编译原理

第七章
运行存储分配

哈尔滨工业大学 陈鄞

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

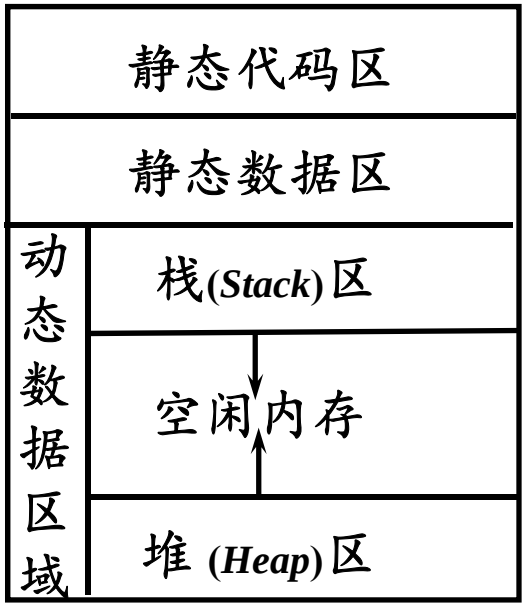
7.5 符号表

7.1 存储组织

- 编译器在工作过程中，必须为源程序中出现的一些数据对象分配运行时的存储空间
- 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

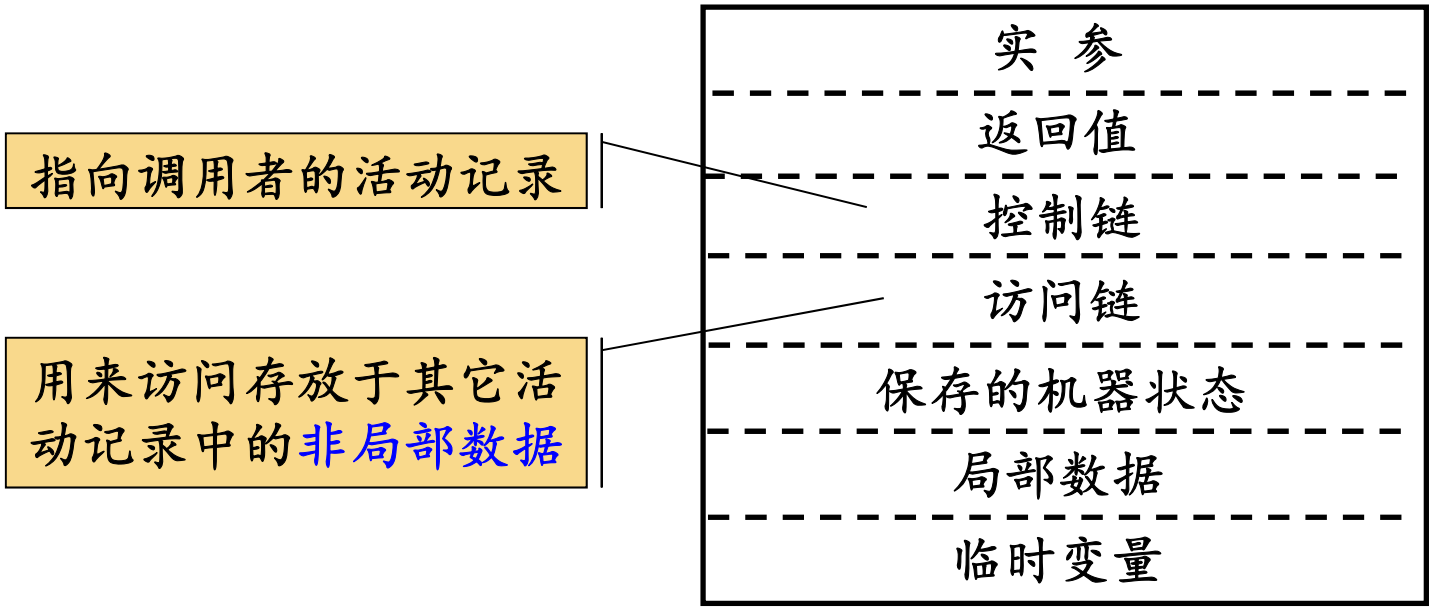
运行时内存的划分



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
实 参
返回值
控制链
访问链
保存的机器状态
局部数据
临时变量

提纲

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

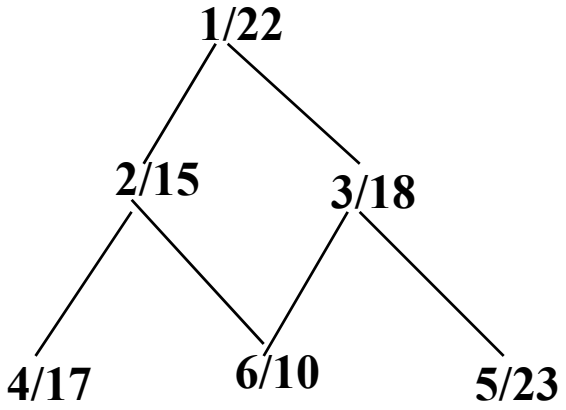
- 适合静态存储分配的语言必须满足以下条件
  - 数组上下界必须是常数
  - 不允许过程的递归调用
  - 不允许动态建立数据实体
- 满足这些条件的语言有BASIC和FORTRAN等

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

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

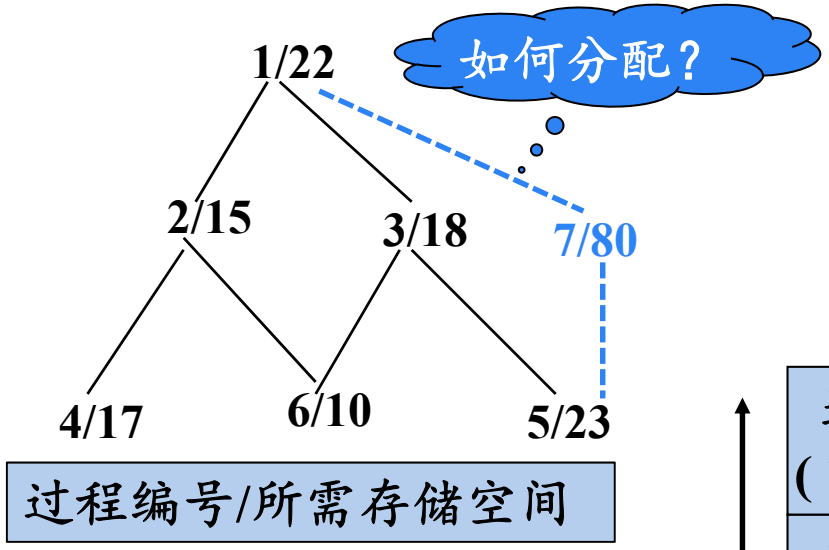
能用更少的空间么？

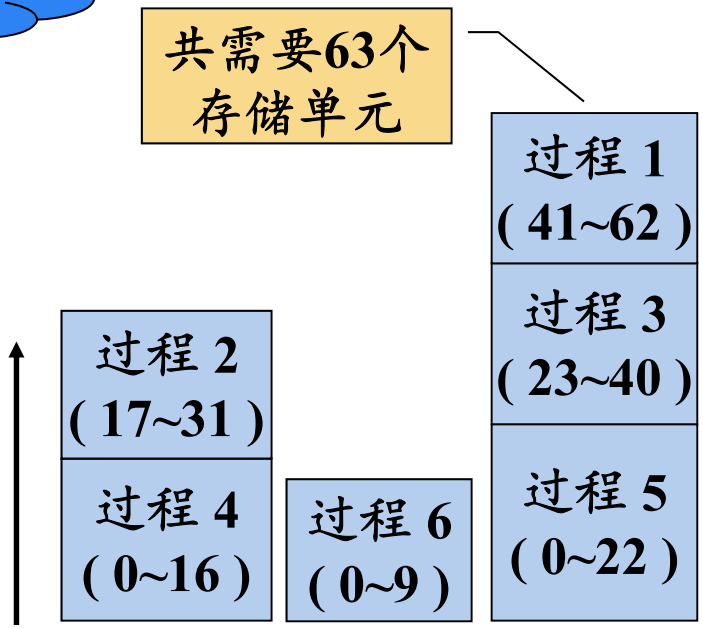
优点：处理上简单

缺点：对内存空间的使用不够经济合理

层次分配法

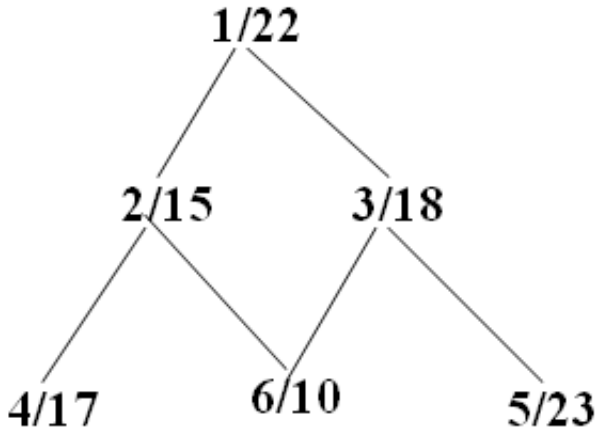
- 通过对过程间的调用关系进行分析，凡属无相互调用关系的并列过程，尽量使其局部数据共享存储空间





