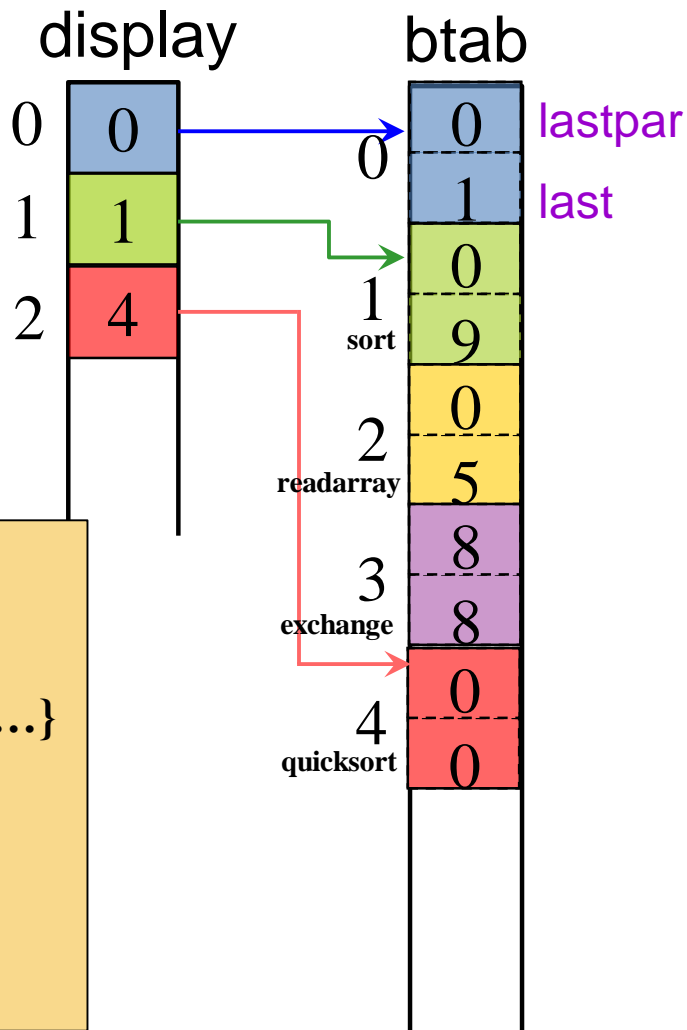
nametab	
0	name link
1	<i>sort</i> 0
2	<i>a</i> 0
3	<i>x</i> 2
4	<i>readarray</i> 3
5	<i>i</i> 0
6	<i>exchange</i> 4
7	<i>i</i> 0
8	<i>j</i> 7
9	<i>quicksort</i> 6

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
      {...}

```



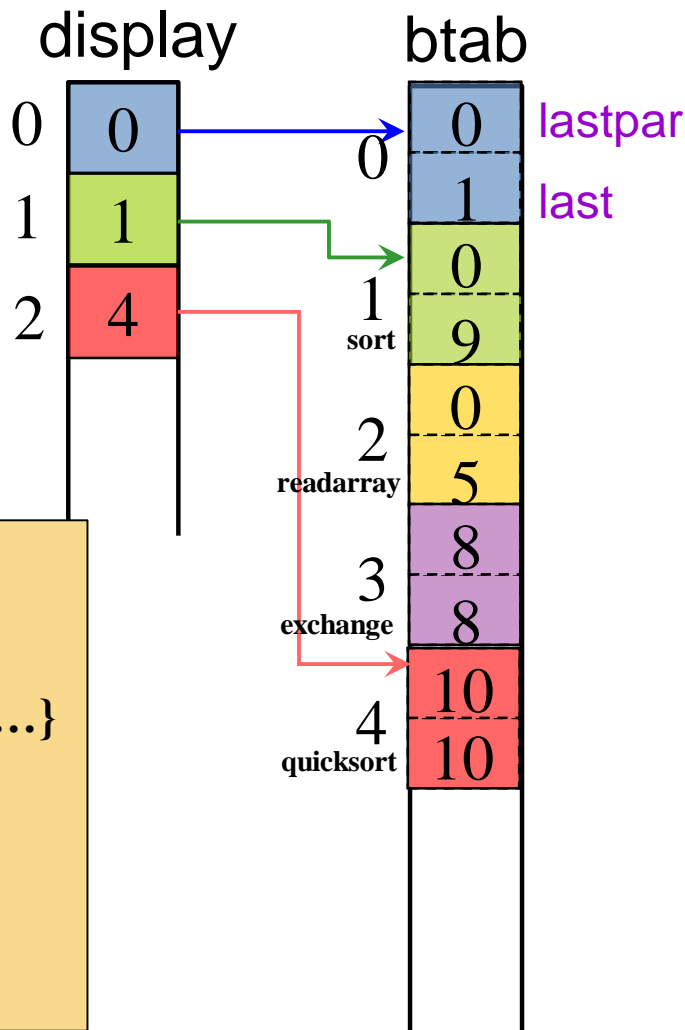
nametab	
0	name link
1	sort 0
2	a 0
3	x 2
4	readarray 3
5	i 0
6	exchange 4
7	i 0
8	j 7
9	quicksort 6
10	m 0

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
      {...}

