第九章：运行存储分配

提纲

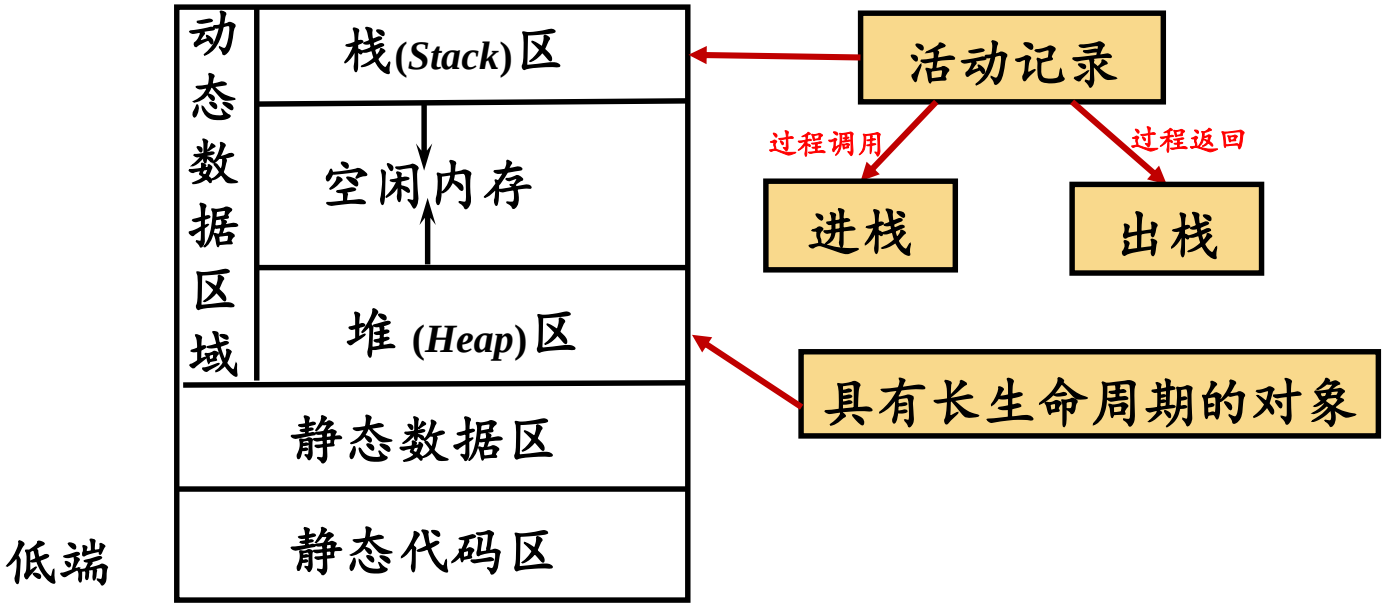
9.1 存储组织
9.2 静态存储分配
9.3 动态存储分配
9.4 非局部数据的访问
9.5 符号表

运行存储分配策略

- 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

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



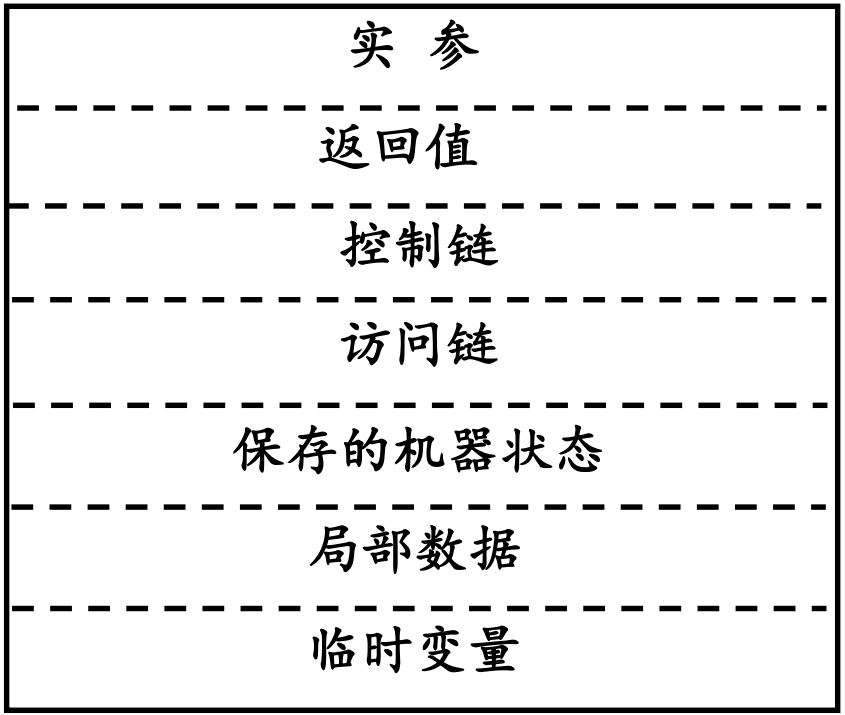
堆栈对比

项目	申请方式	内存大小	使用效率	是否连续	地址方向	存储方式
栈（stack）	自动申请释放	小	高效	连续	由高地址向低地址	先进后出
堆（heap）	手动申请释放	大	缓慢	不连续	由低地址向高地址	先进先出

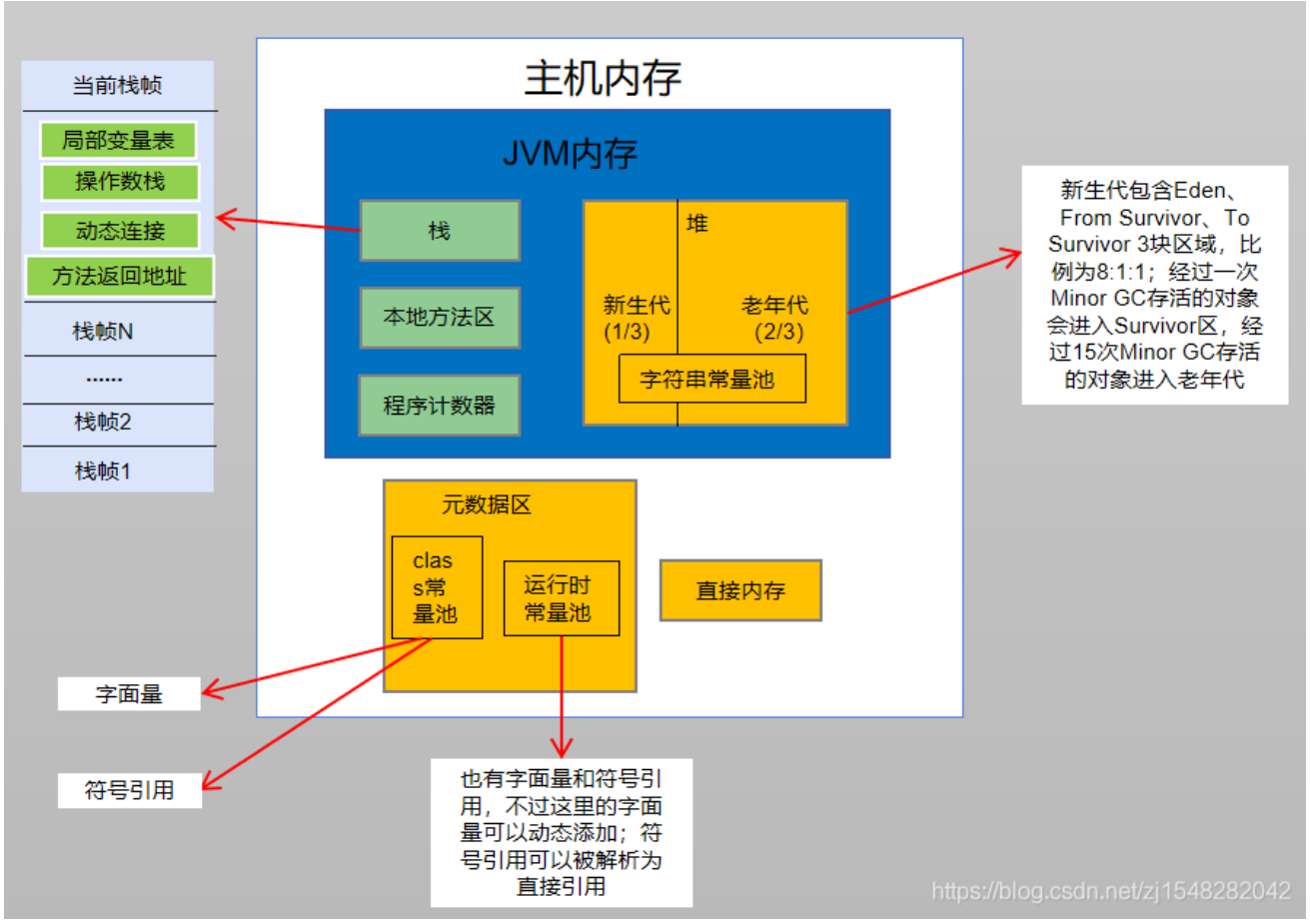
活动记录

- 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*)