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
  - *B*[*i*][*j*]=0： 表示第*i*个过程不调用第*j*个过程

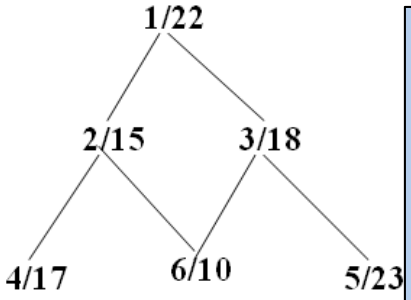


- *Units*[*n*]：过程所需内存量矩阵

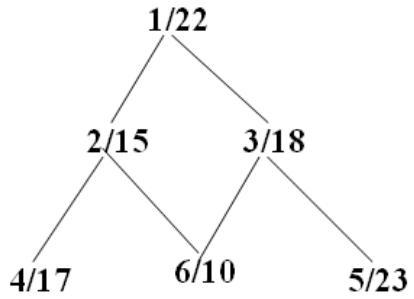
	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	0	1	0	1
3	0	0	0	0	1	1
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0

***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 indicating how many blocks there are */
   int i , j, k, Sum;
   int *allocated; /*used to indicate if the block is allocated */
   allocated=(int*)malloc(sizeof(int) *NoOfBlocks);
   for(i=0; i<NoOfBlocks; i ++)/*Initial arrays base and 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 ];        /*to check out if block i calls some
                                           block which has not been allocated* /
```




```
    if ( ! Sum  &&  ! allocated [ i ])
    { /*Sum=0 means block i  calls no block which has not been allocated;
         allocated [ i ]=0; means block i is not allocated */
        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  maens block k calls block i */
        { /*Since block k calls i , it must be allocated after block i. It
means the base of block k must be greater than base of block i */
            if (base[ k ] < base[ i ]+Units[ i ])
            base[ k ]= base[ i ]+Units[ i ];
            B[k][i]=0; /*Since block in has been allocated B[k][i] should be
modified */
         }
      }
   }
  free(allocated );
}
```

提纲

7.3 栈式存储分配

- 有些语言使用过程、函数或方法作为用户自定义动作的单元，几乎所有针对这些语言的编译器都把它们的(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
- 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
- 这种安排不仅允许活跃时段不交叠的多个过程调用之间共享空间，而且允许以如下方式为一个过程编译代码：它的非局部变量的相对地址总是固定的，和过程调用序列无关

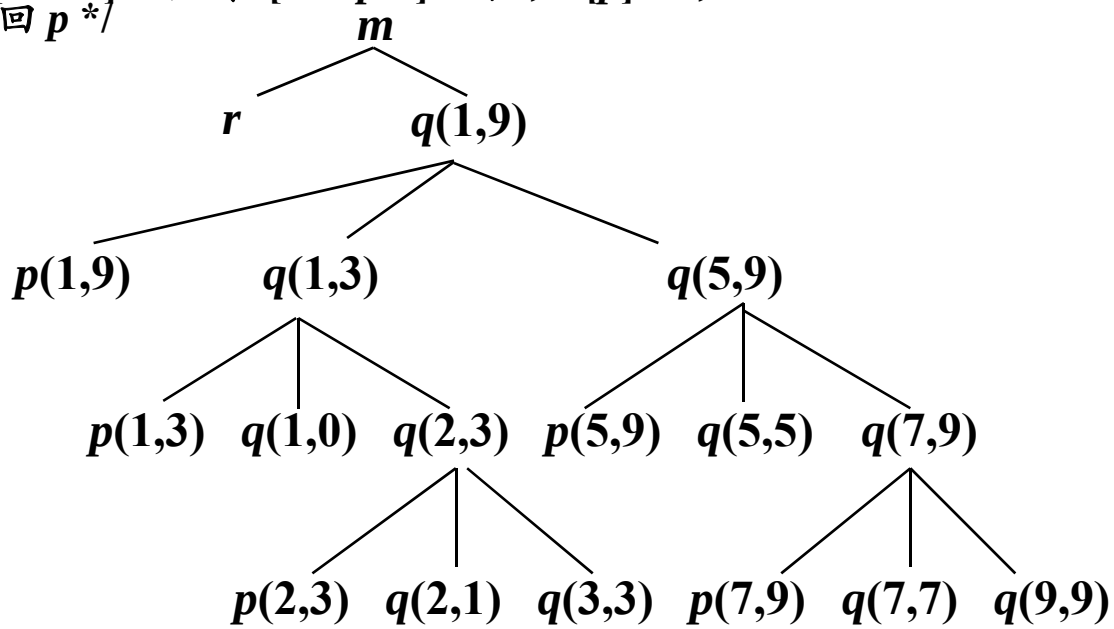
活动树

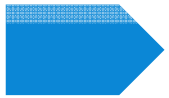
- 用来描述程序运行期间控制进入和离开各个活动的情况的树称为活动树
- 树中的每个结点对应于一个活动。根结点是启动程序执行的*main*过程的活动
- 在表示过程*p*的某个活动的结点上，其子结点对应于被*p*的这次活动调用的各个过程的活动。按照这些活动被调用的顺序，自左向右地显示它们。一个子结点必须在其右兄弟结点的活动开始之前结束

例：一个快速排序程序的概要

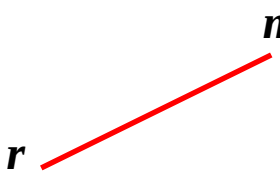
```
int   a[11];
void   readArray( ) /*将9个整数读入到a[1],…,a[9]中 */
{        int   i;
        …
}
int   partition(int  m,  int  n)
{       /*选择一个分割值v，划分a[m…n]，使得a[m…p-1]小于v，a[p]= v，
           a[p+1…n]大于等于v。返回 p */
         …
}
void   quicksort(int  m,  int  n)
{        int   i;
         if (n＞m) {
     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