```



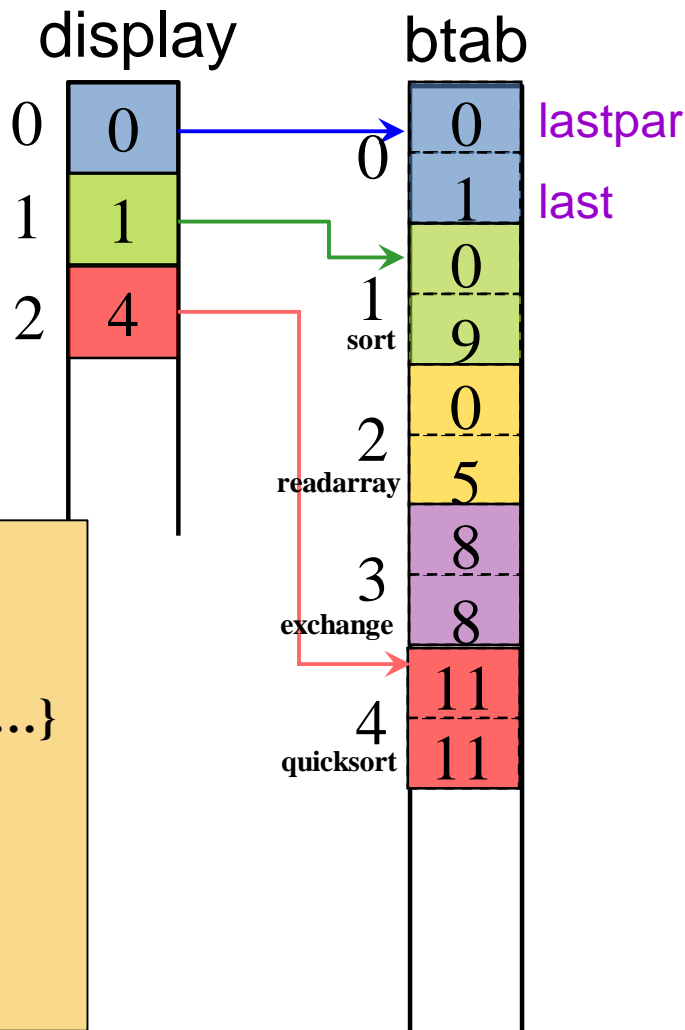
	nametab	
	name	link
0		
1	sort	0
2	a	0
3	x	2
4	readarray	3
5	i	0
6	exchange	4
7	i	0
8	j	7
9	quicksort	6
10	m	0
11	n	10

