第9章　目标程序运行时的存储组织

概述

- 在有操作系统的情况下，目标程序将在自己的逻辑地址空间内存储和运行。
- 编译程序在生成目标代码时应该明确程序的各类对象在逻辑地址空间内是如何存储的，以及目标代码运行时是如何使用和支配自己的逻辑地址空间的。

9.1 运行时存储组织概述

9.1.1 运行时存储组织的作用与任务

- 代码生成前如何安排目标机存储资源的使用
- 几个重要问题
  - 数据表示　目标机中如何表示源语言中各类数据对象
  - 表达式计算　如何组织表达式的计算
  - 存储分配策略　如何为不同作用域或不同生命周期的数据对象分配存储
  - 过程实现　如何实现过程/函数调用以及参数传递

1. 数据表示

- 源程序中数据对象在内存或寄存器中的表示形式
  - **源程序中数据对象的属性**

    名字（*name*），类型（*type*），值（*value*），复合数据对象（*component*），……
  - **数据对象在内存或寄存器中的表示形式**

    位、字节、字、字节序列、……
  - **有些机器要求数据存放时要按某种方式对齐**

    如：要求数据存放的起始地址为能够被4整除

数据表示举例

基本类型数据

char 数据　1 *byte*　　*integer* 数据　4 *bytes*

float 数据　4 *bytes*　　*boolean* 数据　1 *byte*

指针　4 *bytes*

数组　一块连续的存储区（按行/列存放）

结构/记录　所有域（*field*）存放在一块连续的存储区

对象　实例变量像结构的域一样存放在一块连续的存储区，操作例程（方法、成员函数）存放在其所属类的代码区

2. 表达式的计算

- 在何处进行计算

  - **在栈区计算**

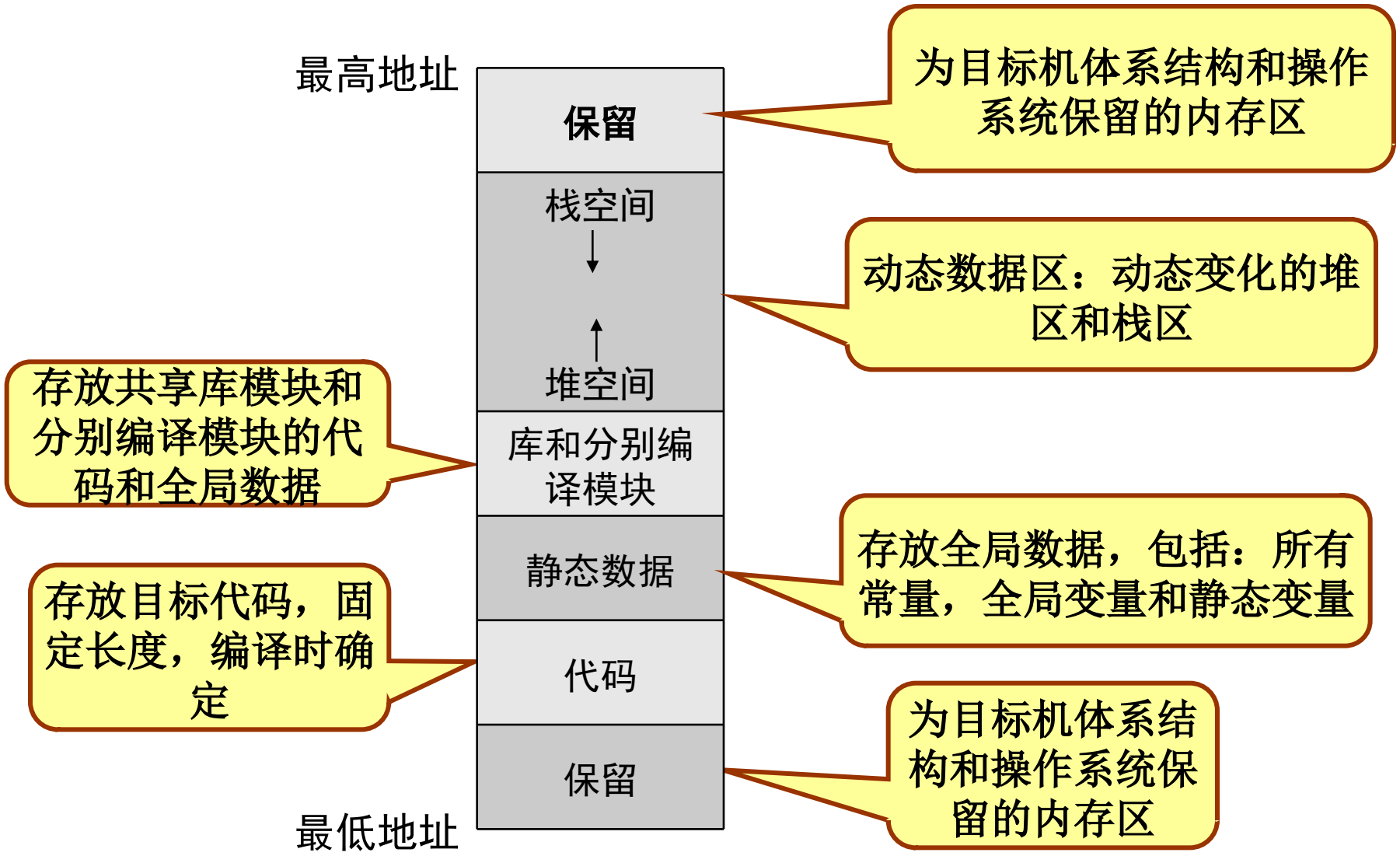
    运算数/中间结果存放于当前活动记录或通用寄存器中

  - **在运算数栈计算**

    某些目标机采用专门的运算数栈用于表达式计算

    对于普通表达式（无函数调用），一般可以估算出能否在运算数栈上进行
    使用了递归函数的表达式的计算通常在栈区

9.1.2 程序运行时存储空间的布局



Windows32下对数据空间的划分

```
#include <stdio.h>
#include <stdlib.h>
int global1;                                    //未初始化全局变量
int global2=200;                                //初始化全局变量
int main(void)
{
    int a, b, i;                                //a, b, i, p局部变量