每个活跃的活动都有一个位于控制栈中的活动记录





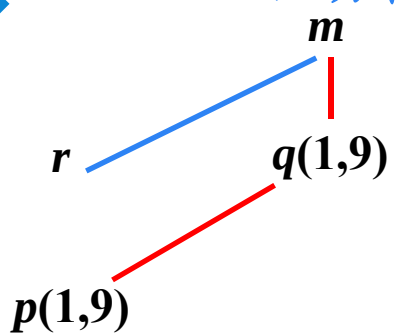
活动树



控制栈

main
int *a*[11]
r
int *i*

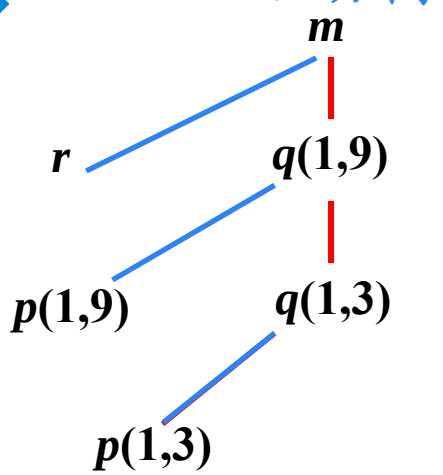
活动树



控制栈

main
int *a*[11]
q(1,9)
int *i*
p(1,9)
int *i*

活动树

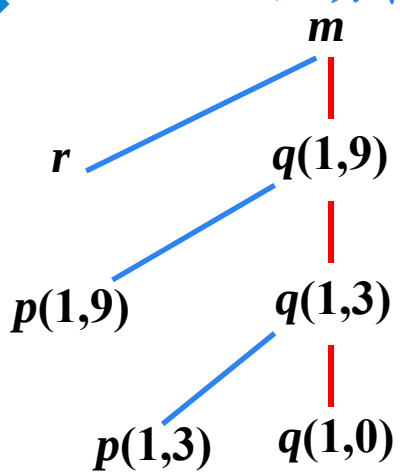


控制栈

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

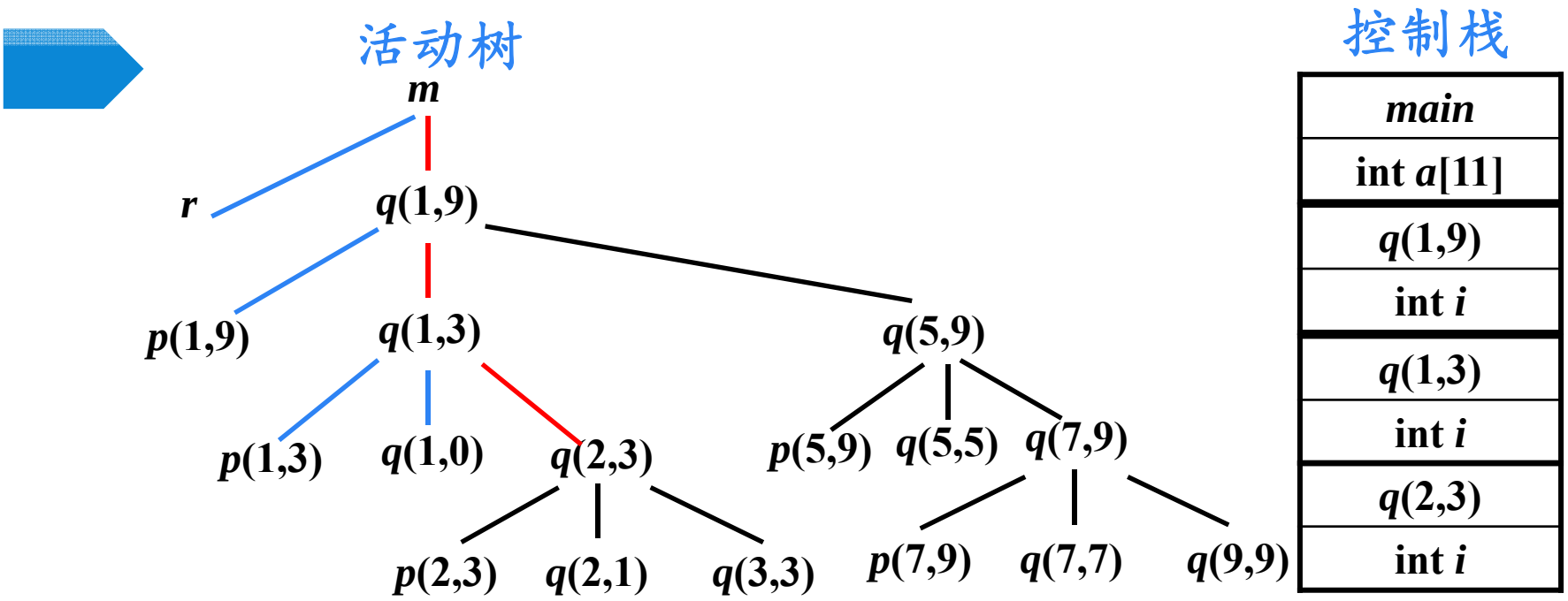
当一个过程是递归的时候，常常会有该过程的多个活动记录同时出现在栈中

活动树



控制栈

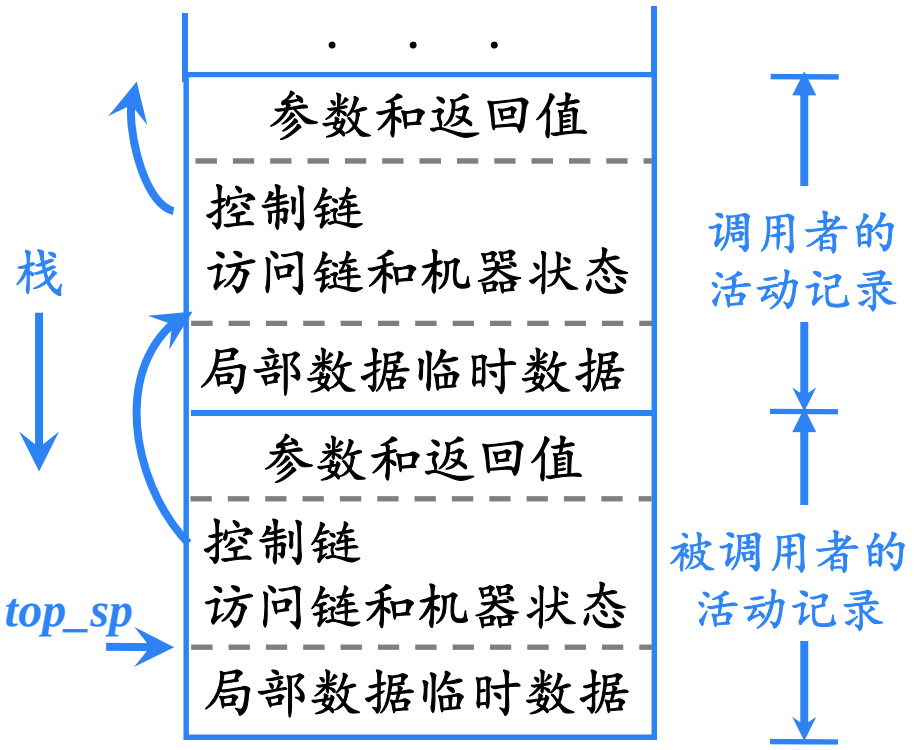
main
int *a*[11]
q(1,9)
int *i*
q(1,3)
int *i*
q(1,0)
int *i*



- 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
  - 控制连、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*指向活动记录中局部数据开始的位置，以该位置作为基地址

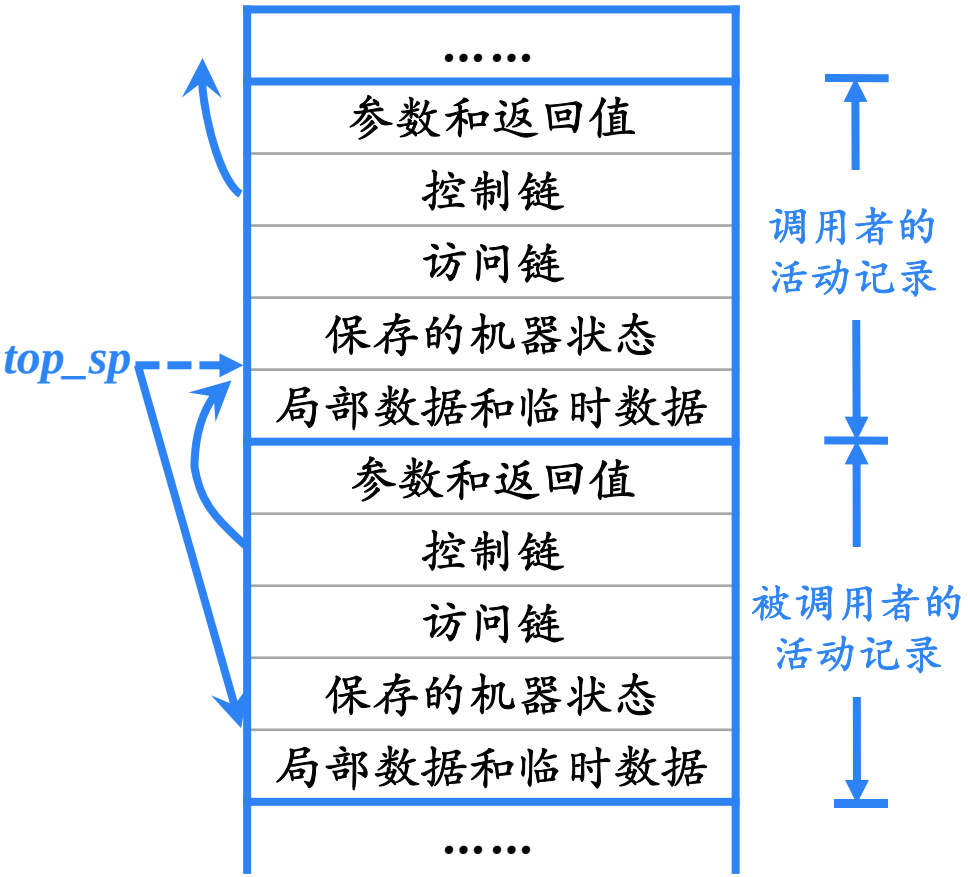


调用序列和返回序列

- 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
  - 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
  - 一个调用代码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。返回序列也是如此

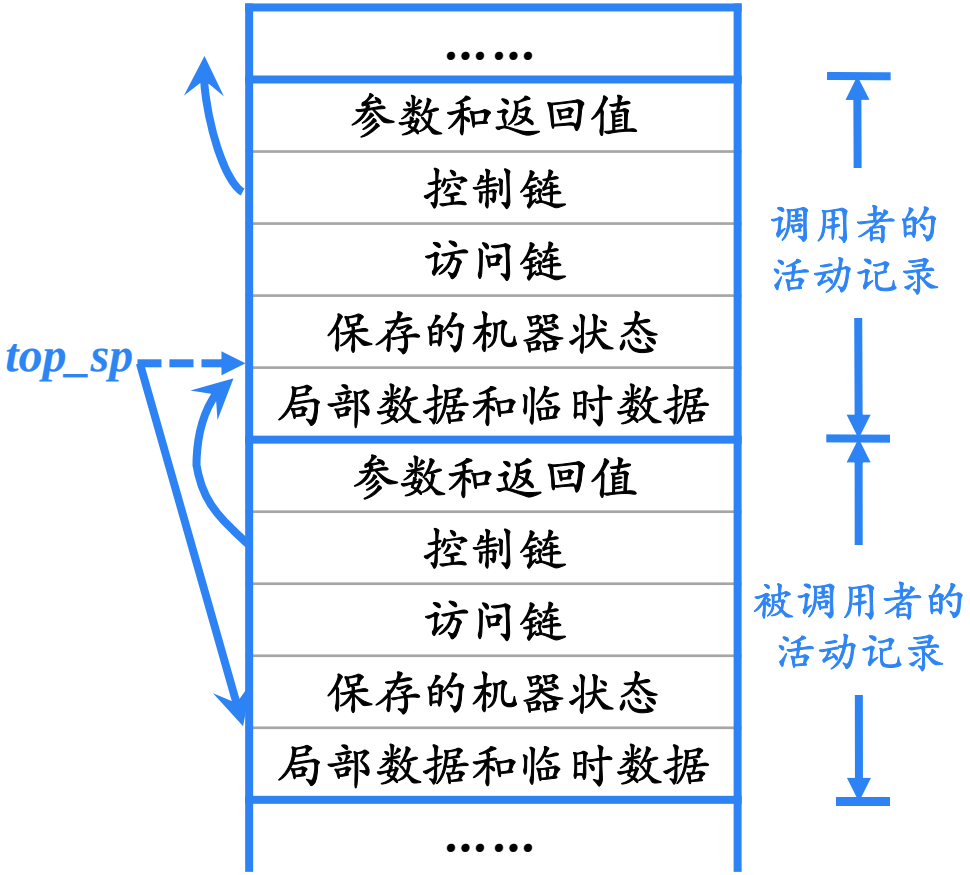
调用序列

- 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。将原来的*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