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i,j:int; {...}
      {...}
      {...}

```



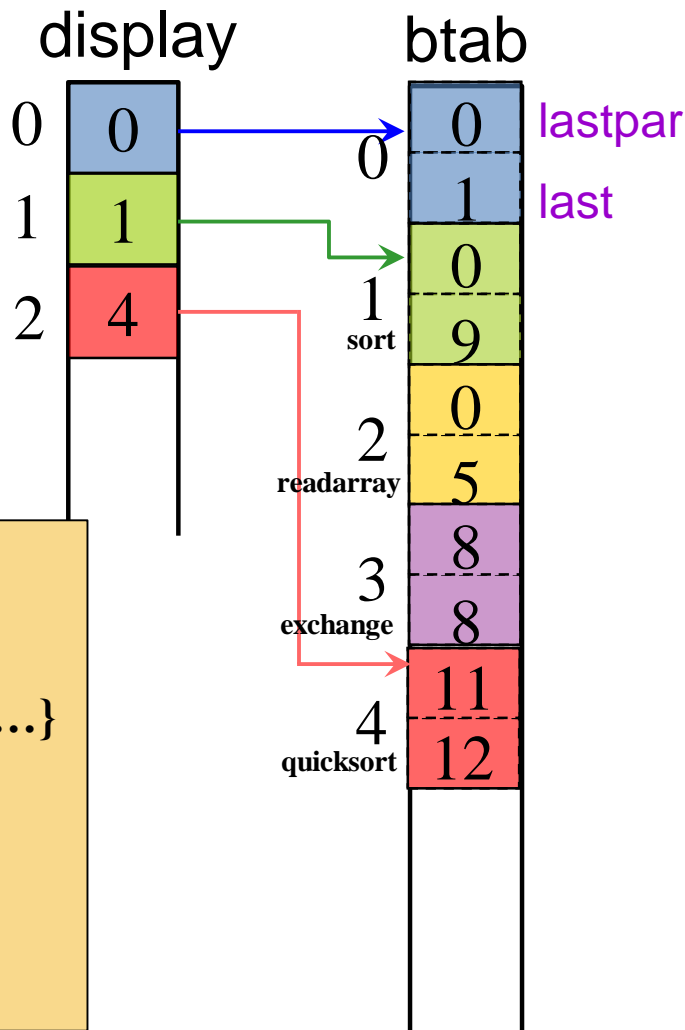
	nametab	
	name	link
0		
1	sort	0
2	a	0
3	x	2
4	readarray	3
5	i	0
6	exchange	4
7	i	0
8	j	7
9	quicksort	6
10	m	0
11	n	10
12	k	11

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
      {...}

```



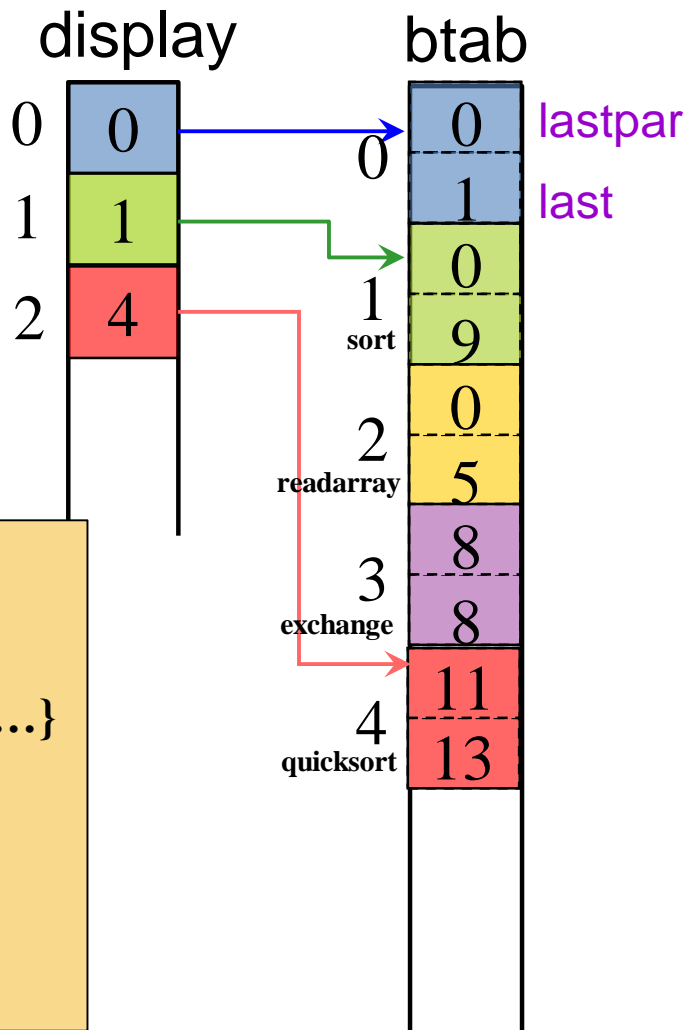
nametab		
	name	link
0		
1	sort	0
2	a	0
3	x	2
4	readarray	3
5	i	0
6	exchange	4
7	i	0
8	j	7
9	quicksort	6
10	m	0
11	n	10
12	k	11
13	v	12