    int *p;                                     //指向整型的指针变量
    scanf("%d%d%d", &a, &b, &i);                //读入a, b, i
    scanf("%d%d", &global1, &global2);                  //读入global1, global2
    p=(int *)malloc(20*sizeof(int));                    //分配80个字节空间给p
    for (i=0; i<20; i++)
            *(p+i)=i;                           //给p数组赋值
    free(p);                                    //释放p
    return 0;
}
```

- **Windows32下，每个段由多个页组成，每个页的大小为4K。如果某页32位线性地址为：XXXXX000~XXXXXFFF，则该页记为第XXXXX页。每个程序共有4G空间，00000-FFFFF页**
- **Windows32的段主要有如下几种：**
  - **代码段：存放指令代码　(00401页)**
  - **只读数据段：只读的常量(字符常量和程序运行中的错误信息 （00422页, 如“%d%d”）**
  - **可读写的经初始化的数据段：全局变量和静态变量 (00424页, global1)**
  - **可读写的未经初始化的数据段：全局变量/静态变量（00425页global2）**
  - **栈段(Stack)：局部变量和返回地址等数据 (0012f页，a, b, i等)**
  - **堆段(Heap)：动态申请/释放的存储段 (00431页，p所指向的内存区域)**

9.1.3 存储分配策略

- 静态分配
  - 在编译期间为数据对象分配存储
- 动态分配
  - 栈式分配

    将数据对象的运行时存储按照栈的方式来管理
  - 堆式分配

    从数据段的堆空间分配和释放数据对象的运行时存储

9.1.3.1 静态存储分配

- 在编译期间就可确定数据对象的大小
  - 不宜处理递归过程或函数
- 某些语言中所有存储都是静态分配
  - 如早期的FORTRAN语言，COBOL语言
- 多数语言只有部分存储进行静态分配
  - 可静态分配的数据对象如大小固定且在程序执行期间可全程访问的全局变量，以及程序中的常量（*literals*）
  - 如 C++ 中的 static 变量

9.1.3.2 栈式存储分配

- 用于有效实现可动态嵌套的程序结构
  - 如实现过程/函数，块层次结构
- 可以实现递归过程/函数
  - 比较：静态分配不宜实现递归过程/函数
- 运行栈中的数据单元是活动记录**AR**（*activation record*）