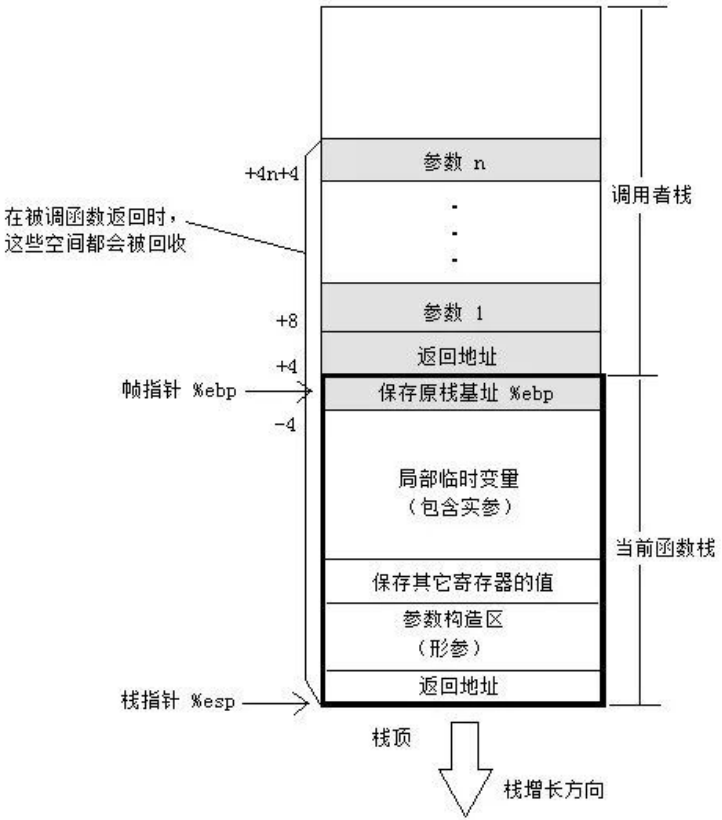
活动记录的一般形式



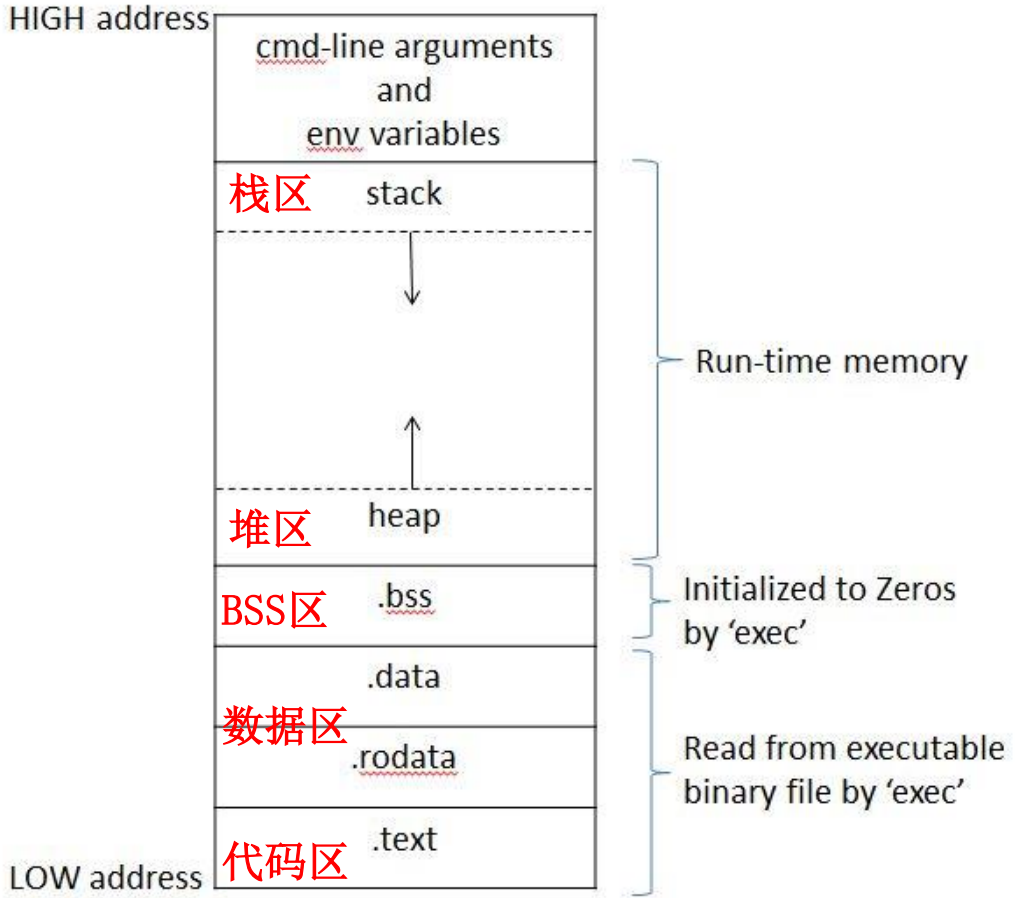
活动记录的一般形式 Java



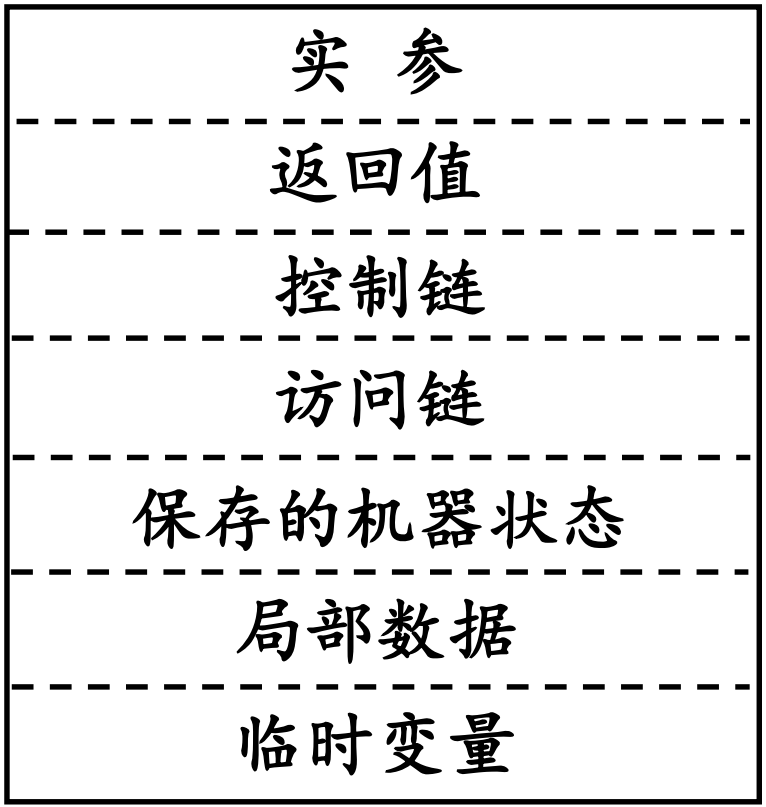
活动记录的一般形式


参数 n
+4n+4
在被调函数返回时，
这些空间都会被回收
调用者栈
参数 1
+8
返回地址
+4
帧指针 %ebp
保存原栈基址 %ebp
-4
局部临时变量
（包含实参）
当前函数栈
保存其它寄存器的值
参数构造区
（形参）
返回地址
栈指针 %esp
栈顶
栈增长方向

C++


HIGH address
cmd-line arguments
and
env variables
栈区
stack
堆区
heap
Run-time memory
BSS区
.bss
Initialized to Zeros
by 'exec'
.data
数据区
.rodata
Read from executable
binary file by 'exec'
代码区
.text
LOW address

活动记录的一般形式



调用过程提供给被调用过程的参数

被调用过程返回给调用过程的值

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

通常包括返回地址和一些寄存器中的内容

在该过程中声明的数据

比如表达式求值过程中产生的临时变量

存储分配策略

- 静态存储分配
  - 顺序分配法
  - 层次分配法
- 动态存储分配
  - 栈式存储分配
  - 堆式存储分配

提纲

9.1 存储组织
9.2 静态存储分配
9.3 动态存储分配
9.4 非局部数据的访问
9.5 符号表

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

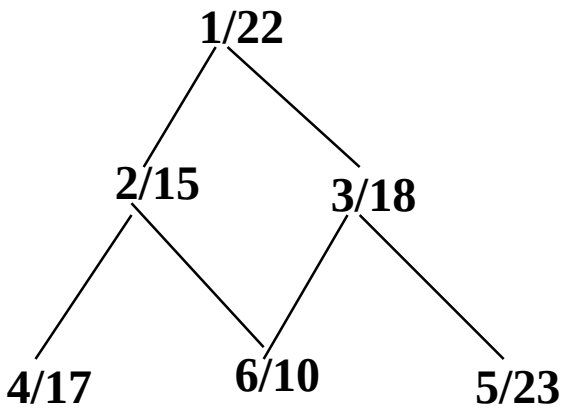
- 适合静态存储分配的语言必须满足以下条件
  - 数组上下界必须是常数
  - 不允许过程的递归调用
  - 不允许用户动态建立数据实体
- 满足这些条件的语言有BASIC和FORTRAN等