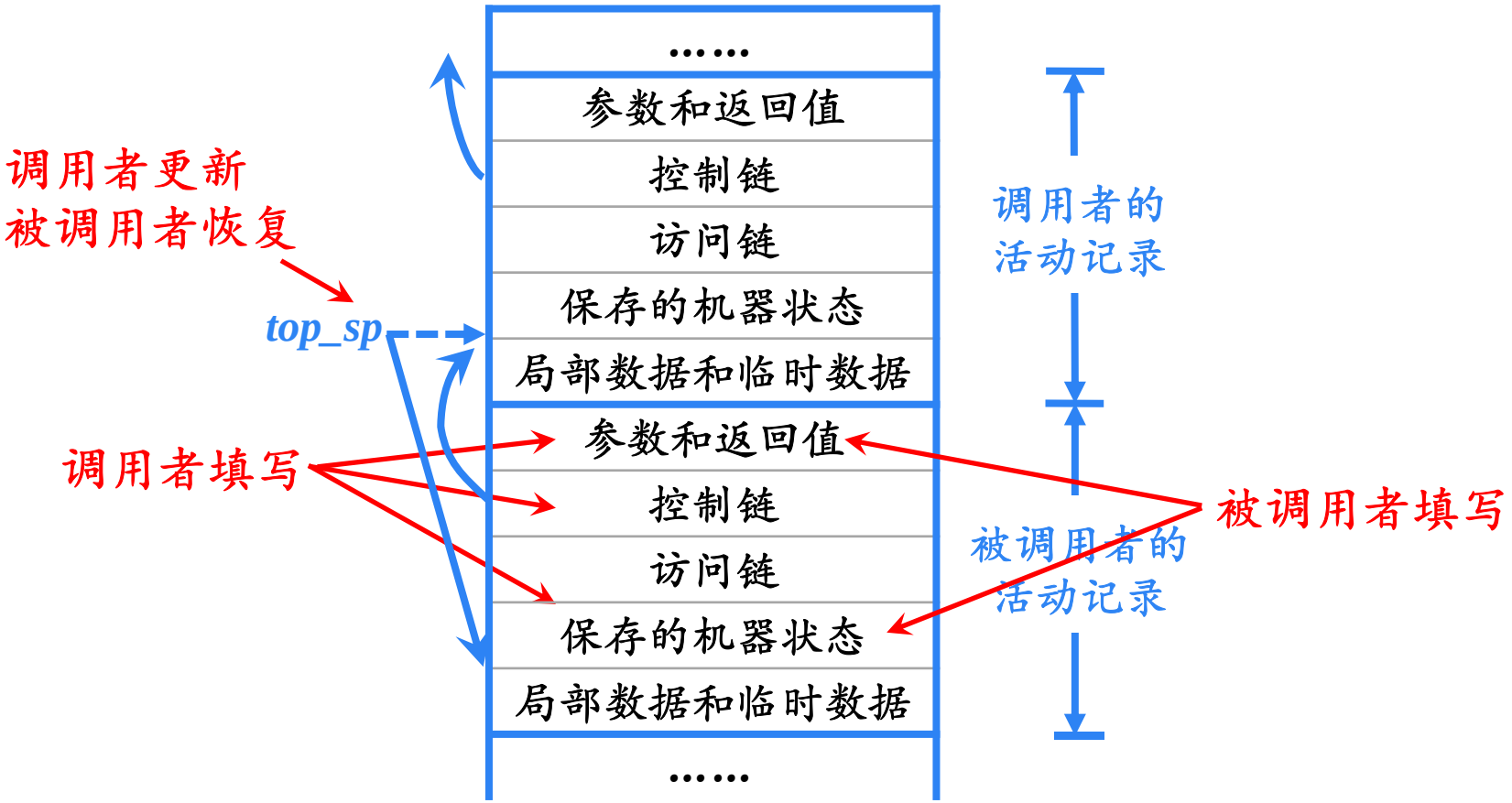


返回序列

- 被调用者将返回值放到与参数相邻的位置
- 使用机器状态字段中的信息，被调用者恢复*top-sp*和其它寄存器，然后跳转到由调用者放在机器状态字段中的返回地址
- 尽管*top-sp*已经被减小，但调用者仍然知道返回值相对于当前*top-sp*值的位置。因此，调用者可以使用那个返回值



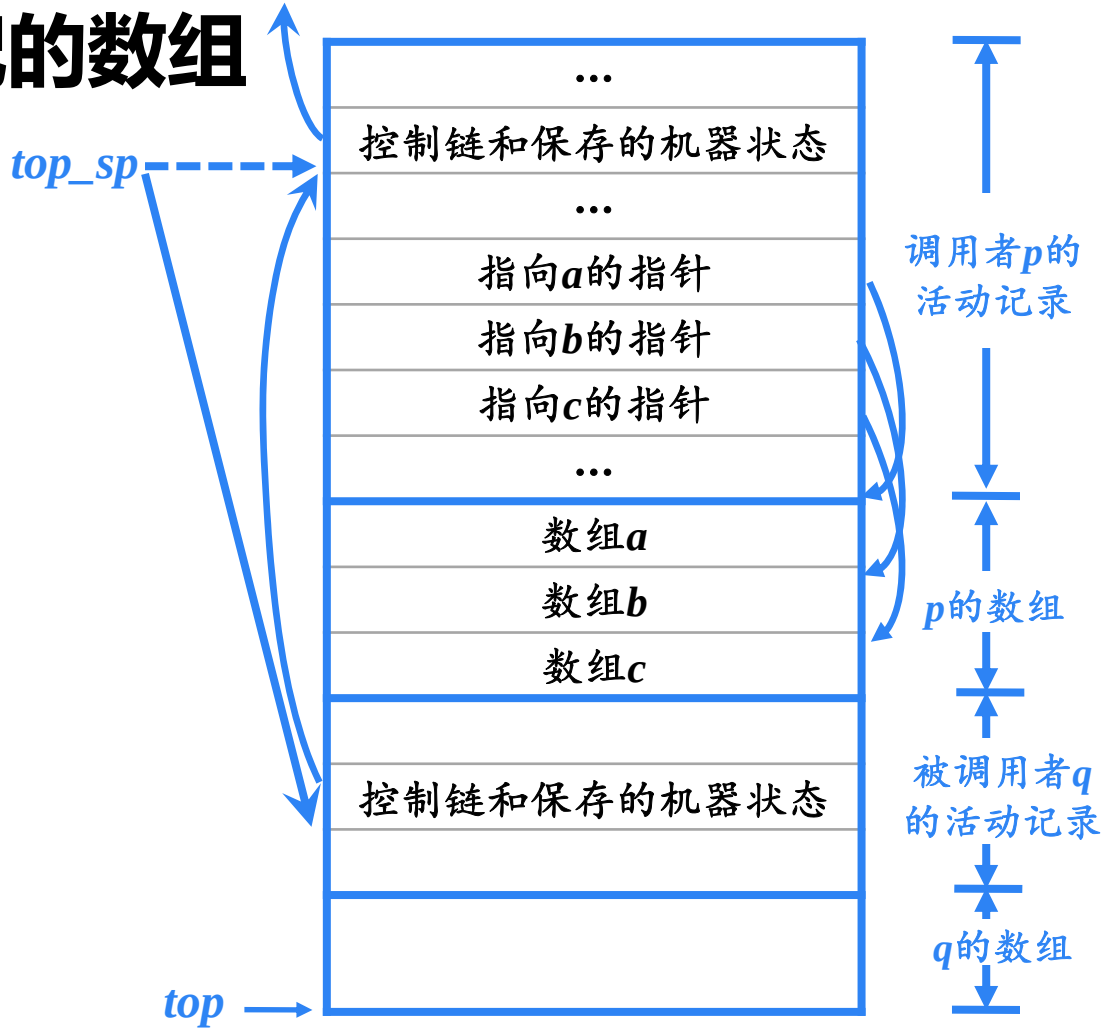
调用者和被调用者之间的任务划分



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将被分配在堆区。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


top_sp
...
控制链和保存的机器状态
...
指向a的指针
指向b的指针
指向c的指针
...
数组a
数组b
数组c
控制链和保存的机器状态
top
调用者p的活动记录
p的数组
被调用者q的活动记录
q的数组

提纲

非局部数据的访问

- 一个过程除了可以使用过程自身定义的局部数据以外，还可以使用过程外定义的非局部数据
- 语言可以分为两种类型
  - 支持过程嵌套声明的语言
    - 可以在一个过程中声明另一个过程
    - 例: ***Pascal***
  - 不支持过程嵌套声明的语言
    - 不可以在一个过程中声明另一个过程
    - 例： ***C***

支持过程嵌套声明的语言

➢ 例: *Pascal*

```
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};
```

一个过程除自身定义的局部数据和全局定义的数据以外，还可以使用外围过程中声明的对象

不支持过程嵌套声明的语言

➢ 例： C

```
int  a[11];
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);
}
```

过程中使用的数据要么是自身定义的局部数据，要么是在所有过程之外定义的全局数据

无过程嵌套声明时的数据访问

- 变量的存储分配和访问
  - 全局变量被分配在静态区，使用静态确定的地址访问它们
  - 其它变量一定是栈顶活动的局部变量。可以通过运行时刻栈的*top_sp*指针访问它们