9.1.3.3 堆式存储分配

- 从堆空间为数据对象分配/释放存储
  - 灵活 数据对象的存储分配和释放不限时间和次序
- 显式的分配或释放（*explicit allocation / dealocation*）
  - 程序员负责应用程序的（堆）存储空间管理（借助于编译器与（或）运行时系统所提供的默认存储管理机制）
- 隐式的分配或释放（*implicit allocation / dealocation*）
  - （堆）存储空间的分配或释放不需要程序员负责，由编译器与（或）运行时系统自动完成

- 某些语言有显式的堆空间分配和释放命令
  - 如：Pascal 中的 ***new*** , ***deposit***
    C++ 中的 ***new*** , ***delete***
  - 比较：C 语言没有堆空间管理机制，***malloc()*** 和 ***free()*** 是标准库中的函数
- 某些语言支持隐式的堆空间释放
  - 采用垃圾回收（*garbage collection*）机制
  - 如：Java 程序员不需要考虑对象的析构

堆式存储分配释放的三种方案

1. 不释放堆空间的方法

- 只分配空间，不释放空间，空间耗尽时停止
- 适合于堆数据对象多数为一旦分配，永久使用的情形
- 在虚存很大，而无用数据对象不致带来很大零乱的情形也可采用

堆式存储分配释放的三种方案

2. 显式释放堆空间的方法

- 用户负责清空无用的数据空间（通过执行释放命令）
- 堆管理程序只维护可供分配命令使用的空闲空间
- 问题：可能导致灾难性的*指针悬挂*错误

例：**Pascal** 代码片断

```
var p,q: ^real;