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i,j:int; {...}
      {...}
      {...}

```



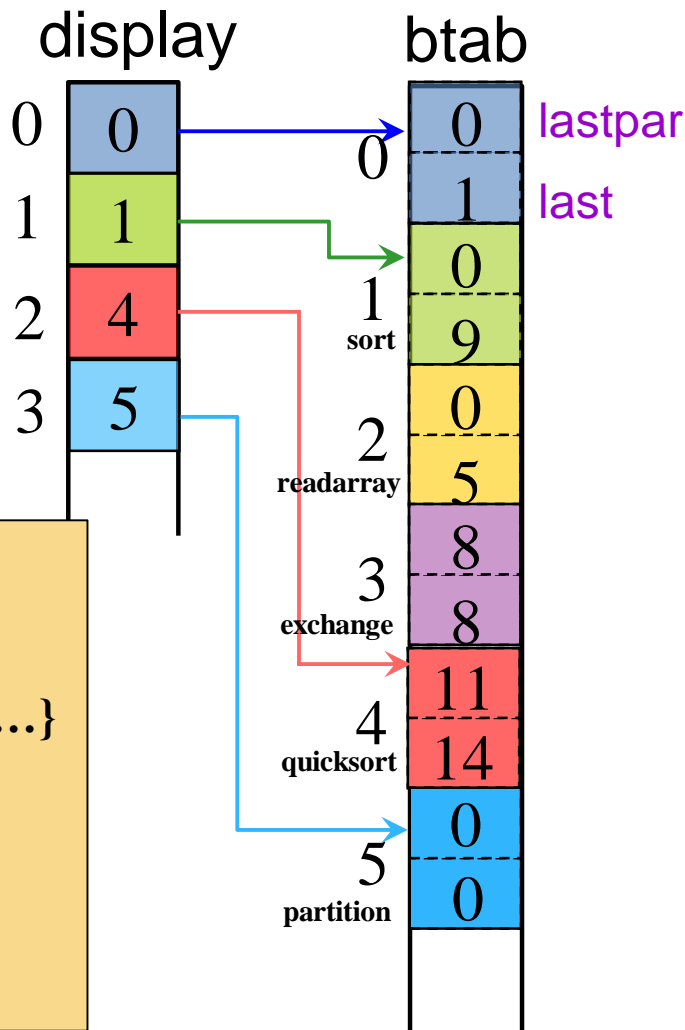
nametab		
	name	link
0		
1	sort	0
2	a	0
3	x	2
4	readarray	3
5	i	0
6	exchange	4
7	i	0
8	j	7
9	quicksort	6
10	m	0
11	n	10
12	k	11
13	v	12
14	partition	13

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
      {...}

```

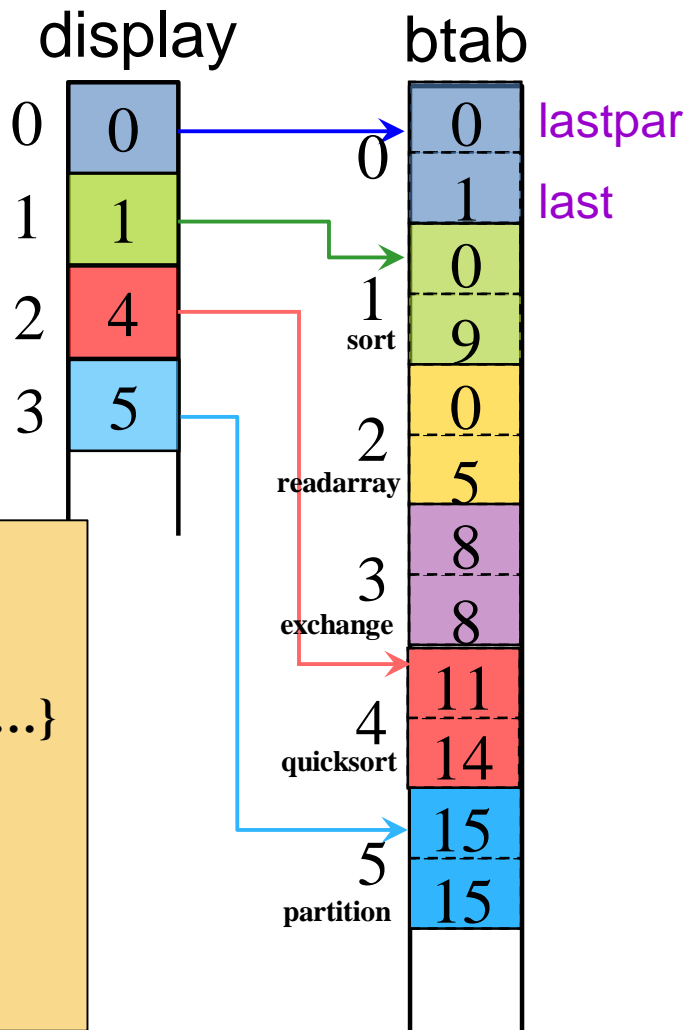


nametab	
name	link
sort	0
a	0
x	2
readarray	3
i	0
exchange	4
i	0
j	7
quicksort	6
m	0
n	10
k	11
v	12
partition	13
y	0