常用的静态存储分配策略

- 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

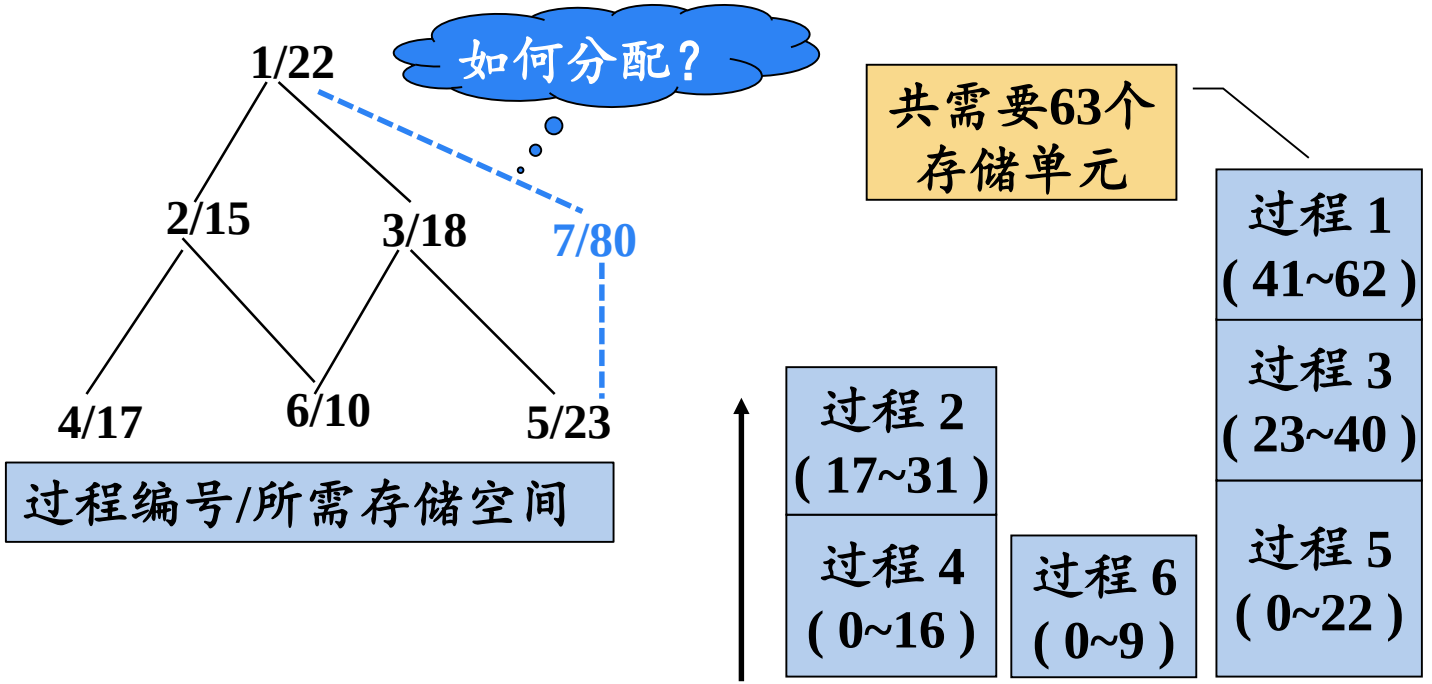
能用更少的空间么？

优点：处理上简单

缺点：对内存空间的使用不够经济合理

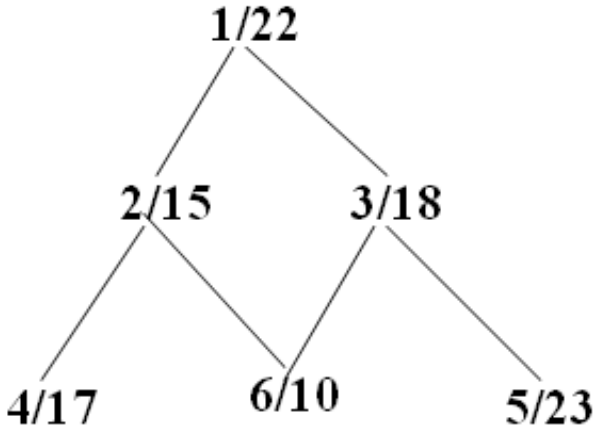
层次分配法

- 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

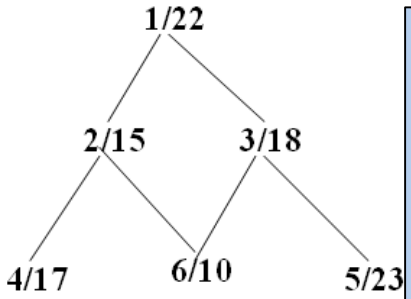


	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	0	1	0	1
3	0	0	0	0	1	1
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0

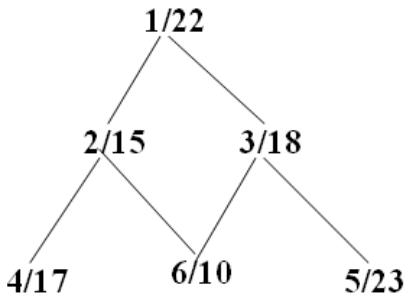
- *Units*[*n*]：过程所需内存数量

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 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

9.1 存储组织
9.2 静态存储分配
9.3 动态存储分配
9.4 非局部数据的访问
9.5 符号表

动态存储分配

- 分配策略
  - 栈式存储分配
    - 有些语言使用过程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
    - 这种安排允许活跃时段不交叠的多个过程调用之间共享空间
  - 堆式存储分配

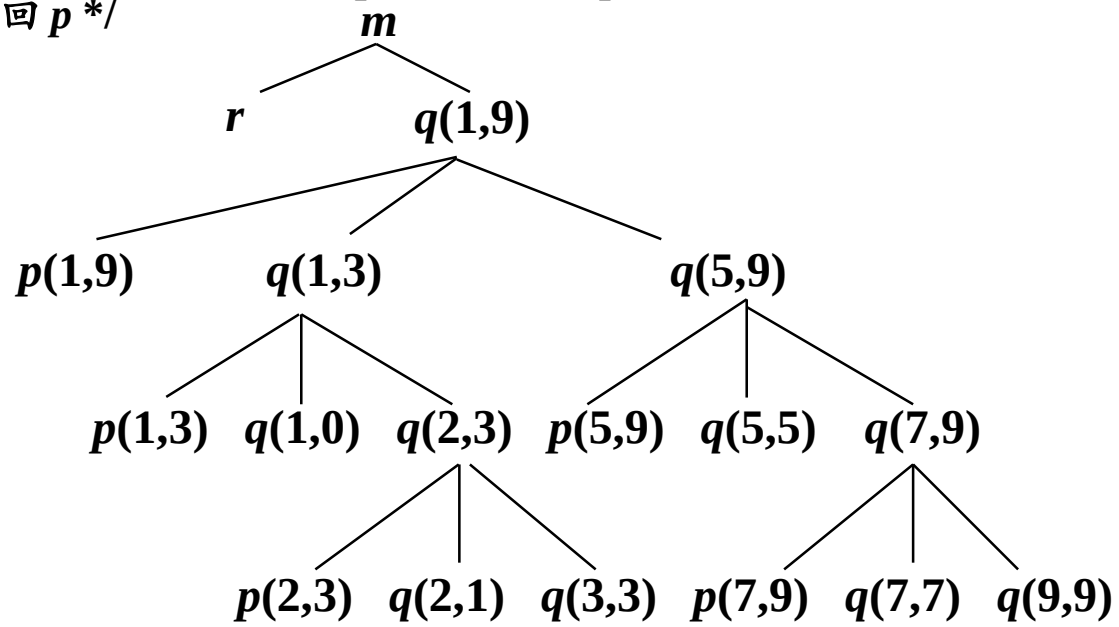
活动树

- 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（activation tree）
- 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