…
new(p);
q:=p;
dispose(p);
q^:=1.0;
```

C++ 代码片断

```
float  * p,*q;
…
p=new  float;
q=p;
delete p;
*q:=1.0;
```

堆式存储分配释放的三种方案

3. 隐式释放堆空间的方法

- 主要技术： 垃圾回收（*garbage collection*）机制
- 典型的语言环境：**Java，QT**

堆式空间的管理

- 分配算法　面对多个可用的存储块，选择哪一个

  如：最佳适应算法（选择浪费最少的存储块）

  　　最先适应算法（选择最先找到的足够大的存储块）

  　　循环最先适应算法（起始点不同的最先适应算法）

- 碎片整理算法　压缩合并小的存储块，使其更可用

9.2 活动记录

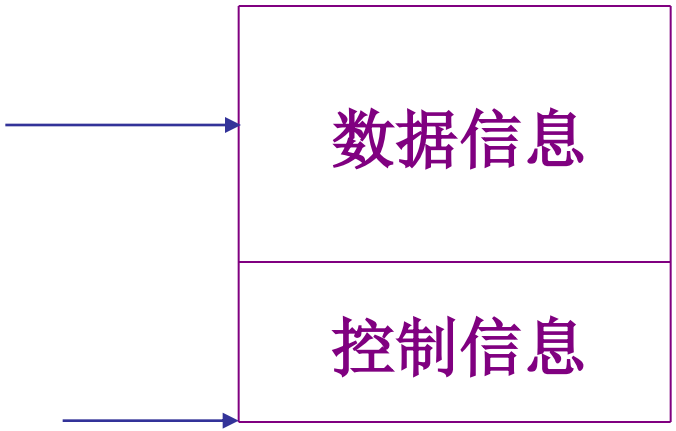
➢ 过程活动记录

- 函数/过程调用或返回时，在运行栈上创建或从运行栈上消去的栈帧（*frame*）

包含局部变量，函数实参，临时值（用于表达式计算的中间单元）等数据信息以及必要的控制信息

某个数据对象的地址=
活动记录起始地址
+ 偏移地址（*offset*）

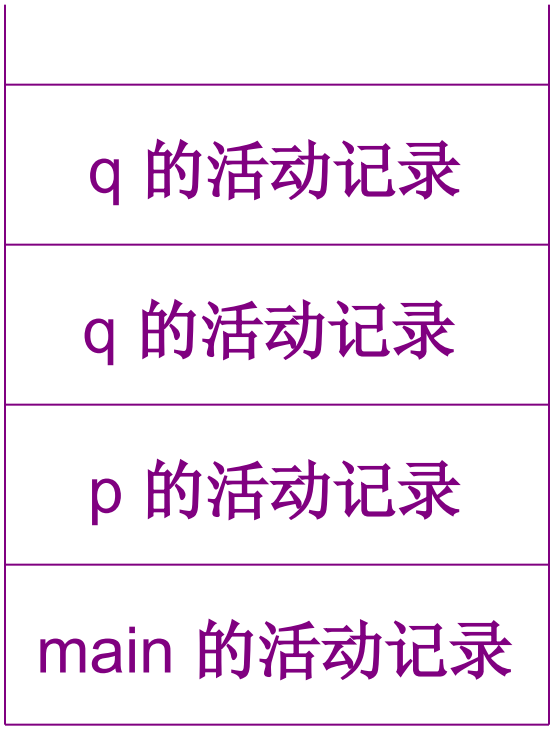
活动记录起始地址



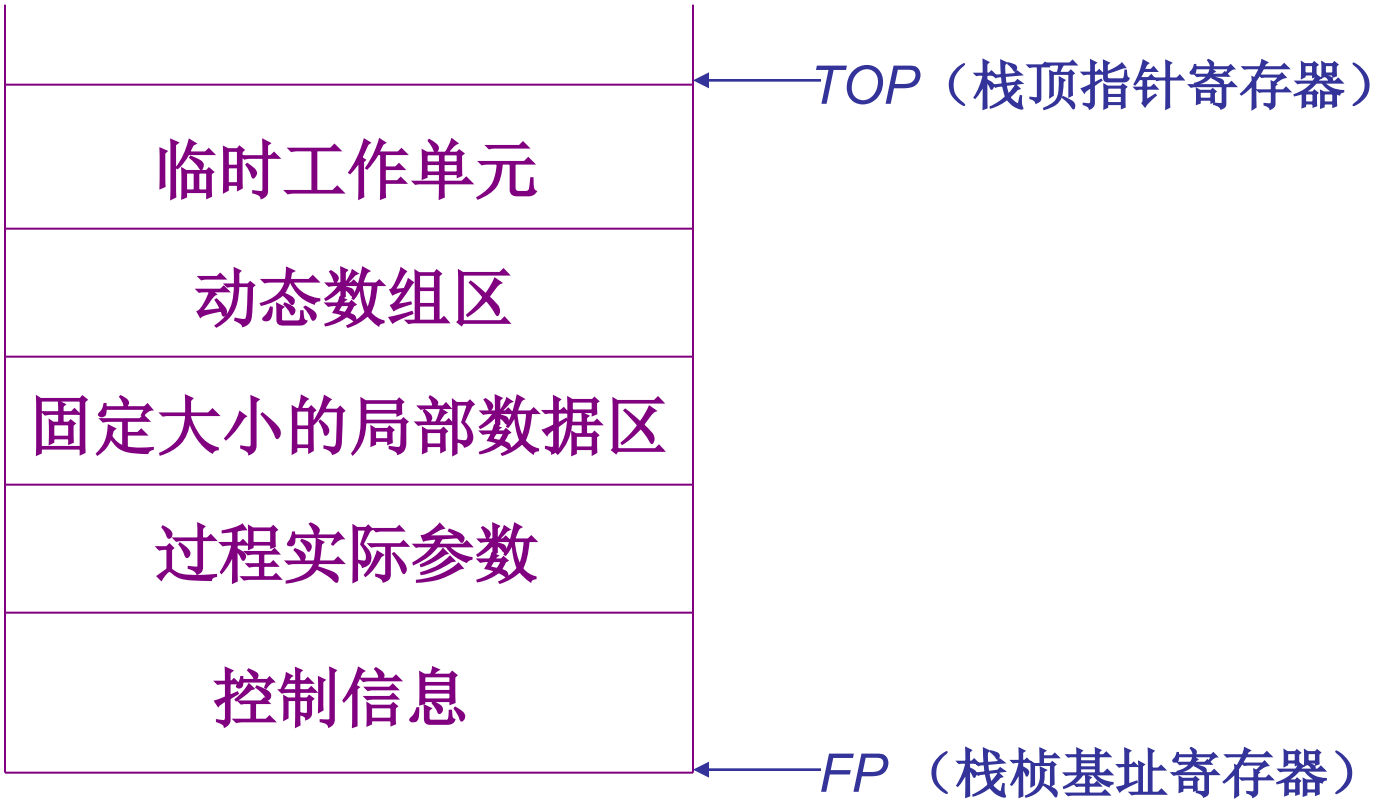
过程活动记录的栈式分配举例

```
void p( ) {
   …
   q( );
}

void q( ) {
   …
   q( );
}

int main {
   p( );
}
```