有过程嵌套声明时的数据访问

- 嵌套深度
  - 过程的嵌套深度
    - 不内嵌在任何其它过程中的过程，设其嵌套深度为1
    - 如果一个过程*p*在一个嵌套深度为*i*的过程中定义，则设定*p*的嵌套深度为*i* +1
  - 变量的嵌套深度
    - 将变量声明所在过程的嵌套深度作为该变量的嵌套深度

例

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

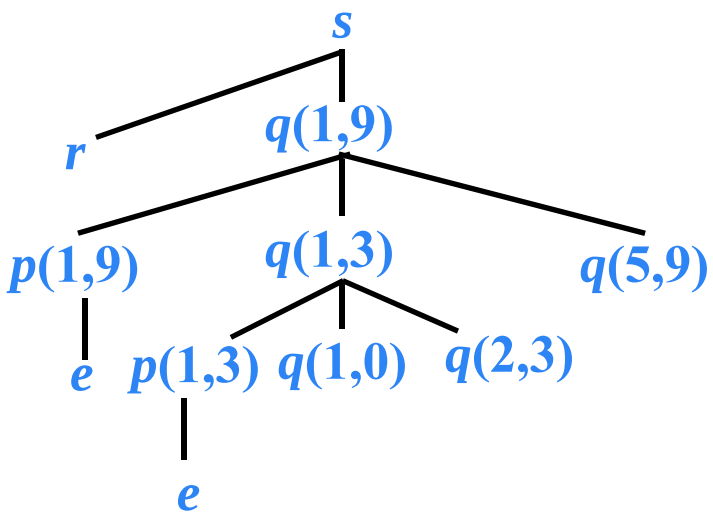
```
program sort ( input, output );
    var a: array[0..10] of integer;
        x: integer;
    procedure readarray;
            var i: integer;
            begin … a … end {readarray} ;
    procedure exchange(i, j:integer);
            begin x=a[i];a[i]=a[j];a[j]=x; end {exchange} ;
    procedure quicksort(m,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

访问链 (*Access Links*)

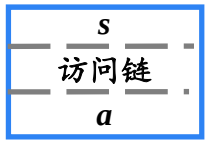
- 静态作用域规则：只要过程*b*的声明嵌套在过程*a*的声明中，过程*b*就可以访问过程*a*中声明的对象
- 可以在相互嵌套的过程的活动记录之间建立一种称为访问链(*Access link*)的指针，使得内嵌的过程可以访问外层过程中声明的对象
  - 如果过程*b*在源代码中直接嵌套在过程*a*中(*b*的嵌套深度比*a*的嵌套深度多1)，那么*b*的任何活动中的访问链都指向最近的*a*的活动

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};
```



控制栈

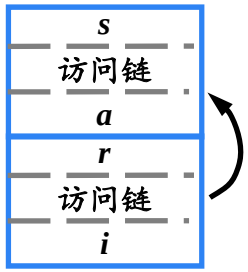


活动树

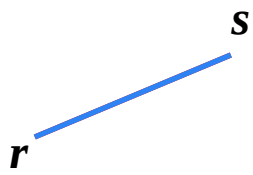
s

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

控制栈

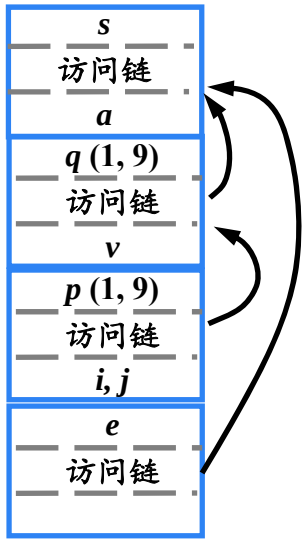


活动树

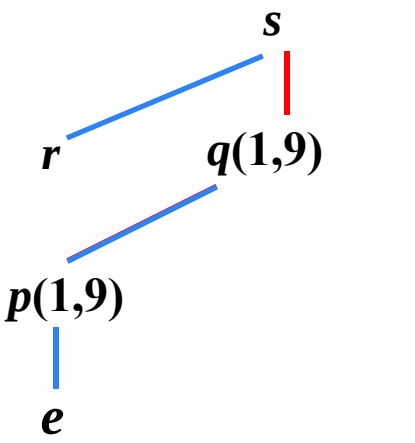


过程	嵌套深度
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

控制栈

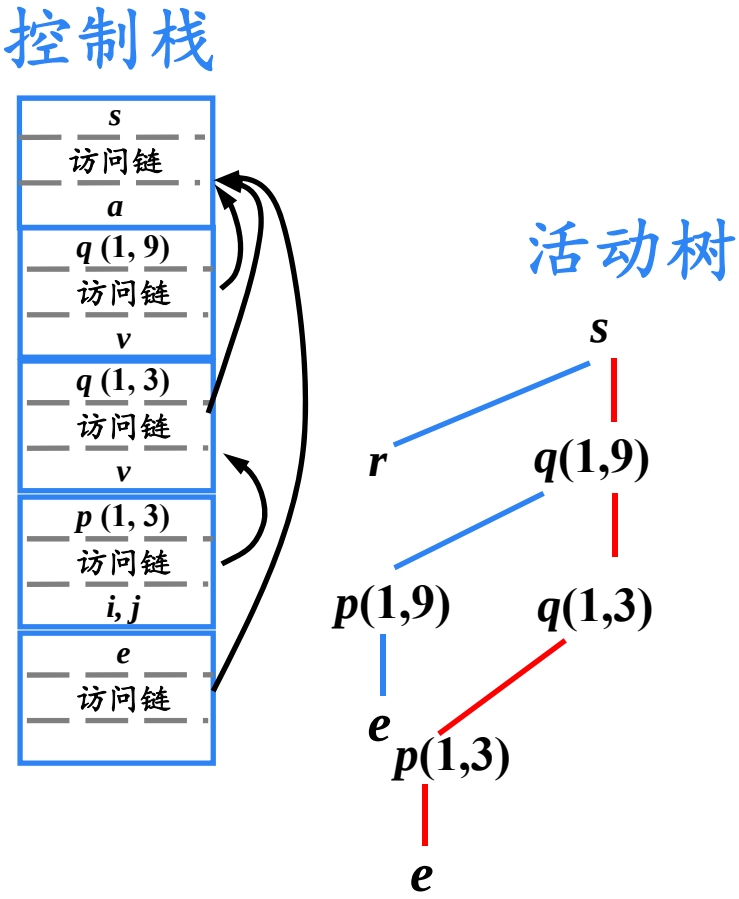


活动树



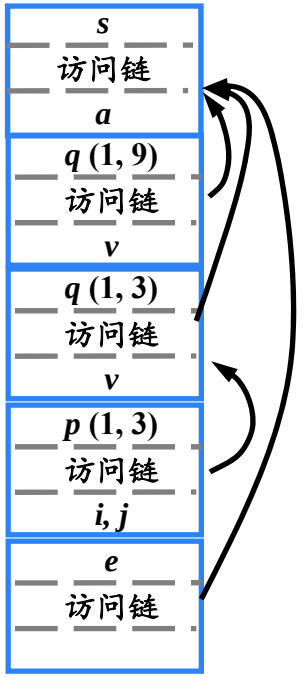
过程	嵌套深度
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

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



访问链的建立

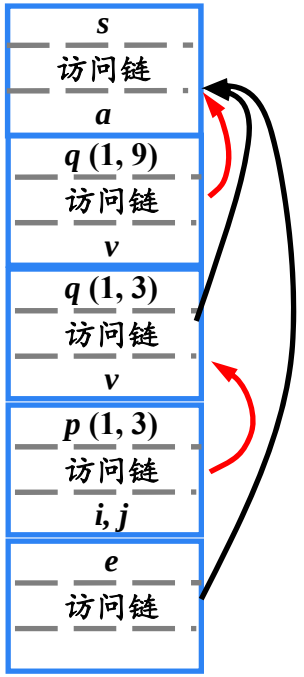
- 建立访问链的代码属于调用序列的一部分
- 假设嵌套深度为n_x的过程x调用嵌套深度为n_y的过程y ($x \rightarrow y$)
  - $n_x < n_y$的情况(外层调用内层)
    - y一定是直接在x中定义的 (例如：$s \rightarrow q$, $q \rightarrow p$) ，因此，$n_y = n_x + 1$
    - 在调用代码序列中增加一个步骤：在y的访问链中放置一个指向x的活动记录的指针



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

访问链的建立

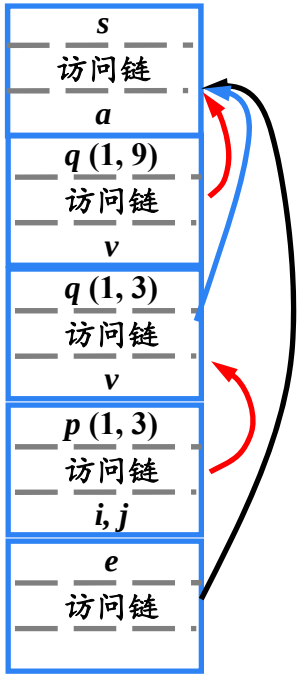
- 建立访问链的代码属于调用序列的一部分
- 假设嵌套深度为n_x的过程x调用嵌套深度为n_y的过程y $(x \rightarrow y)$
  - $n_x < n_y$的情况(外层调用内层)
  - $n_x = n_y$的情况(本层调用本层)
    - 例如：递归调用 $(q \rightarrow q)$
    - 被调用者的活动记录的访问链与调用者的活动记录的访问链是相同的，可以直接复制