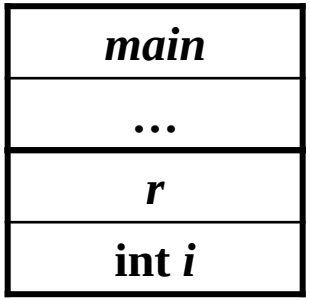
```
int  a[11];
void  readArray( ) /*将9个整数读入到a[1],…,a[9]中 */
{       int  i;
        …
}
int  partition(int  m,  int  n)
{       /*选择一个分割值v，划分a[m…n]，使得a[m…p-1]小于v，a[p]= v，
          a[p+1…n]大于等于v。返回 p */
        …
}
void  quicksort(int  m,  int  n)
{       int  i;
        if (n>m) {
            i=partition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

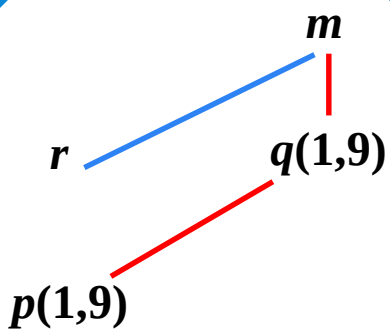


活动树

控制栈



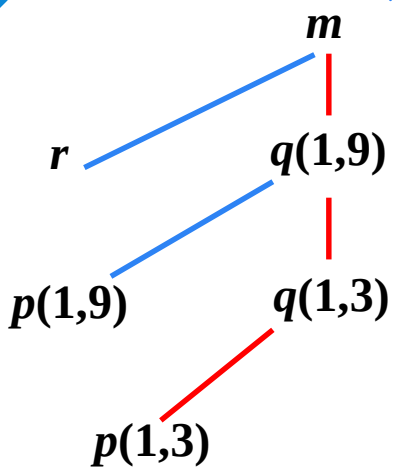
活动树



控制栈

main
…
q(1,9)
int *i*
p(1,9)
int *i*

活动树

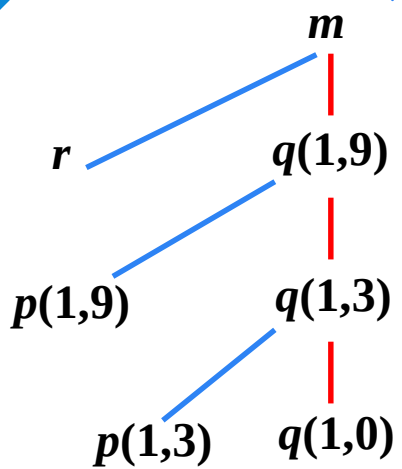


控制栈

main
…
q(1,9)
int i
q(1,3)
int i
p(1,3)
int i

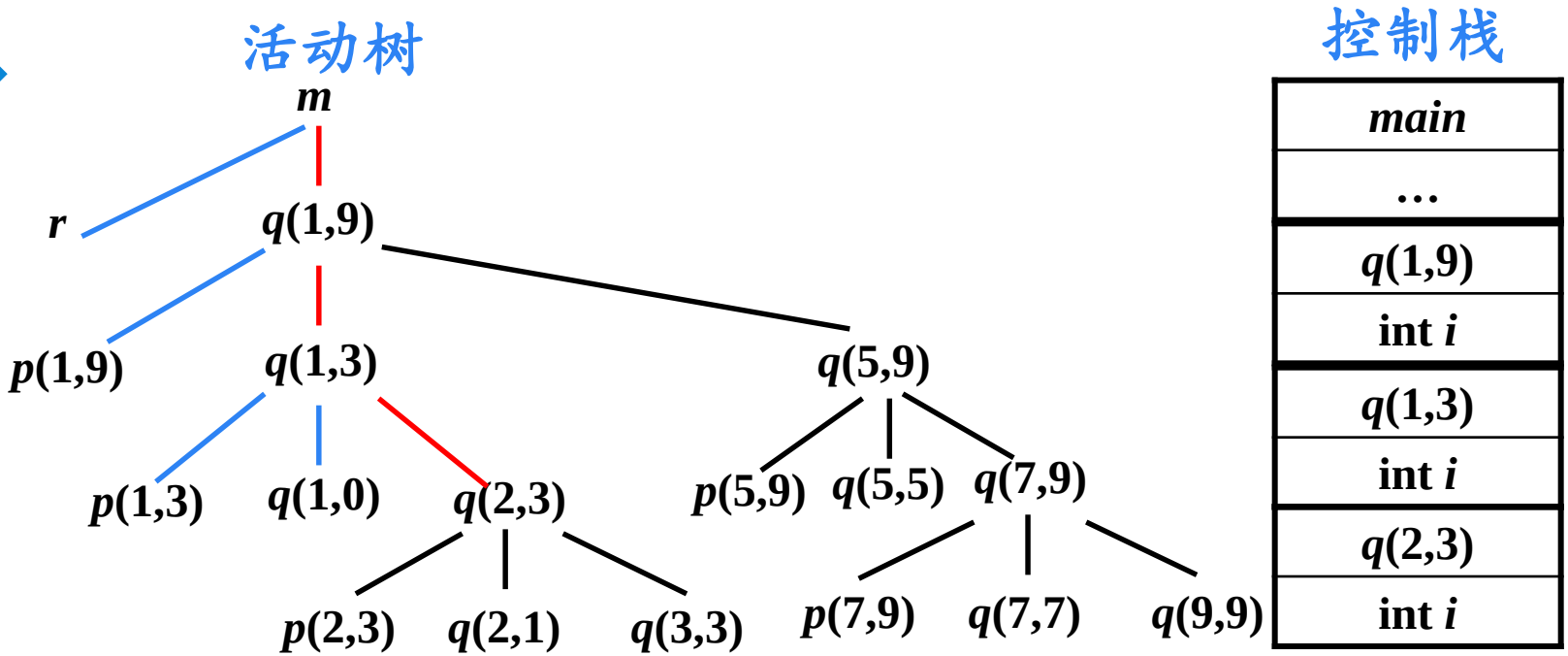
当一个过程是递归的时候，常常会有该过程的多个活动记录同时出现在栈中

活动树



控制栈

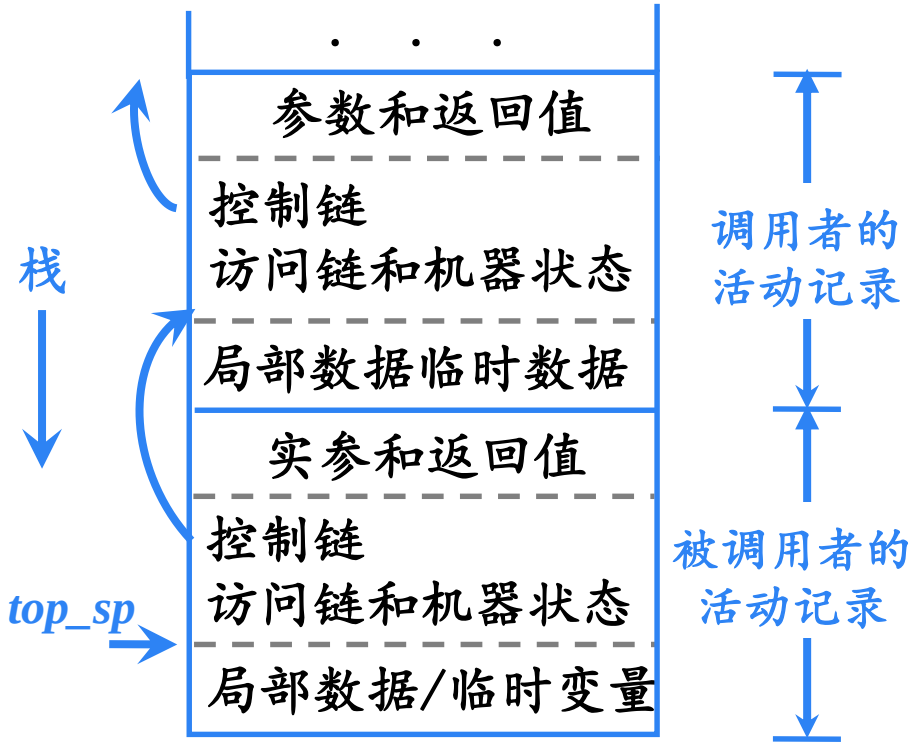
main
…
q(1,9)
int *i*
q(1,3)
int *i*
q(1,0)
int *i*



- 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

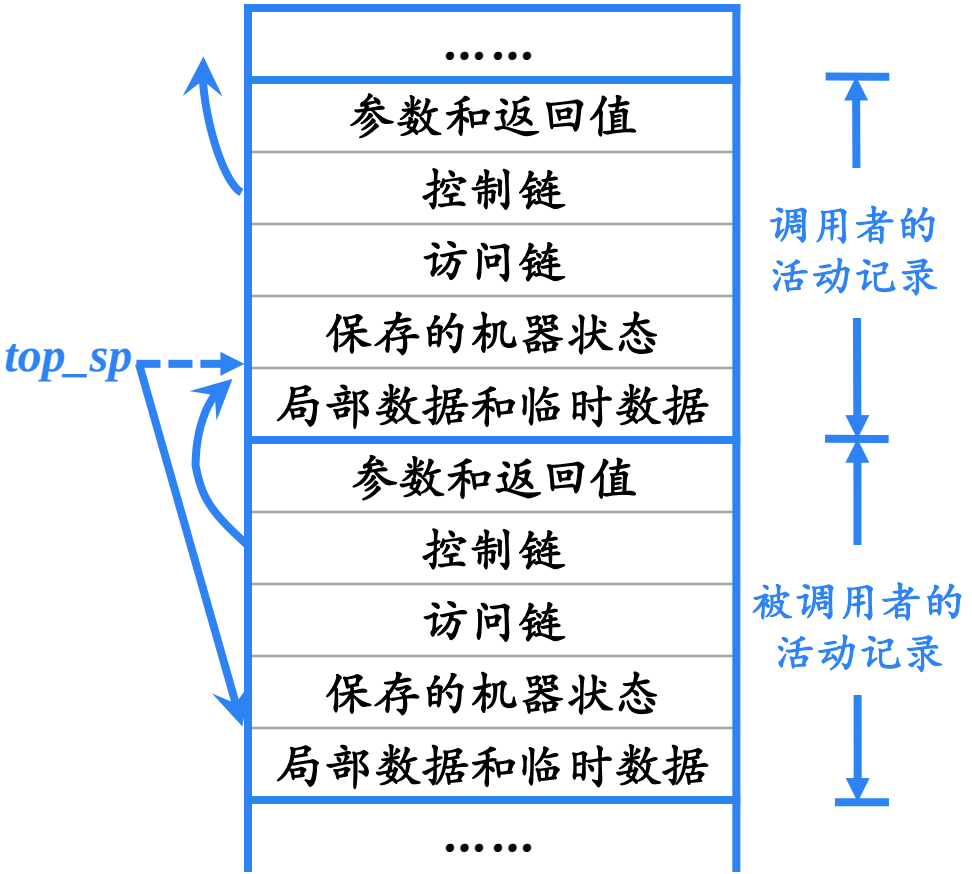


调用代码序列和返回代码序列

- 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
  - 调用代码序列
    - 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
  - 返回代码序列
    - 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。返回代码序列也是如此

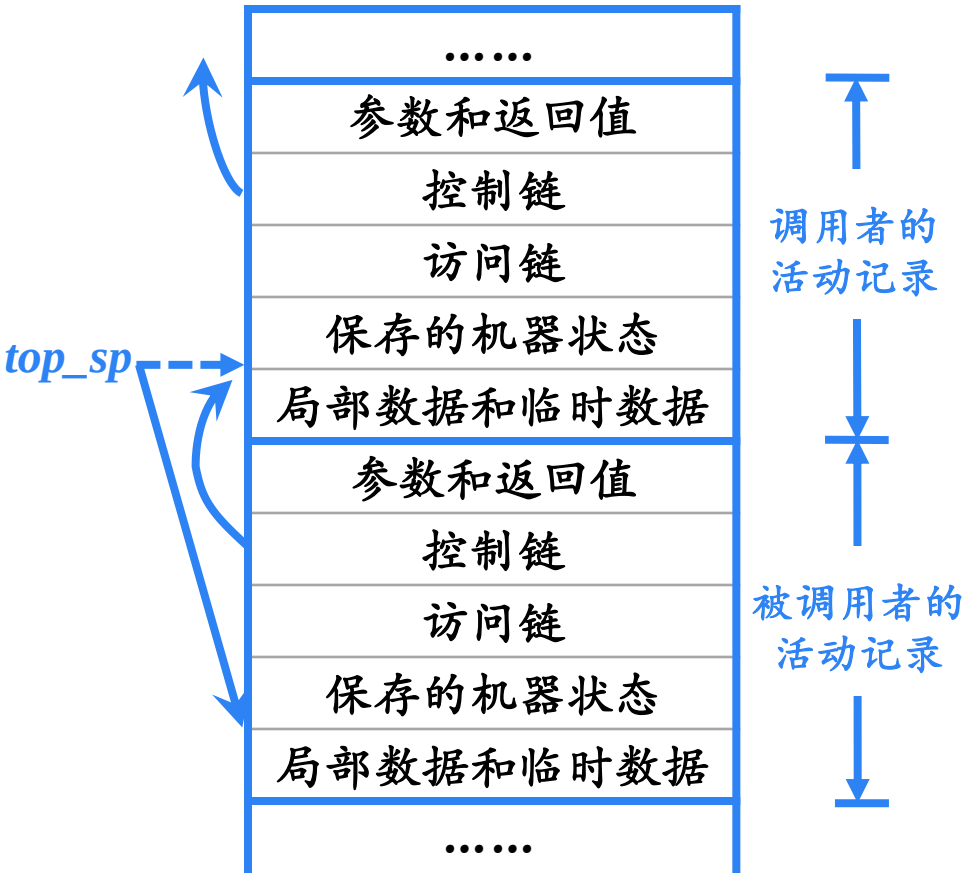
调用代码序列

- 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

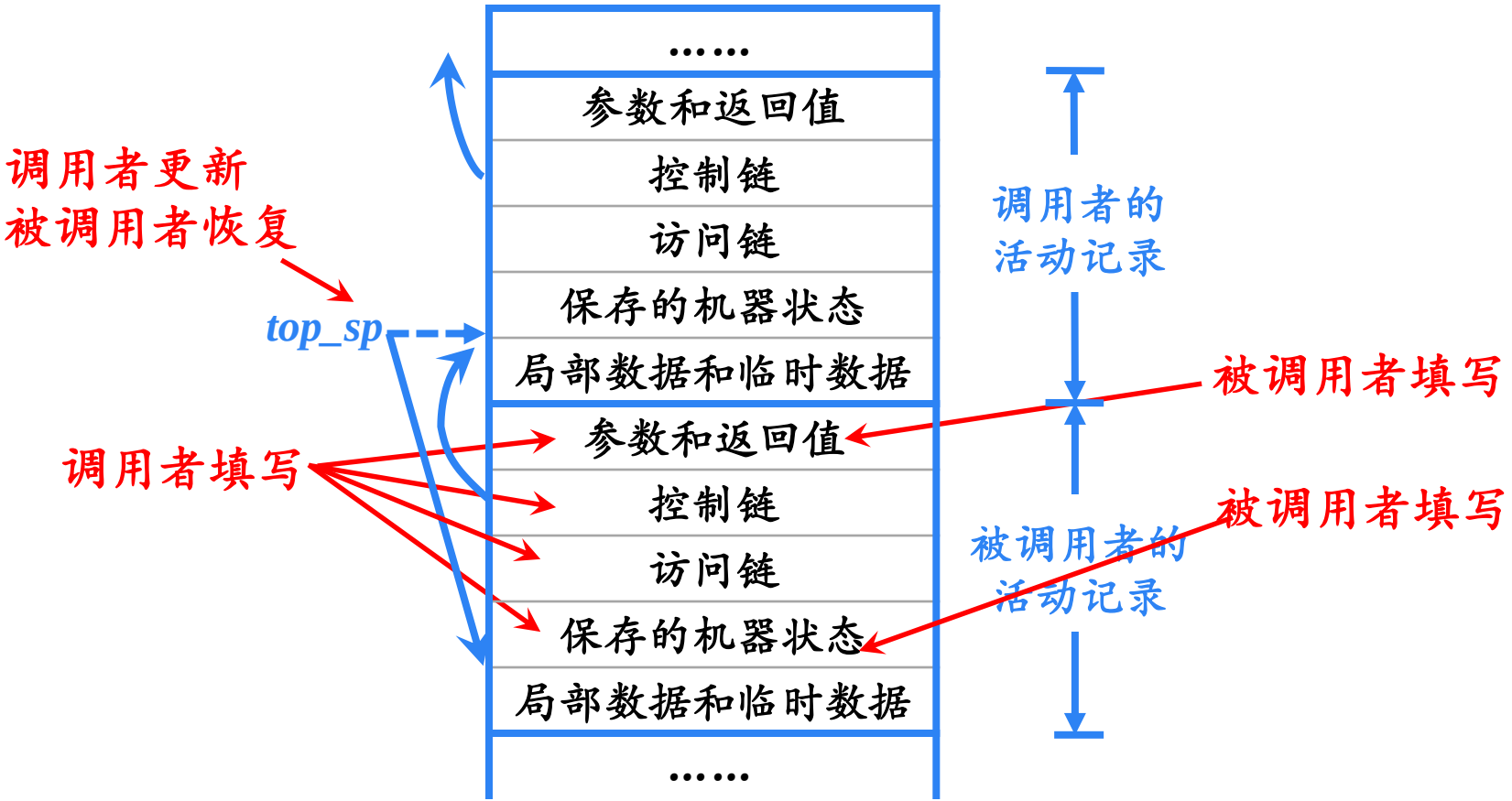


返回代码序列

- 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



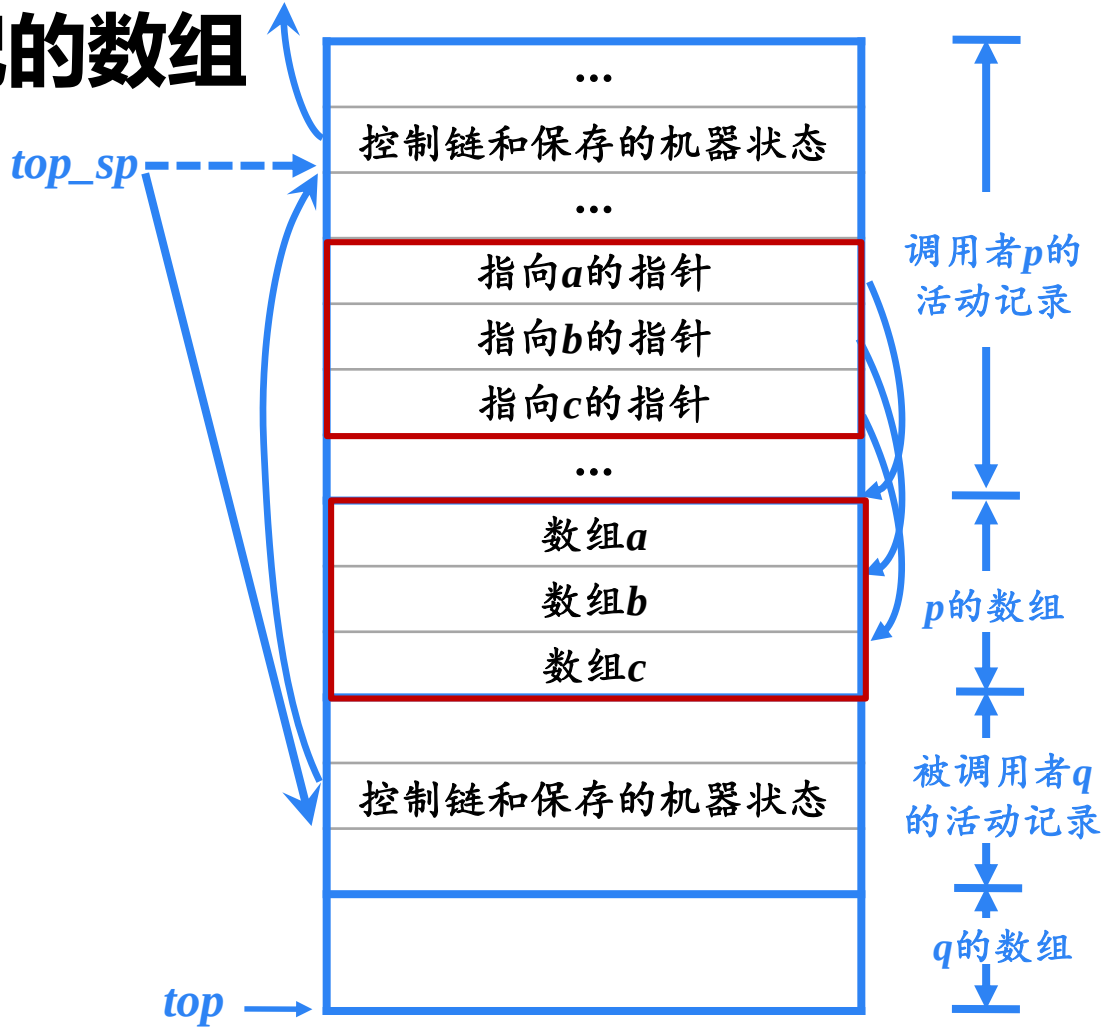
调用者和被调用者之间的任务划分



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

动态存储分配

- 分配策略
  - 栈式存储分配
    - 有些语言使用过程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
    - 这种安排允许活跃时段不交叠的多个过程调用之间共享空间
  - 堆式存储分配

堆管理

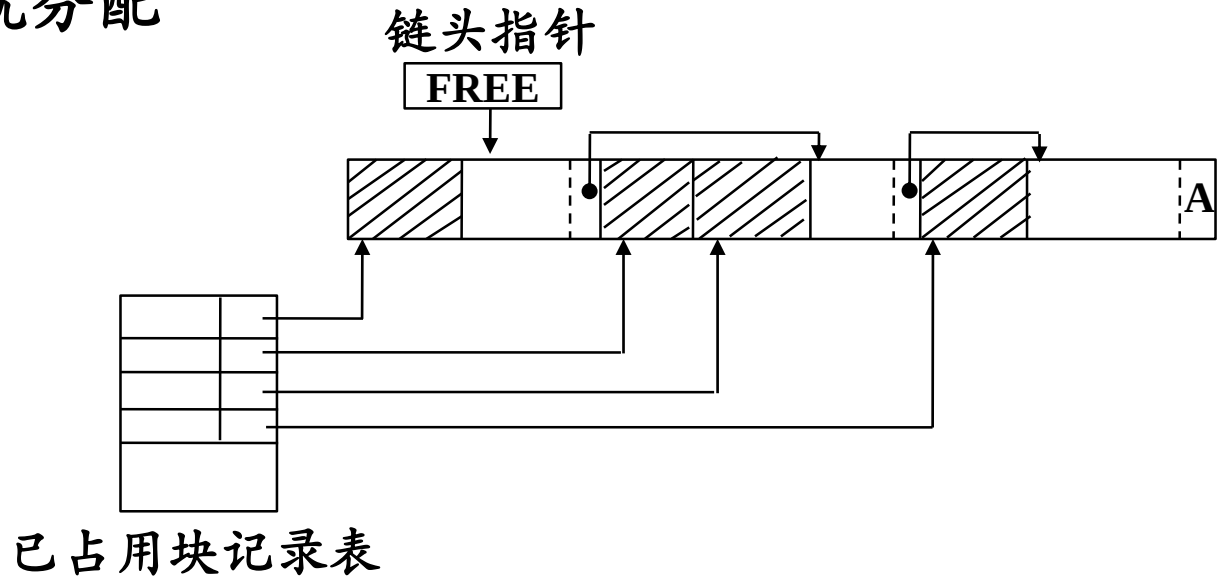