函数 q 被第二次调用时运行栈上活动记录分配情况



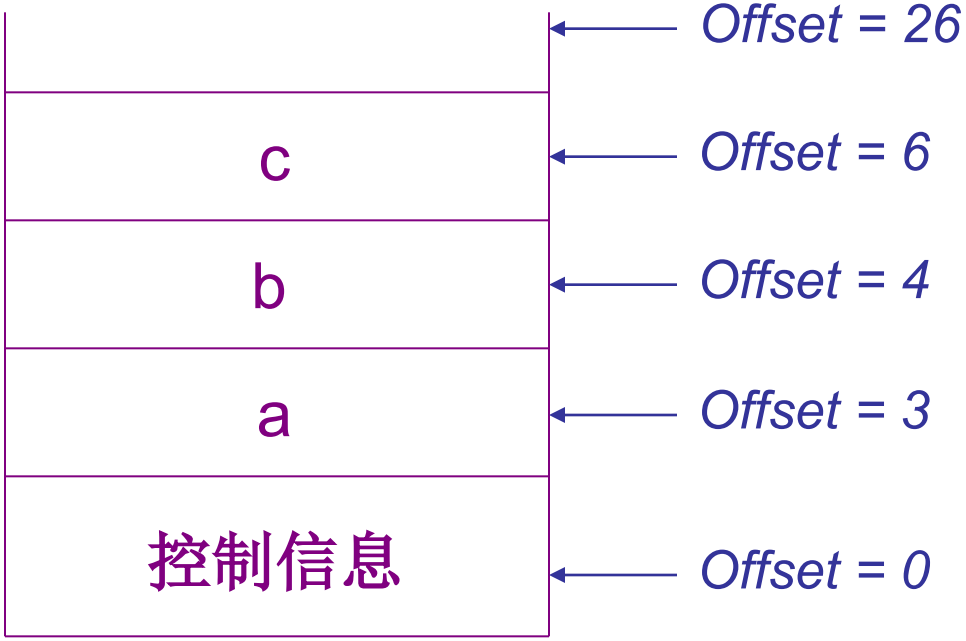
典型的过程活动记录结构



过程活动记录举例

```
void p( int a)  {
   float b;
   float c[10];
   b=c[a];
}
```

函数 p 的活动记录



过程活动记录举例

```
static int N;

void p( int a)  {
   float b;
   float c[10];
   float d[N];
   float e;
   …

}
```

/*d为动态数组*/

9.2 嵌套过程定义中非局部量的访问

Pascal、ML等程序设计语言允许嵌套的过程/函数定义，需要解决的一个重要问题是非局部量的访问。

- 主要问题

  解决对非局部量的引用（存取）

- 解决方案

  采用 Display 表

  为活动记录增加静态链域

```
(1)  program sort(input, output)
(2)      var a:array [0..10] of integer;
(3)      x: integer;
(4)      procedure readarray;
(5)          var i:integer;
(6)      begin …a… end {readarray};
(7)      procedure exchange(i,j: integer)
(8)      begin
(9)          x:=a[i]; a[i]:=a[j]; a[j]:=x;
(10)     end{exchange};
(11)     procedure quicksort(m,n:integer)
(12)         var k, v: integer;
(13)         function partition(y,z:integer):integer;
(14)             var i,j:integer;
(15)         begin …a…
(16)             …v…
(17)             …exchange(i,j); …
(18)         end{partition};
(19)     begin…end{quicksort};
(20) begin … end {sort}.
```