过程	嵌套深度
sort	1
readarray	2
exchange	2
quicksort	2
partition	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必定嵌套在某个过程z中，而z中直接定义了被调用者y
    - 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



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

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

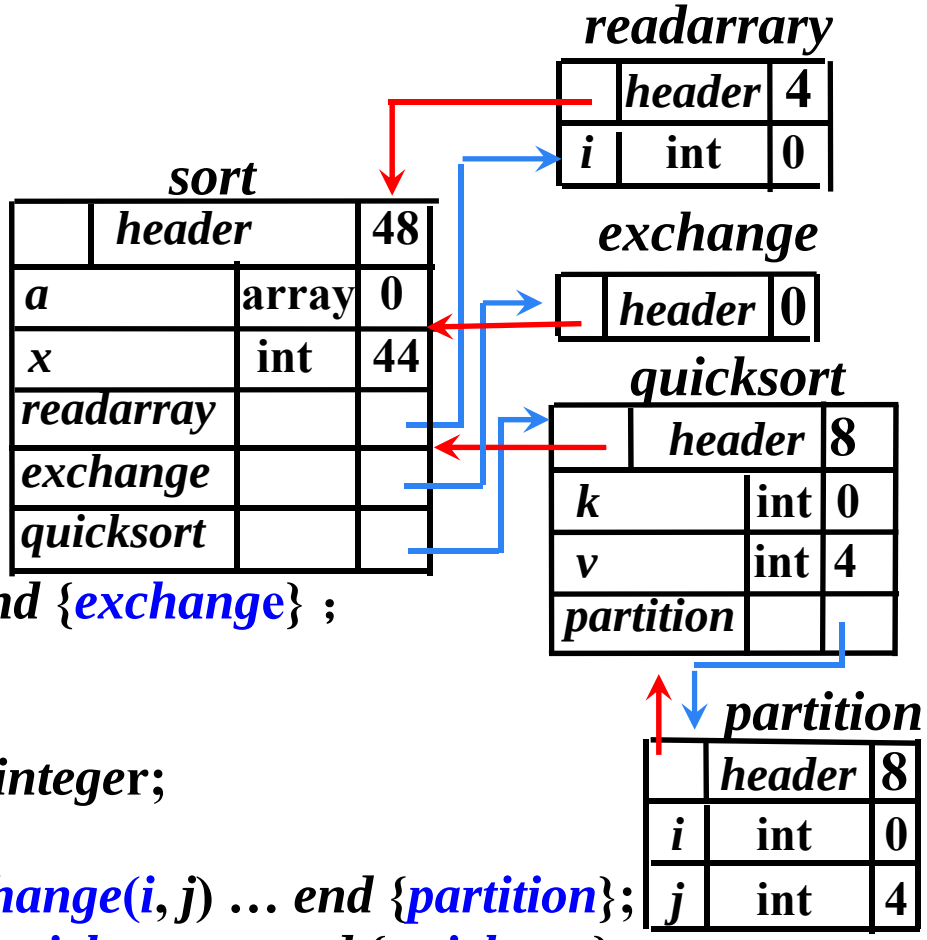
7.5 符号表

7.5 符号表

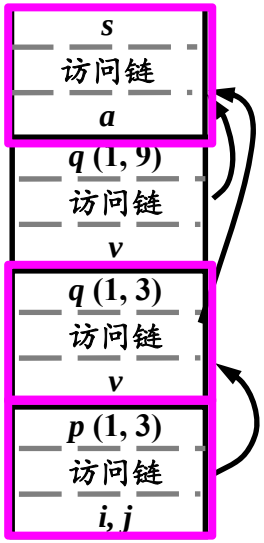
- 符号表的组织
  - 为每个作用域（程序块）建立一个独立的符号表

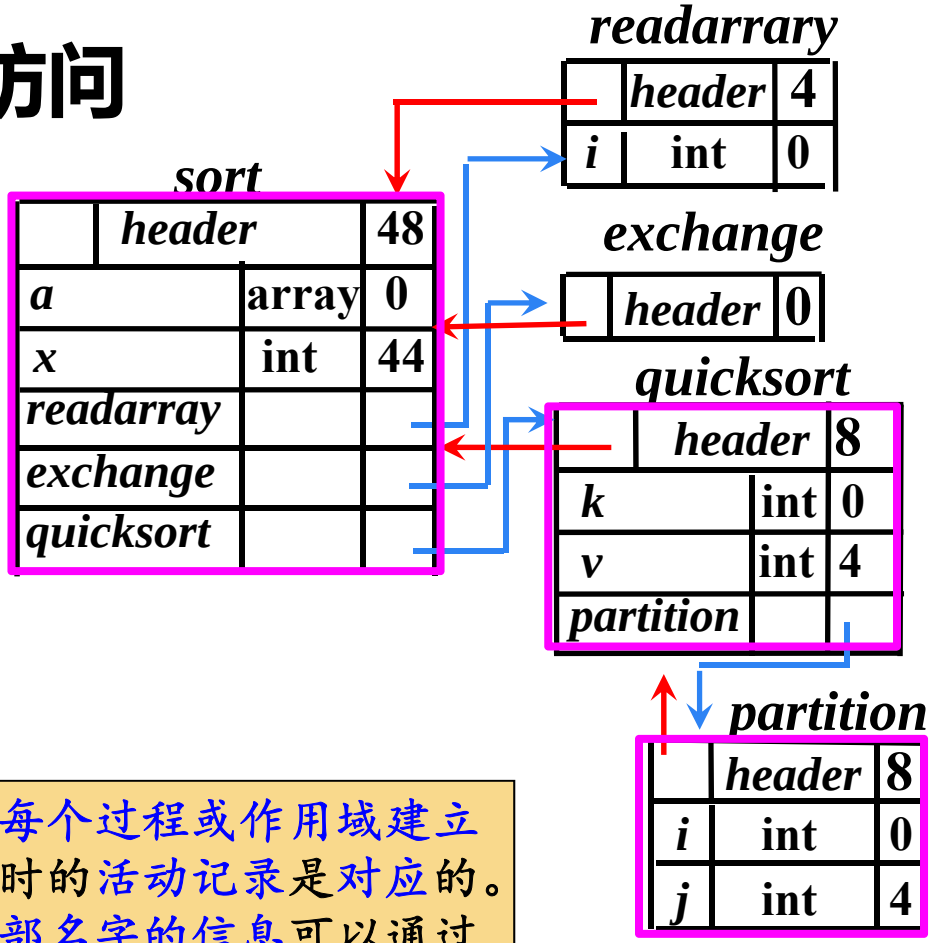
例

```
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};
```