- 堆是存储空间的一部分，它用来存储那些生命周期不确定，或者将生存到被程序显示删除为止的数据。

  malloc， new……

- 存储管理器（**memory manager**），是分配和回收堆区空间的子系统，是应用程序和操作系统之间的一个接口。

存储管理器

- 分配：把连续存储区域分成块，当活动记录或其它对象请求内存时就分配



- 回收：把被回收的空间返还到空闲空间的缓冲池中，可以被其他请求复用，但通常不会将内存返还给OS。

- 分配
  - 设当前自由块总长为*M*，欲申请长度为*n*
    - 如果存在若干个长度大于或等于*n*的存储块，可按以下策略之一进行存储分配
      - 取长度*m*满足需求的第1个自由块，将长度为*m-n*的剩余部分仍放在自由链中 – first fit
      - 取长度*m*满足需求的最小的自由块 – best fit
      - 取长度*m*满足需求的最大的自由块 – max fit

- 分配
  - 设当前自由块总长为M，欲申请长度为n
    - 如果存在若干个长度大于或等于n的存储块，可按以下策略之一进行存储分配
    - 如果不存在长度大于或等于n的存储块
      - 如果$M \geq n$，将自由块在堆中进行移位和重组（对各有关部分都需作相应的修改，是一件十分复杂和困难的工作）
      - 如果$M < n$，则应采用更复杂的策略来解决堆的管理问题

- 分配
- 回收
  - 只需将被释放的存储块作为新的自由块插入自由链中，并删除已占块记录表中相应的记录即可

- 为实现堆式存储管理
  - 须完成大量的辅助操作
    - 如排序、查表、填表、插入、删除、……
  - 其空间和时间的开销较大，期望存储管理器具有如下特性：
    - 空间效率：存储管理器应该能够使一个程序所需的堆区空间的总量达到最小。空间效率是通过使存储碎片达到最少而得到的。

- 为实现堆式存储管理
  - 须完成大量的辅助操作
    - 如排序、查表、填表、插入、删除、……
  - 其空间和时间的开销较大，期望存储管理器具有如下特性：
    - 空间效率
    - 程序效率：存储管理器应该充分利用存储子系统，使程序可以运行的更快。

通常认为程序90%的时间用来执行10%的代码

- 为实现堆式存储管理
  - 须完成大量的辅助操作
    - 如排序、查表、填表、插入、删除、……
  - 其空间和时间的开销较大，期望存储管理器具有如下特性：
    - 空间效率
    - 程序效率
    - 低开销：存储分配和回收是程序中的常用操作，高效很重要。分配开销由小型请求决定，大型对象开销相对不重要。

练习

1.编译方法中，动态存储分配的含义是(　)

A：在编译阶段为源程序中的量进行分配

B：在编译阶段为源程序中的量进行分配，运行时可动态调整

C：在运行阶段为源程序中的量进行分配

D：都不正确

2.以下说法正确的是(　)

A：对于数据空间的存储分配，FORTRAN采用动态存储分配策略

B：对于数据空间的存储分配，C语言仅采用栈式存储分配策略

C：动态存储分配是指在编译阶段对源程序中的量进行分配，以使目标代码在运行时加快运行速度

D：如果两个临时变量的作用域不相交，则可以将它们分配在同一单元中

练习

3. PASCAL语言中过程声明的局部变量地址分配在(　　)

A：调用者的数据区中　　B：被调用者的数据区中

C：主程序的数据区中　　D：公共数据区中

4.以下说法正确的是(　　)

A：编译程序除解决源程序中用户定义的量在运行时刻的存储组织与分配问题之外，还应完成为临时变量和参与运算的寄存器组织好存储空间的任务

B：由于C语言的函数允许递归调用，因此对C语言中的所有变量的单元分配一律采用动态分配方式

C：动态数组的存储空间在编译时即可完全确定

D：“运算符与运算对象类型不符”属于语法错误

练习

5. C语言函数f的定义如下：