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
    {...}
  
```



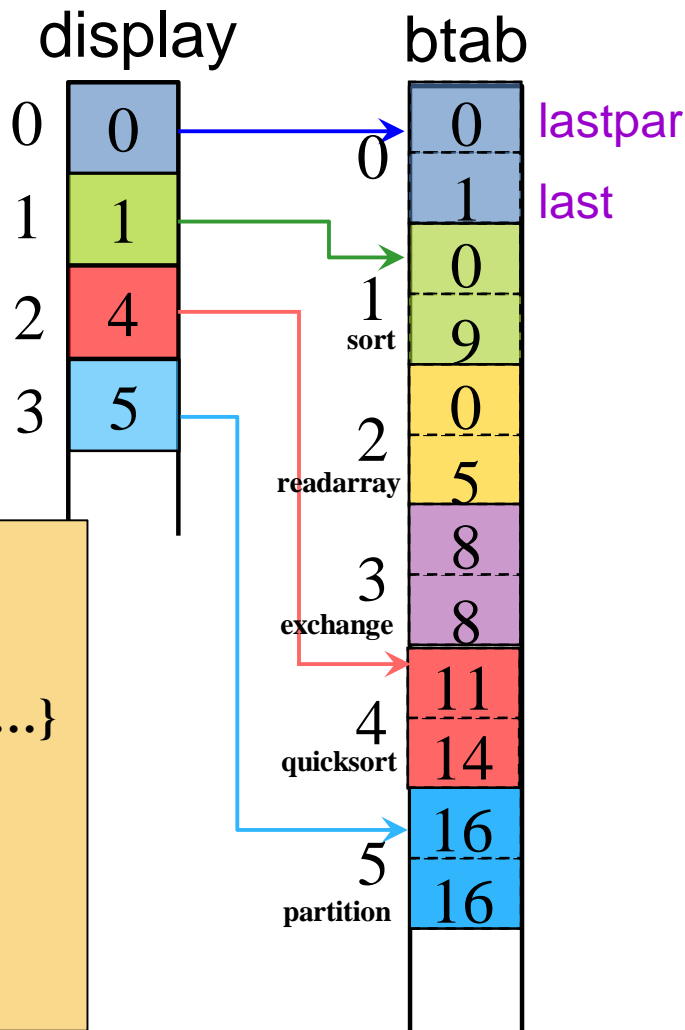
nametab		
	name	link
0		
1	sort	0
2	a	0
3	x	2
4	readarray	3
5	i	0
6	exchange	4
7	i	0
8	j	7
9	quicksort	6
10	m	0
11	n	10
12	k	11
13	v	12
14	partition	13
15	y	0
16	z	15

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
      {...}

```



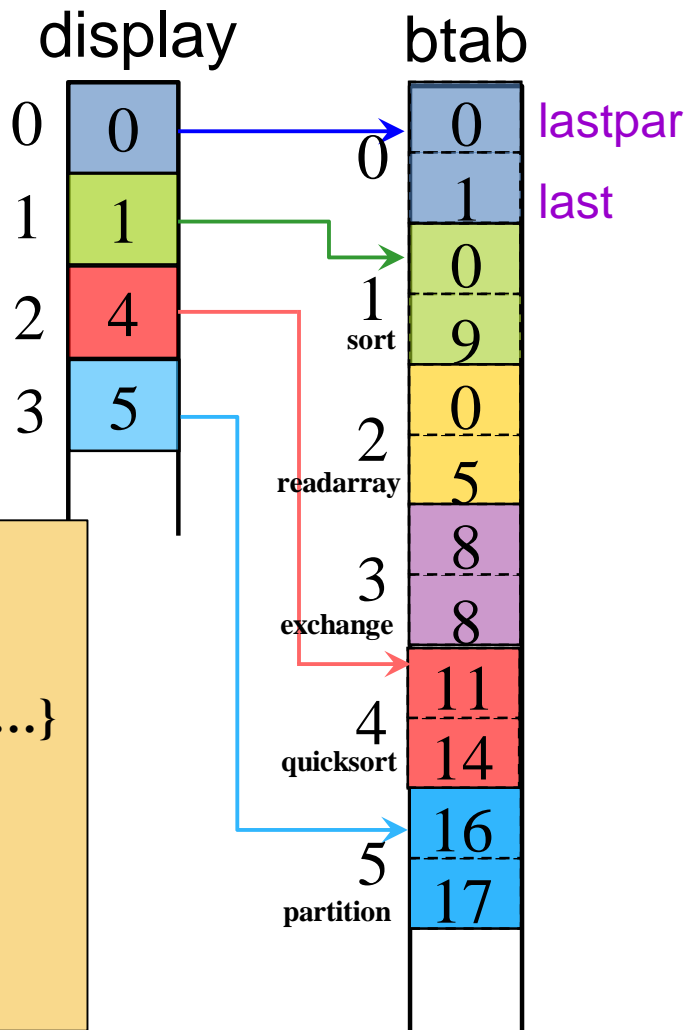
nametab	
0	name link
1	sort 0
2	a 0
3	x 2
4	readarray 3
5	i 0
6	exchange 4
7	i 0
8	j 7
9	quicksort 6
10	m 0
11	n 10
12	k 11
13	v 12
14	partition 13
15	y 0
16	z 15
17	i 16