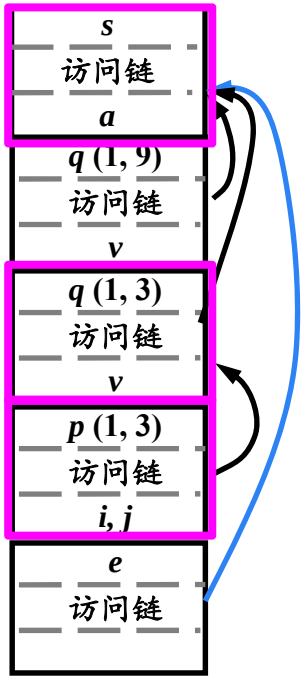
根据符号表进行数据访问

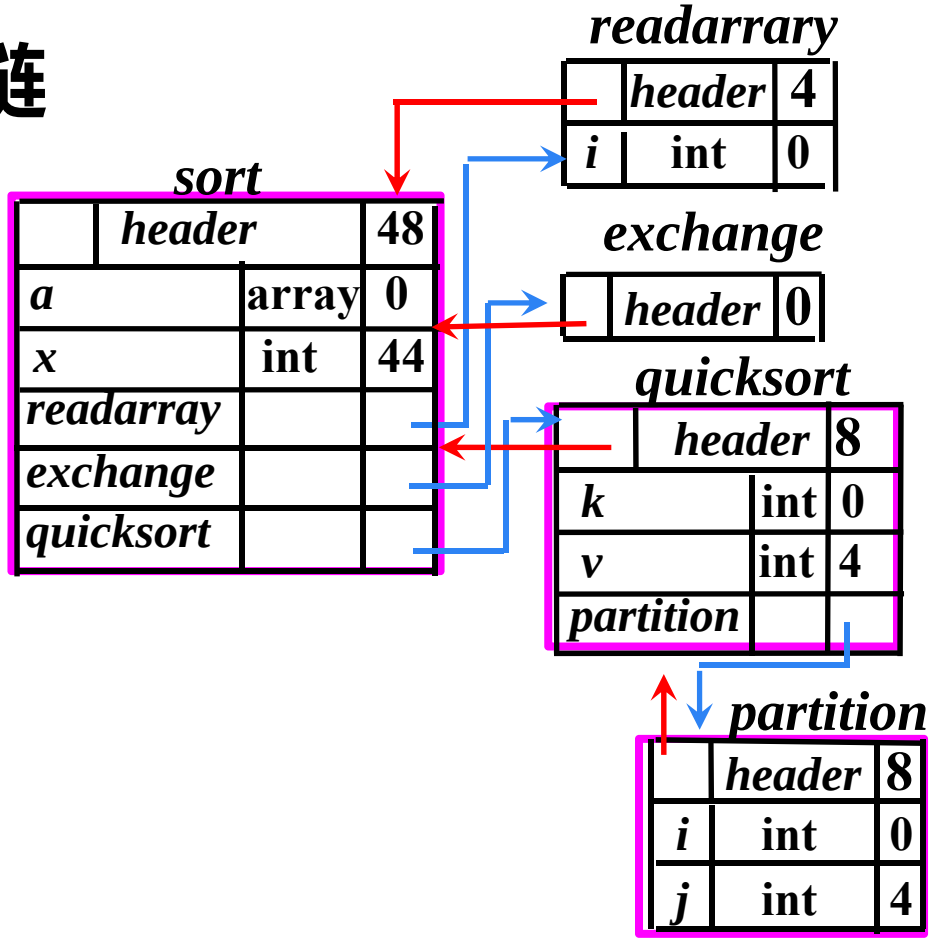




实际上，这种为每个过程或作用域建立的符号表与编译时的活动记录是对应的。一个过程的非局部名字的信息可以通过扫描外围过程的符号表而得到

根据符号表构造访问链


s
访问链
a
q (1, 9)
访问链
v
q (1, 3)
访问链
v
p (1, 3)
访问链
i, j
e
访问链


readarrary
header 4
i int 0
sort
header 48
a array 0
x int 44
readarray
exchange
quicksort
exchange
header 0
quicksort
header 8
k int 0
v int 4
partition
partition
header 8
i int 0
j int 4

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
  - 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”

符号表的建立

- 嵌套过程声明语句的文法

  $P \rightarrow D$

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

- 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*声明语句的处理

嵌套过程声明语句的*SDT*

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

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

$D \rightarrow D_1\ D_2$

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

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

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

mktable(previous)
创建一个新的符号表，并返回指向新表的指针。参数*previous*指向先前创建的符号表(外围过程的符号表)

addwidth(table, width)
将*table*指向的符号表中所有表项的宽度之和*width*记录在符号表的表头中

enterproc(table, name, newtable)
在*table*指向的符号表中为过程*name*建立一条记录，*newtable*指向过程*name*的符号表

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

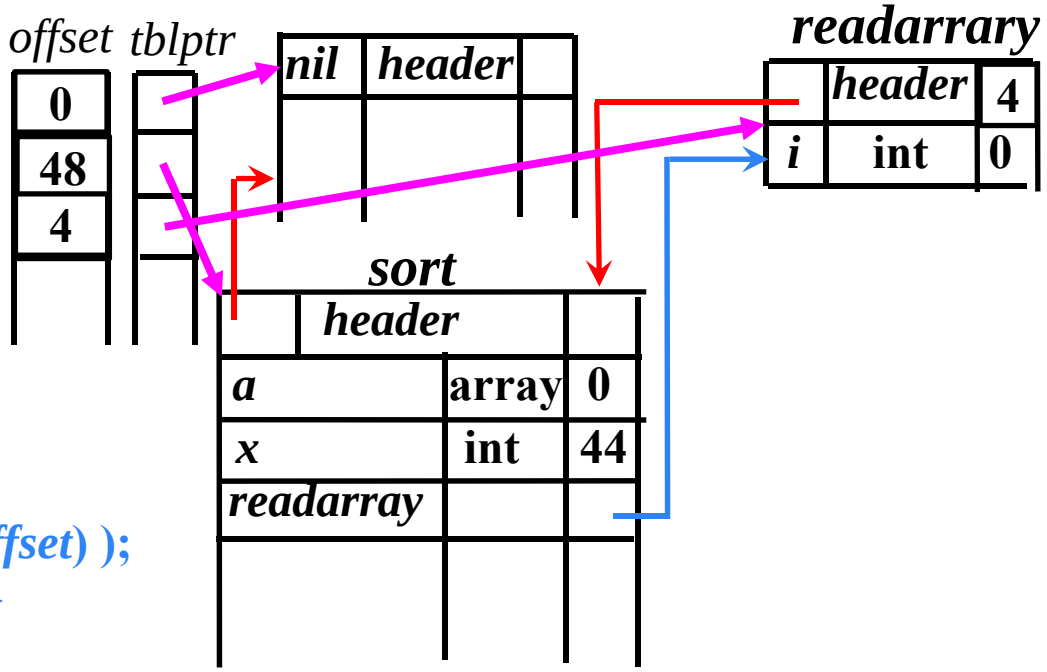
例

```
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; {…}
        {…}
    {…}
```



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

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

$D \rightarrow D_1\ D_2$

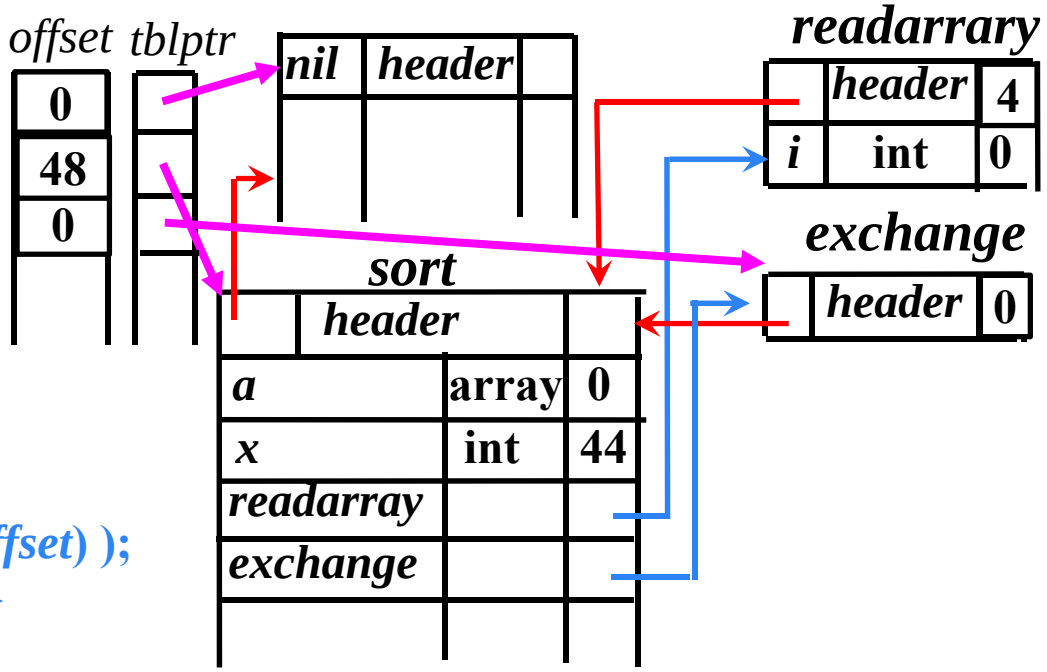
$D_p \rightarrow$ proc id ; $N\ D_1\ S$ { *t = top(tblptr); addwidth(t, top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), id.lexeme, t); }*

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

$N \rightarrow \varepsilon$ { *t = mktable(top(tblptr)); push(t, tblptr); push(0, 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; {…}
        {…}
    {…}
```



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

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

$D \rightarrow D_1\ D_2$

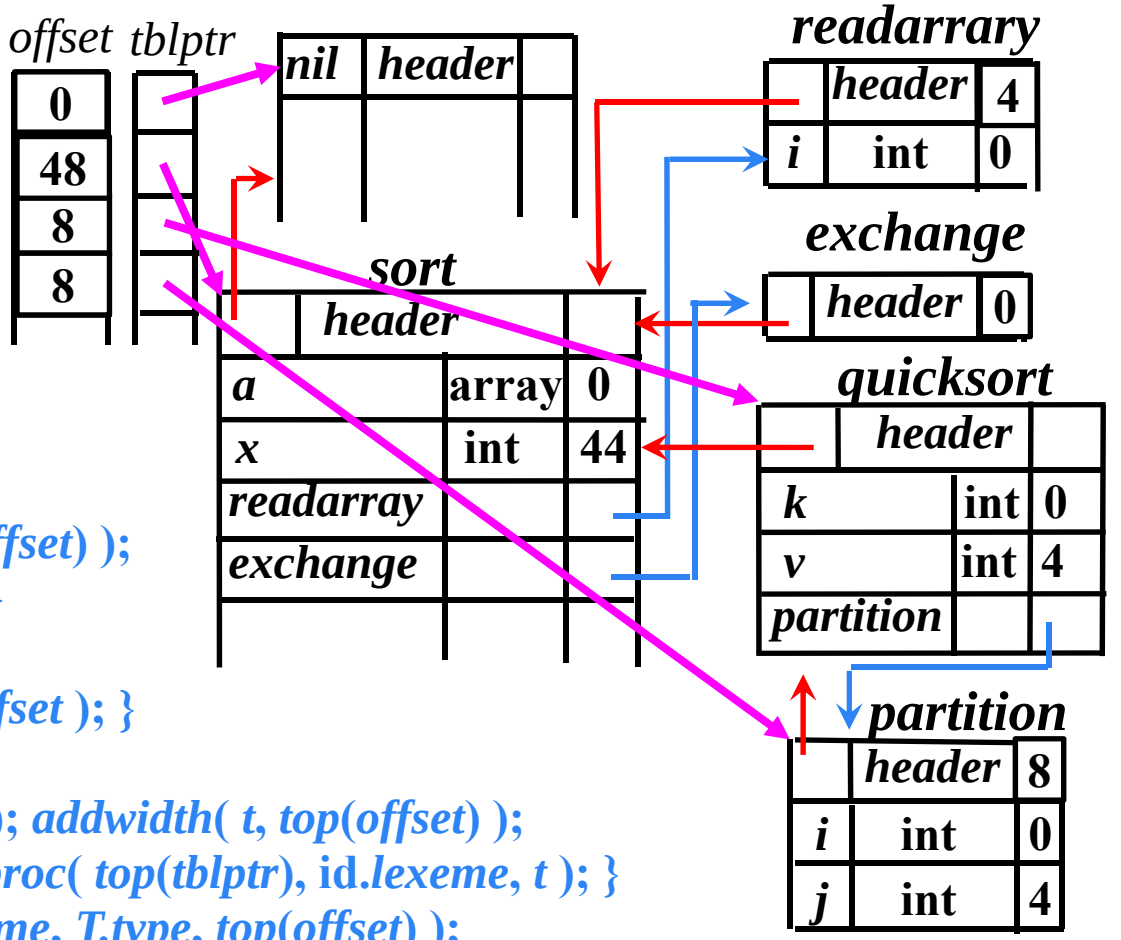
$D_p \rightarrow$ proc id ; $N\ D_1\ S$ { *t = top(tblptr); addwidth(t, top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), id.lexeme, t); }*

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

$N \rightarrow \varepsilon$ { *t = mktable(top(tblptr)); push(t, tblptr); push(0, 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; {…}
        {…}
    {…}
```