```
int f ( int x , *py , **ppz ) {
    **ppz += 1;  *py += 2;  x += 3;  return x + *py + **ppz;
}
```

变量a是一个指向b的指针；变量b是一个指向c的指针，而c是一个当前值为4的整数变量。如果我们调用f（c，b，a），返回值是什么？

课堂作业

6. 右图中是递归计算Fiabonacci数列的C语言代码。假设 f 的活动记录按顺序包含下列元素：（返回值，参数 n，局部变量 s，局部变量 t）。通常在活动记录中还会有其它元素。下面的问题假设初始调用是f (5) 。

1）给出完整的活动树。

2）当第1个f (1)调用即将返回时，运行时刻栈和其中的活动记录是什么样子的？

3）当第5个f (1)调用即将返回时，运行时刻栈和其中的活动记录是什么样子的？

```
int f( int n ) {
    int t,s;
    if (n < 2) return 1;
    s = f(n-1);
    t = f(n-2);
    return s+t;
}
```

提纲

9.1 存储组织
9.2 静态存储分配
9.3 动态存储分配
9.4 非局部数据的访问
9.5 符号表

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

块结构语言的静态作用域规则

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

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　　*C*语言的程序块机制
- 如何访问非局部数据?
  - 访问链（静态链）
  - display表（嵌套层次显示表）

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

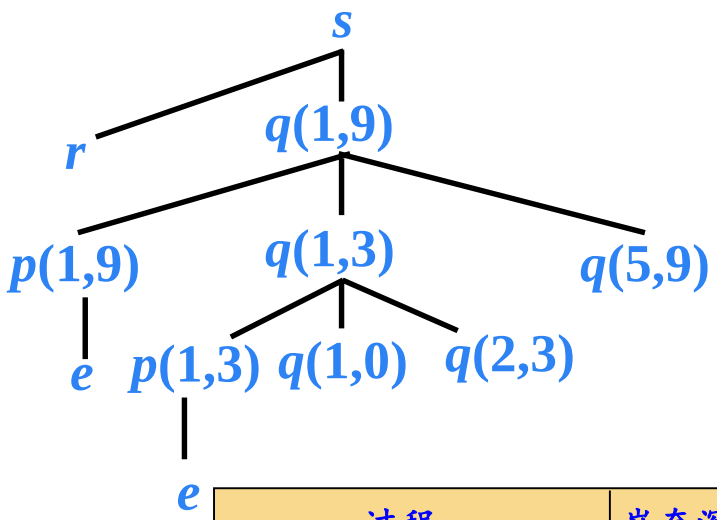
```
program sort ( input, output );
    var a: array[0..10] of integer;
        x: integer;
    procedure readarray;
            var i: integer;
            begin … a … end {readarray} ;
    procedure exchange(i, j:integer);
            begin x=a[i];a[i]=a[j];a[j]=x; end {exchange} ;
    procedure quicksort(m,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