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
    {...}
  {...}

```



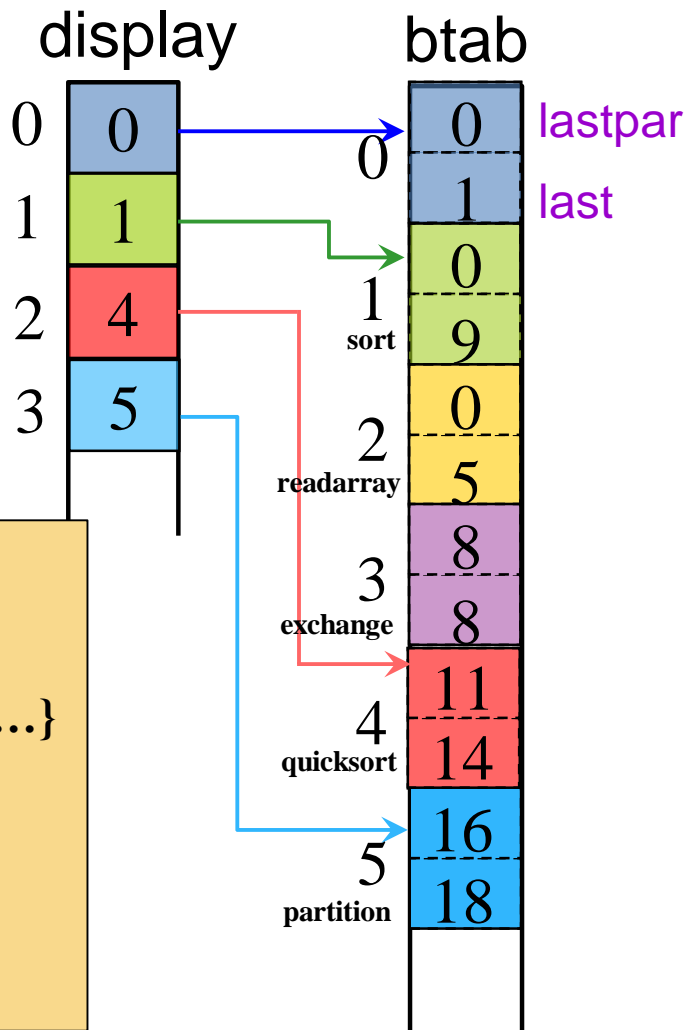
nametab		
	name	link
0		
1	sort	0
2	a	0
3	x	2
4	readarray	3
5	i	0
6	exchange	4
7	i	0
8	j	7
9	quicksort	6
10	m	0
11	n	10
12	k	11
13	v	12
14	partition	13
15	y	0
16	z	15
17	i	16
18	j	17

例

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
    {...}
  {...}

```



nametab		
	name	link
0		
1	sort	0
2	a	0
3	x	2
4	readarray	3
5	i	0
6	exchange	4
7	i	0
8	j	7
9	quicksort	6
10	m	0
11	n	10
12	k	11
13	v	12
14	partition	13
15	y	0
16	z	15
17	i	16
18	j	17