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

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

$D \rightarrow D_1\ D_2$

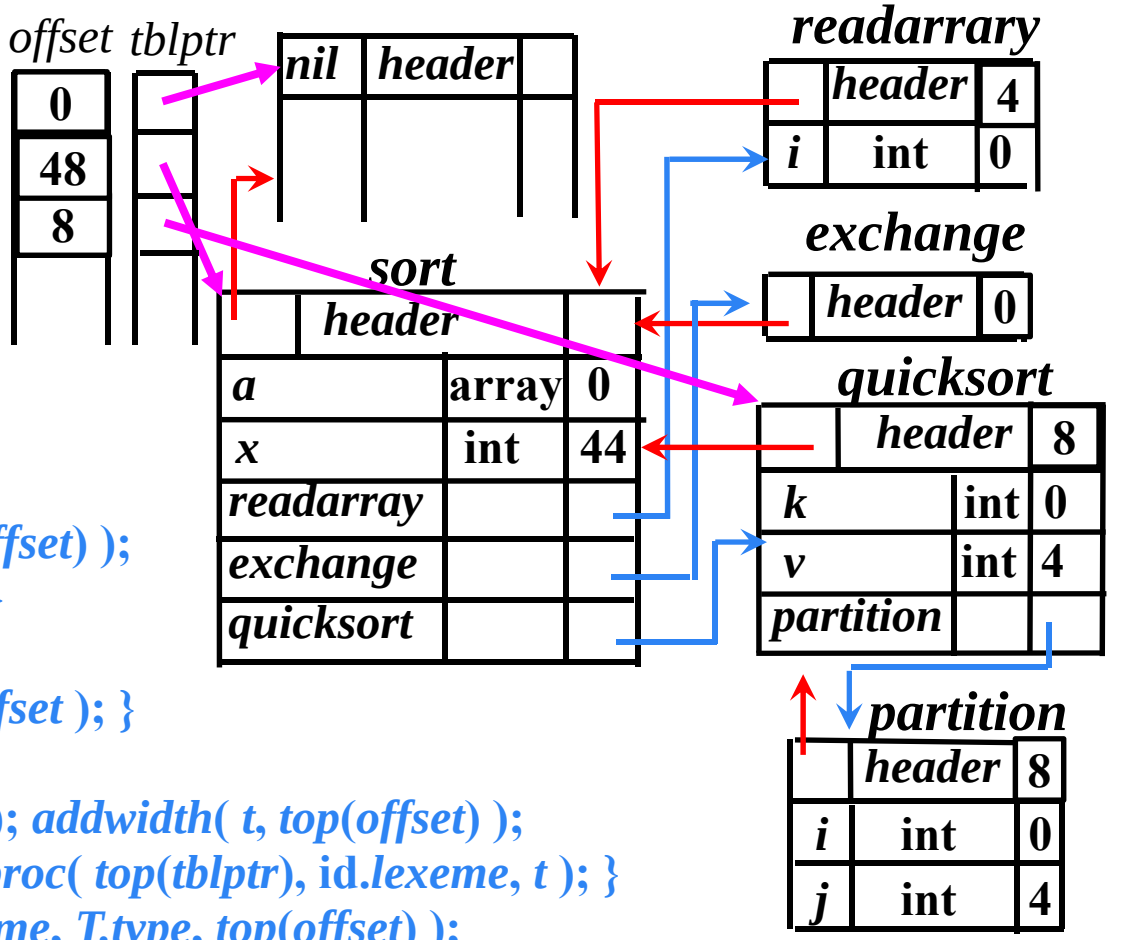
$D_p \rightarrow$ proc id ; $N\ D_1\ S$ { *t = top(tblptr); addwidth(t, top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), id.lexeme, t); }*

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

$N \rightarrow \varepsilon$ { *t = mktable(top(tblptr)); push(t, tblptr); push(0, 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; {…}
        {…}
    {…}
```



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

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

$D \rightarrow D_1\ D_2$

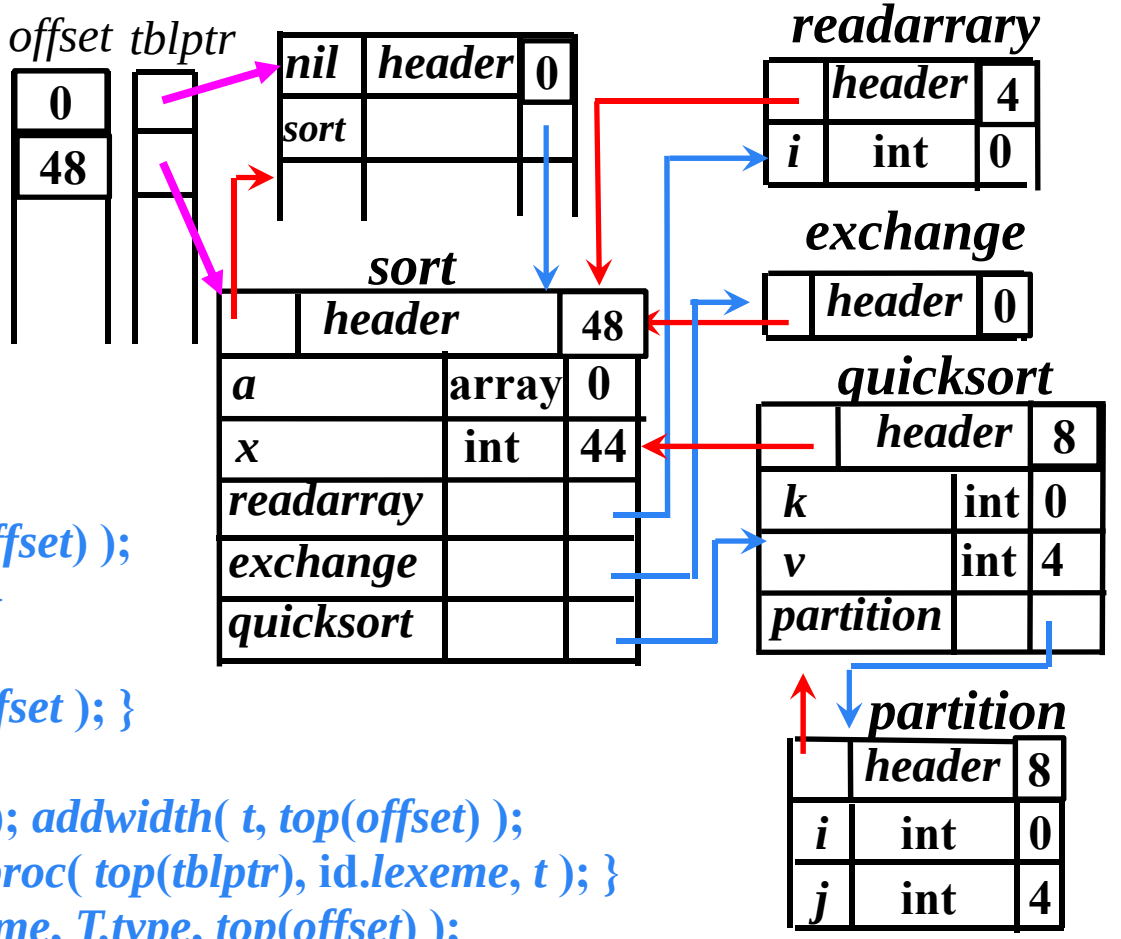
$D_p \rightarrow$ proc id ; $N\ D_1\ S$ { *t = top(tblptr); addwidth(t, top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), id.lexeme, t); }*

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

$N \rightarrow \varepsilon$ { *t = mktable(top(tblptr)); push(t, tblptr); push(0, 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; {…}
        {…}
    {…}
```



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

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

$D \rightarrow D_1\ D_2$

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

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

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

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