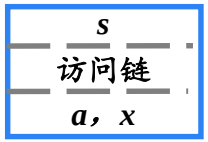
```
program sort ( input, output );
    var a: array[0..10] of integer;
        x: integer;
    procedure readarray;
            var i: integer;
            begin … a … end {readarray} ;
    procedure exchange(i, j:integer);
            begin x=a[i];a[i]=a[j];a[j]=x; end {exchange} ;
    procedure quicksort(m,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



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

不内嵌在任何其它块中的块，设其嵌套深度为1

控制栈

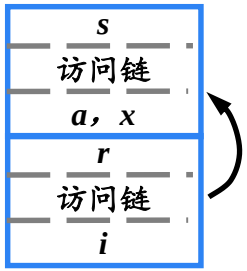


活动树

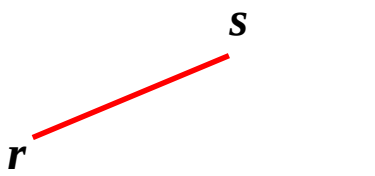
s

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

控制栈

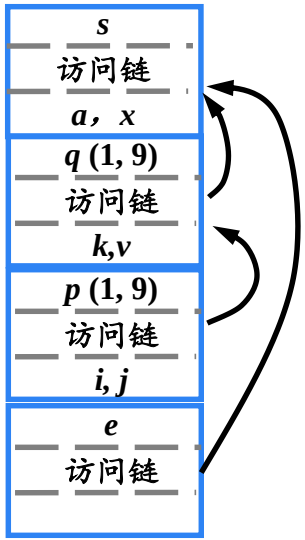


活动树

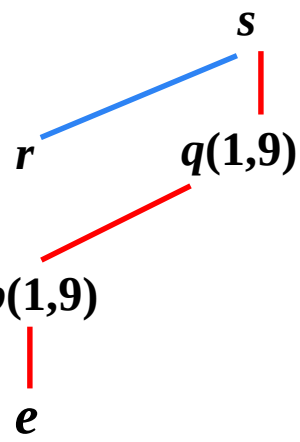


过程	嵌套深度
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

控制栈



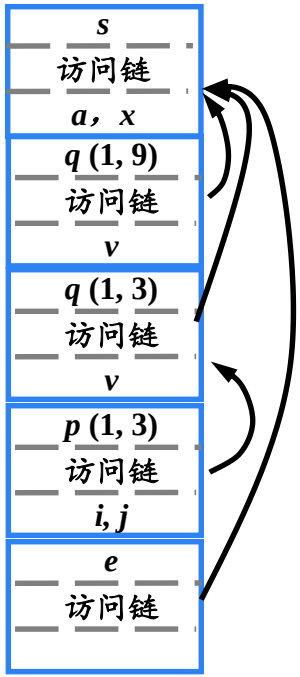
活动树



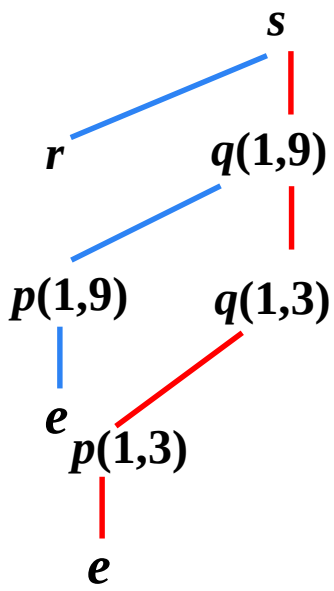
过程	嵌套深度
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

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

控制栈

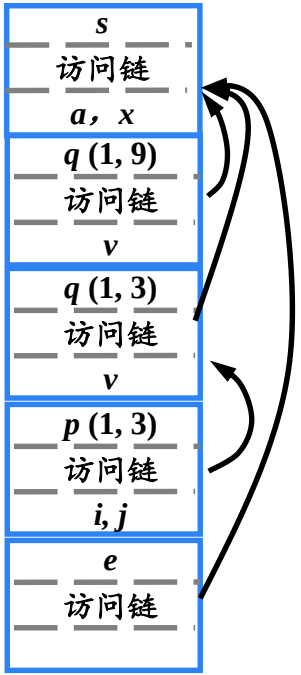


活动树



访问链的建立方法

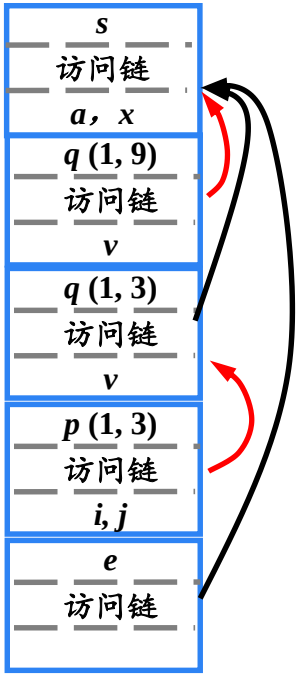
- 建立访问链的代码属于调用代码序列的一部分
- 假设嵌套深度为n_x的过程x调用嵌套深度为n_y的过程y ($x \rightarrow y$)
  - $n_x < n_y$的情况(外层调用内层)
    - y一定是直接在x中定义的 (例如：$s \rightarrow q$, $q \rightarrow p$) ，因此，$n_y = n_x + 1$
    - 在调用序列代码中增加一个步骤：在y的访问链中放置一个指向x的活动记录的指针



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

访问链的建立

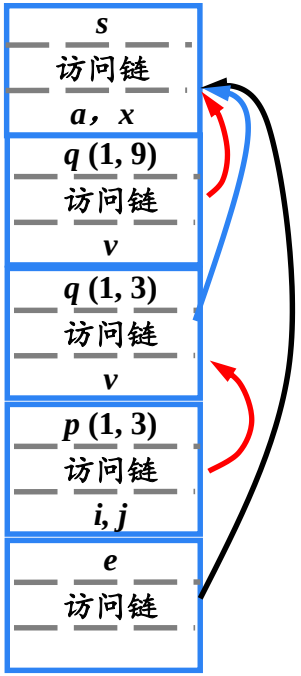
- 建立访问链的代码属于调用代码序列的一部分
- 假设嵌套深度为n_x的过程x调用嵌套深度为n_y的过程y $(x \rightarrow y)$
  - $n_x < n_y$的情况(外层调用内层)
  - $n_x = n_y$的情况(本层调用本层)
    - 递归调用(例如：$q \rightarrow q$)
    - 被调用者的活动记录的访问链与调用者的活动记录的访问链是相同的，可以直接复制



过程	嵌套深度
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代码序列的一部分
- 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： $p \rightarrow e$)
    - 过程x必定嵌套在某个过程z中，而z中直接定义了过程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

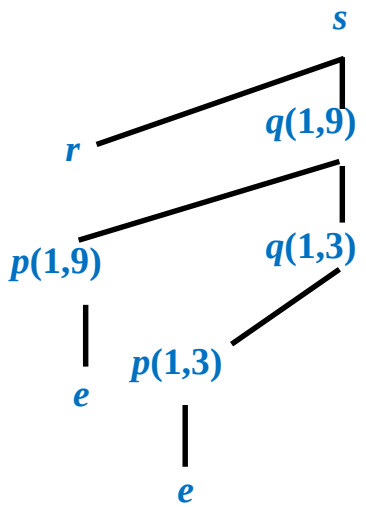
例题

5. 代码为一个快速排序程序的概要，下图是可能在程序的某次执行过程中得到的当前活动树。要求给出：

（1）每个过程的嵌套深度

（2）图示出当前栈中的活动记录，每个活动记录只包含过程名和实参如：*quicksort*(2,3)或简写为*q*(2,3)、访问链和局部变量，如：*i*。并为各个活动建立访问链。

```
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};
```



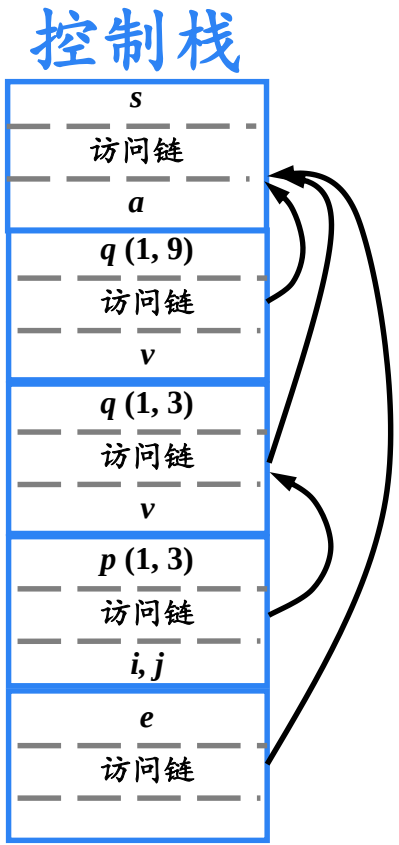
课堂作业

答:

(1) 每个过程的嵌套深度为:

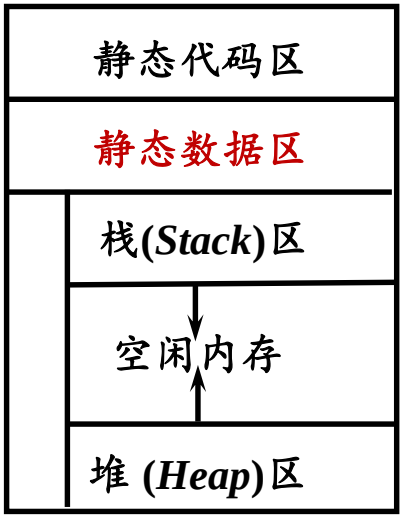
过程	嵌套深度
sort	1
readarray	2
exchange	2
quicksort	2
partition	3

(2)活动记录和访问链为:



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。



提纲

9.1 存储组织
9.2 静态存储分配
9.3 动态存储分配
9.4 非局部数据的访问
9.5 符号表

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

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

- 符号表的作用
  - 进行地址分配，辅助代码生成
  - 一致性检查

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

符号表的组织

- 单个过程符号表的组织
- 多个过程符号表的组织

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

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

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）

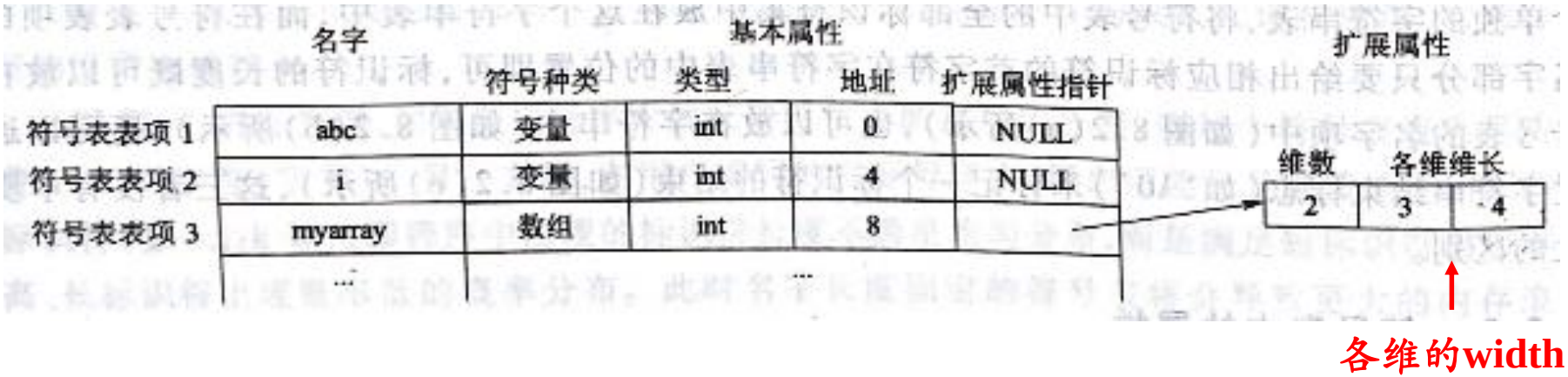
例

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];



多个过程符号表的组织

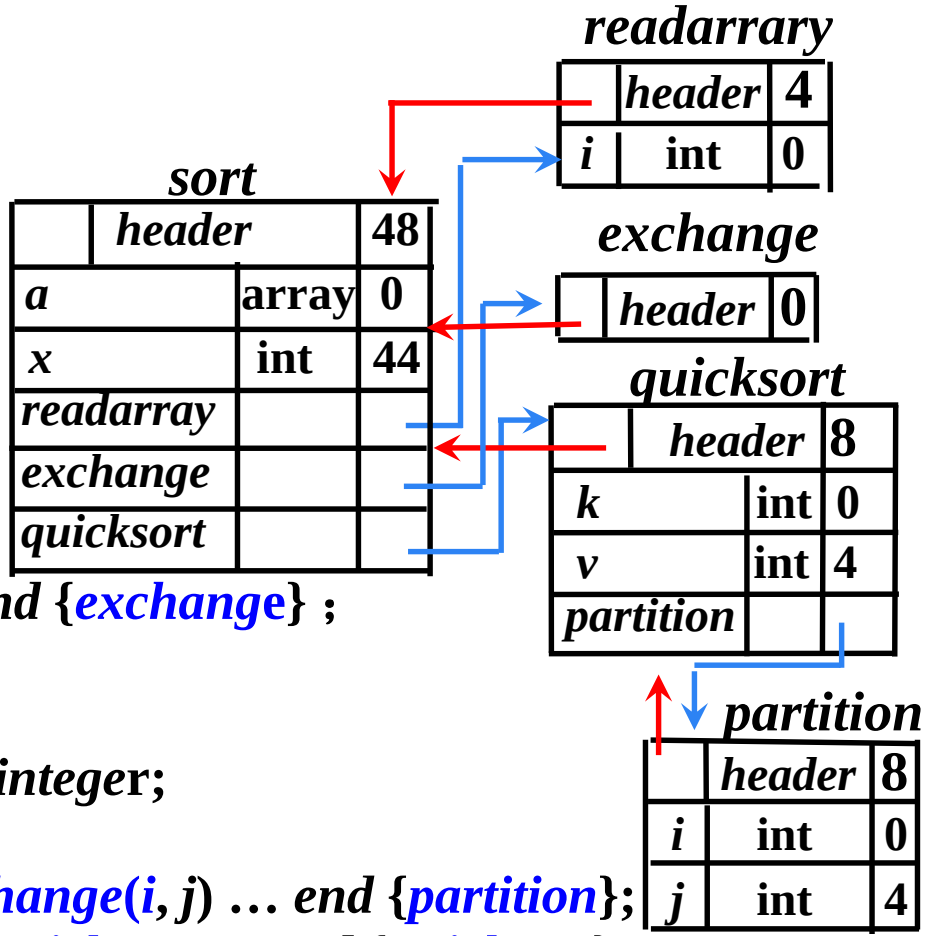
- 需要考虑的问题

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

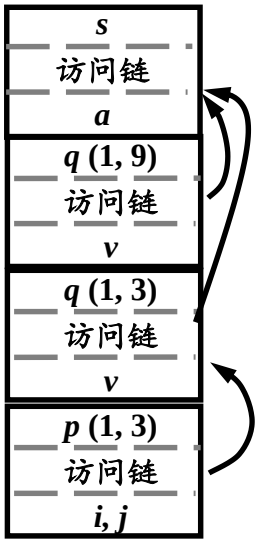
  - 刻画过程之间的嵌套关系（作用域信息）
  - 重名问题
- 常用的组织方式
  - 每个过程建立一个符号表，同时需要建立起这些符号表之间的联系，用来刻画过程之间的嵌套关系

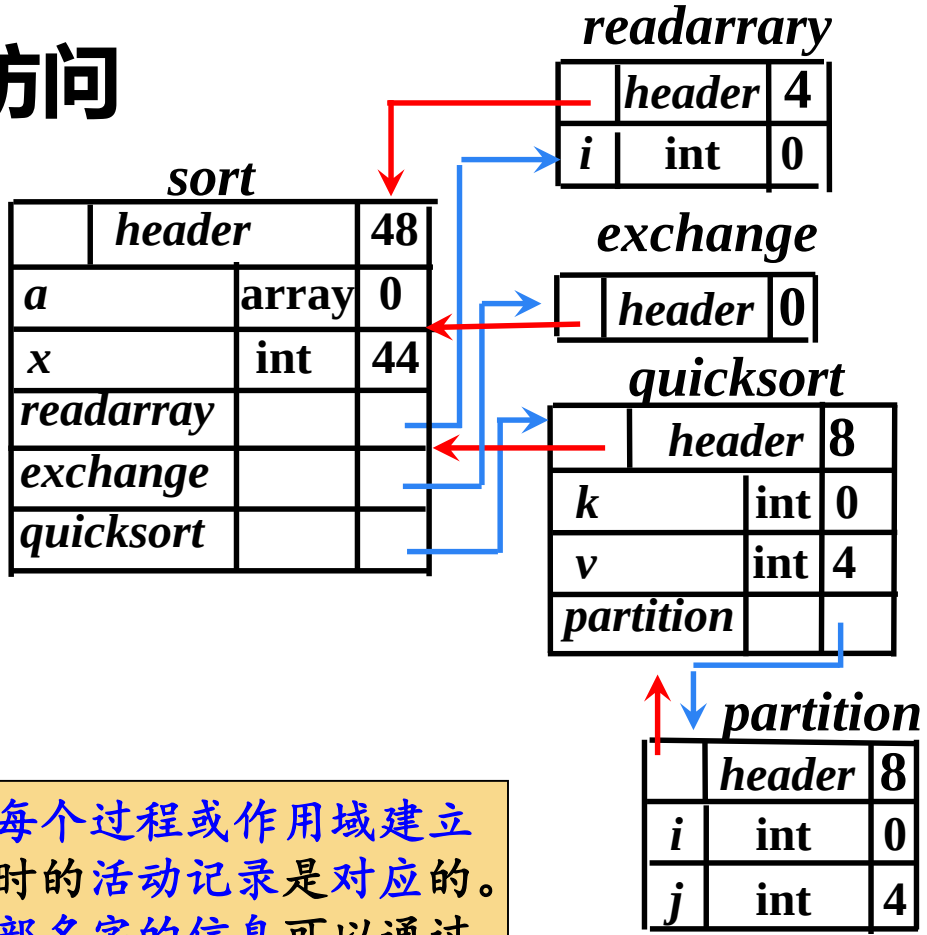
例