例

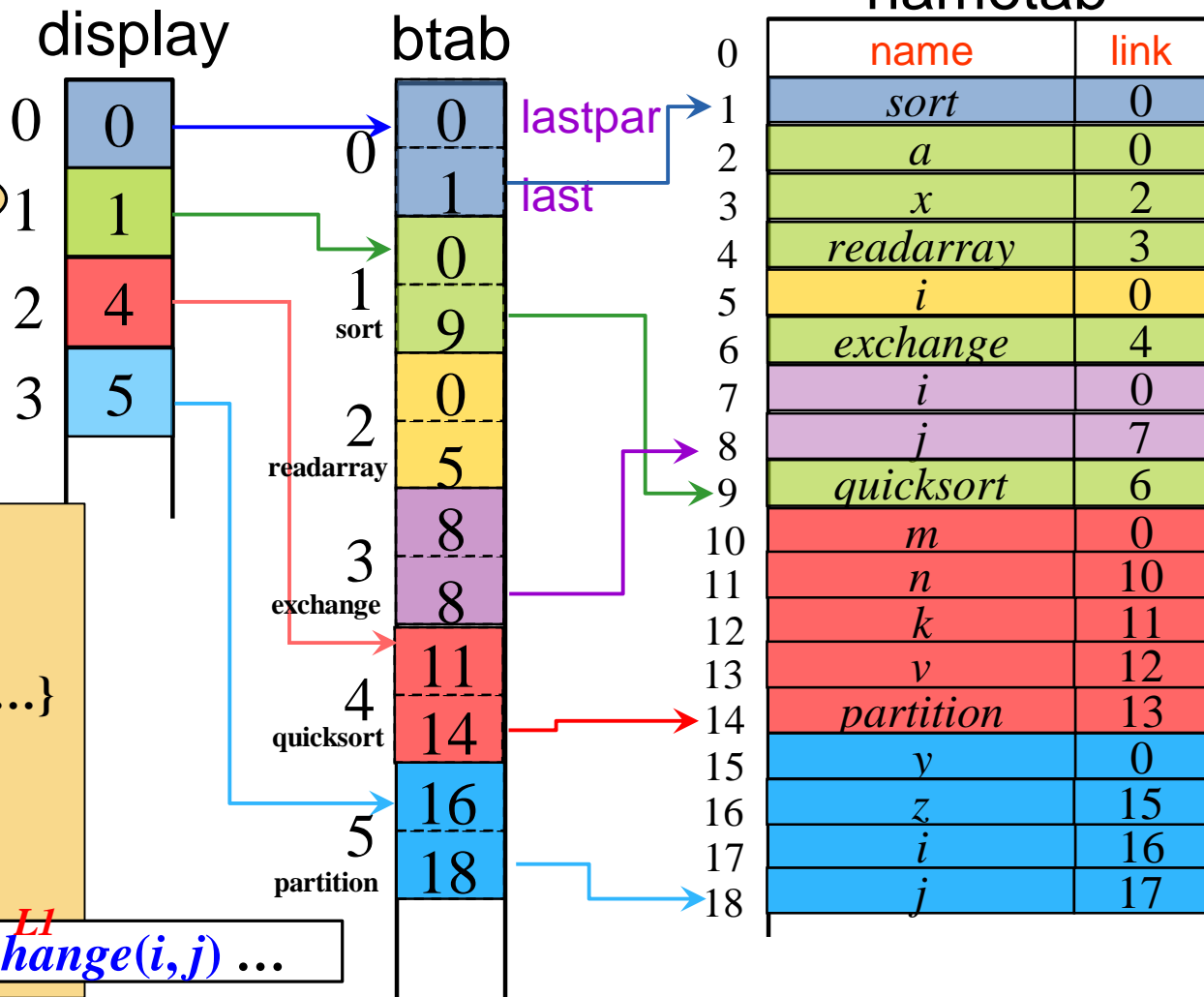
如何根据符号表生成中间代码中的地址信息？

地址 = 层号 + 偏移地址

```

program sort;
  var a:int[11]; x:int;
  proc readarray;
    var i:int; {...}
  proc exchange (i,j:integer); {...}
  proc quicksort(m, n:integer);
    var k, v:int;
    func partition(y, z:integer);
      var i, j:int; {...}
      {...}
      {...}
  end func;
end proc;
end program;
  
```

... ^{L1}*a* ... ^{L2}*v* ... ^{L1}*exchange(i,j)* ...