```
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};
```



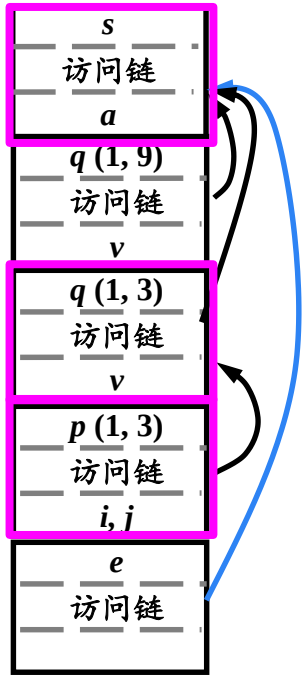
根据符号表进行数据访问

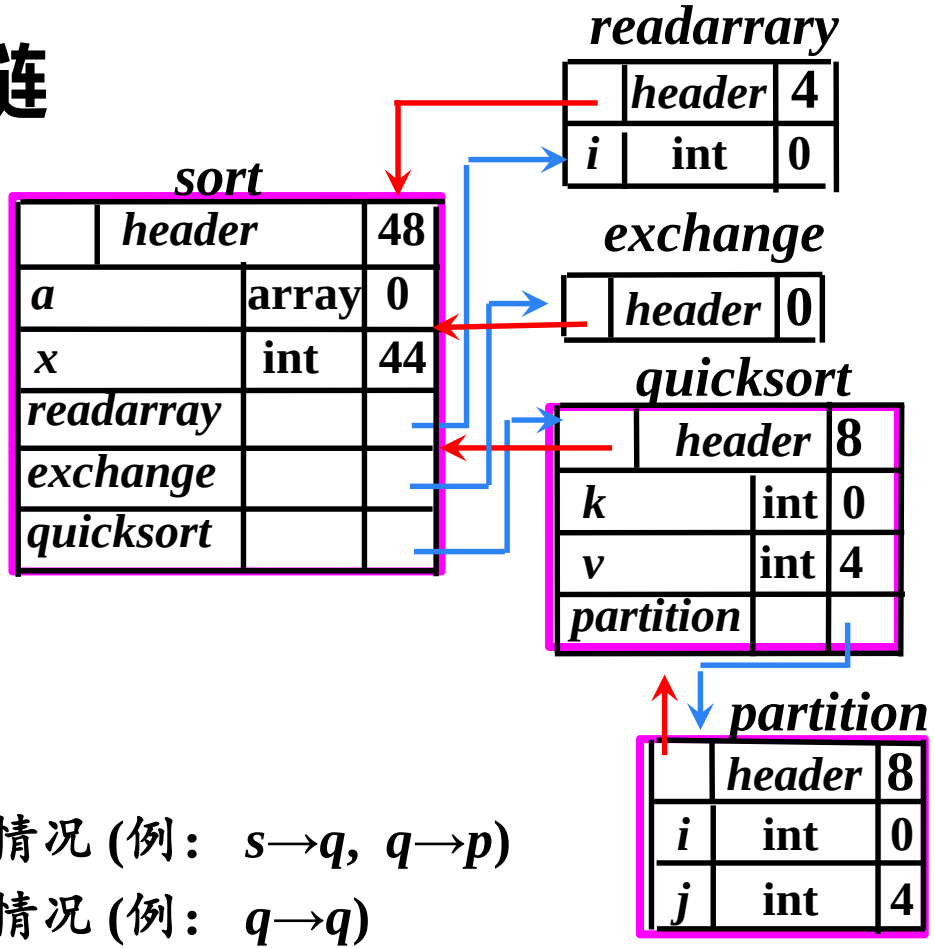




实际上，这种为每个过程或作用域建立的符号表与编译时的活动记录是对应的。一个过程的非局部名字的信息可以通过扫描外围过程的符号表而得到

根据符号表构造访问链





- $x \rightarrow y$
  - $n_x < n_y$的情况 (例： $s \rightarrow q$, $q \rightarrow p$)
  - $n_x = n_y$的情况 (例： $q \rightarrow q$)
  - $n_x > n_y$的情况 (例： $p \rightarrow e$)

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”

例

1.在目标代码生成阶段，符号表用于（ ）

A：目标代码生成　　B：语义检查

C：语法检查　　D：地址分配

2.以下说法正确的是（ ）

A：符号表由词法分析程序建立，由语法分析程序使用

B：符号表的内容在词法分析阶段填入并在以后各个阶段得到使用

C：对一般的程序设计语言而言，其编译程序的符号表应包含哪些内容及何时填入这些信息不能一概而论

D：“运算符与运算对象类型不符”属于语法错误

作业

1.使用访问链分别画出下面Pascal程序执行到

（1）第1次调用r之后的控制栈（运行栈）的内容；

（2）第2次调用r之后的控制栈（运行栈）的内容。

```
program pascal1;
    procedure p;
          var x: integer;
          procedure q;
               procedure r;
               begin
                    x:=2;
                    …
                    if … then p;
               end;{r}
           begin
                r;
           end; {q}
    begin
          q;
    end; {p}
begin
    p;
end. { pascal1 }
```