本章小结

- 存储组织
- 静态存储分配
- 栈式存储分配
- 非局部数据的访问
- 符号表



结束



C语言的符号表

- 以“{”、“}”作为块的分界符
- 一个过程可能包含多个块
- 如何组织符号表？
 - 方法一：将所有块的符号表放在一个**大数组**中，然后再引入一个**程序块表**来描述个程序块的符号表在大数组中的位置及其相互关系
 - 方法二：将描述符号所属程序块的信息（如**块号**）放在符号表的**表项**中，和**符号的名字**共同形成符号表表项的**关键字**。此时的符号表非常类似于未考虑作用域的符号表

方法一：

- 将所有块的符号表放在一个大数组中，然后再引入一个**程序块表**来描述个程序块的符号表在大数组中的位置及其相互关系

```
(1) int main()  
(2) {  
(3)     int a = 0;  
(4)     int b = 0;  
(5)     {  
(6)         int b = 1;  
(7)         {  
(8)             int a = 2;  
(9)             printf("%d %d\n", a, b);  
(10)         }  
(11)         {  
(12)             int b = 3;  
(13)             printf("%d %d\n", a, b);  
(14)         }  
(15)     }  
(16)     printf("%d %d\n", a, b);  
(17) }  
(18)
```

Block structure diagram:
- B₀ (main) contains B₁ and a printf statement.
- B₁ contains B₂, B₃, and a printf statement.
- B₂ contains an int declaration and a printf statement.
- B₃ contains an int declaration and a printf statement.

外层块	符号个数	起始指针	名字	属性
0	-1	2	a	
1	0	1	b	
2	1	1	b	
3	1	1	b	
			a	

程序块表

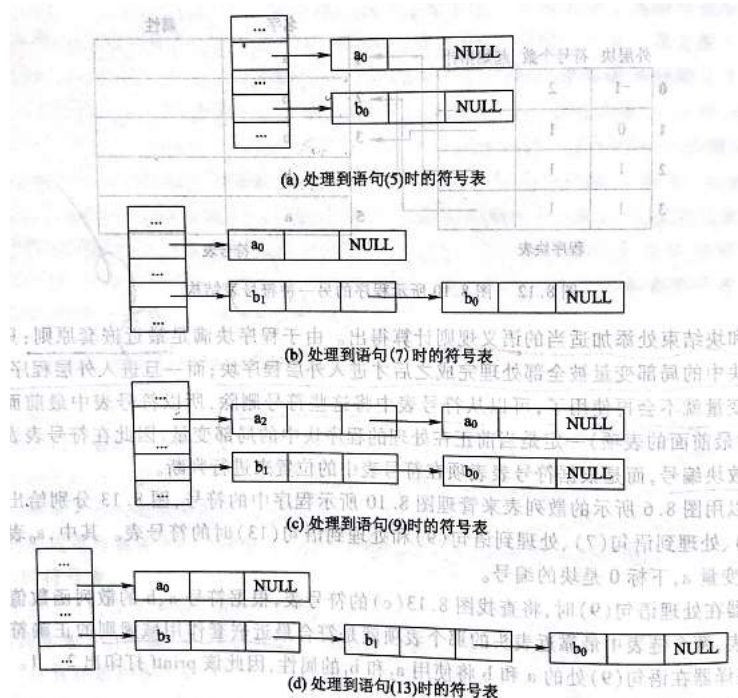
符号表

方法二：

- 将描述符号所属程序块的信息（如块号）放在符号表的表项中，和符号的名字共同形成符号表表项的关键字

```
(1) int main()  
(2) {  
(3)     int a = 0;  
(4)     int b = 0;  
(5)     {  
(6)         int b = 1;  
(7)         {  
(8)             int a = 2;  
(9)             printf("%d %d\n", a, b);  
(10)         }  
(11)         {  
(12)             int b = 3;  
(13)             printf("%d %d\n", a, b);  
(14)         }  
(15)         printf("%d %d\n", a, b);  
(16)     }  
(17) }  
(18)
```

Block structure diagram:
- B₀ (main) contains B₁ and the final printf.
- B₁ contains B₂, B₃, and the middle printf.
- B₂ contains the first printf.
- B₃ contains the second printf.



非局部数据的访问

- 一个过程（块）除了可以使用过程（块）自身声明的局部数据以外，还可以使用过程（块）外声明的非局部数据
- 全局数据
- 外围过程（块）定义的数据
- 例

```
procedure A;  
  var x: integer;  
  ...  
procedure B;  
  var x: integer;  
  ...  
procedure C;  
  var a: real;  
  begin ... x... end {C};  
begin ... end {B};  
begin ... end {A};
```

名字 **x** 的当前出现对应的是哪个声明语句？

课程主要内容

1. 绪论 (2学时)
2. 语言及其文法 (2学时)
3. 词法分析 (3学时)
4. 语法分析 (9学时)
5. 语法制导翻译 (6学时)
6. 中间代码生成 (7学时)
7. 运行时的存贮组织 (3学时)
8. 代码优化 (6学时)
9. 代码生成 (2学时)