过程定义的嵌套情况：

sort

- readarray
- exchange
- quicksort
  - partition

readarray, exchange, quicksort中引用的a都不是局部变量，而是sort过程的局部变量。

9.2.2.1 Display表

Display 表记录各嵌套层当前过程的活动记录在运行栈上的起始位置（基地址）

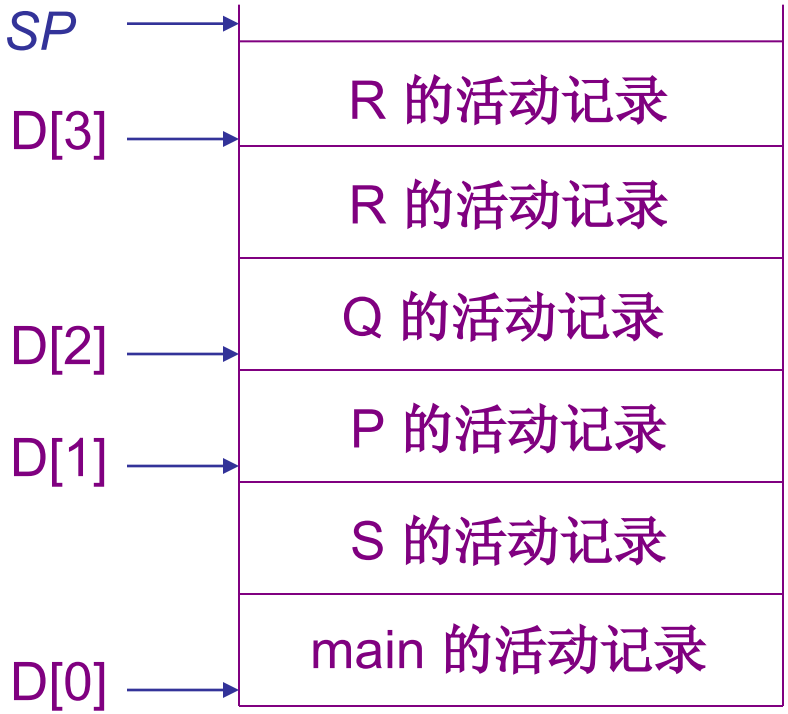
若当前激活过程的层次为K（主程序的层次设为*0*），则对应的 Display 表含有 K+1 个单元，依次存放着现行层，直接外层…直至最外层的每一过程的最新活动记录的基地址

嵌套作用域规则确保每一时刻Display 表内容的唯一性

Display 表的大小（即最多嵌套的层数）取决于实现

Display 表方案举例

过程 R 被第二次激活后运行栈和
Display 寄存器 D[i] 的情况




```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
      begin
        … R; …
      end;   /*R*/
    begin
      … R; …
    end;   /*Q*/
  begin
    … Q; …
  end;   /*P*/
procedure S;
  begin
    … P; …
  end;   /*S*/
begin
  …  S; …
end.   /*main*/
```

调用层次：Main→S→P→Q→R→R

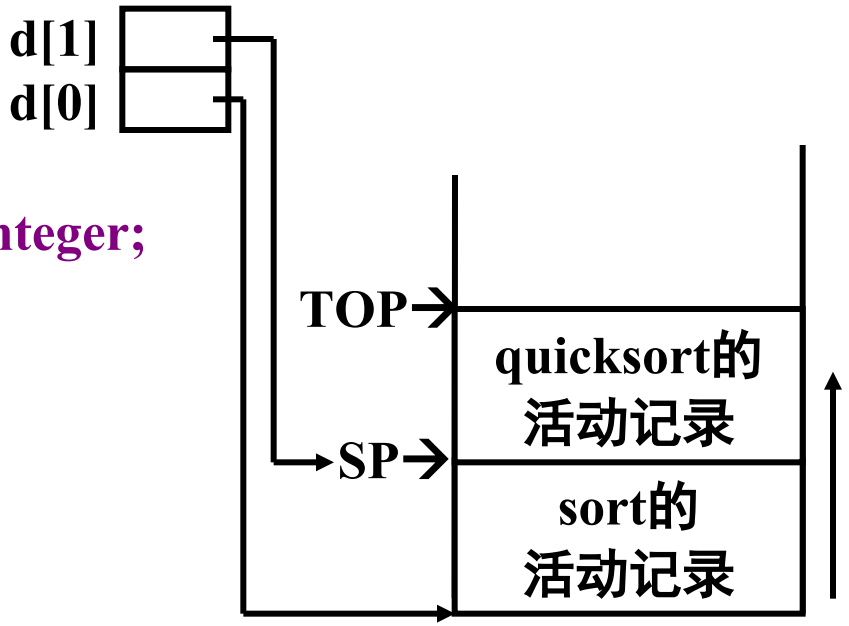
```
(1)  program sort(input, output)
(2)      var a:array [0..10] of integer;
(3)      x: integer;
(4)      procedure readarray;
(5)          var i:integer;
(6)      begin …a… end {readarray};
(7)      procedure exchange(i,j: integer)
(8)      begin
(9)          x:=a[i]; a[i]:=a[j]; a[j]:=x;
(10)     end{exchange};
(11)     procedure quicksort(m,n:integer)
(12)         var k, v: integer;
(13)         function partition(y,z:integer):integer;
(14)             var i,j:integer;
(15)         begin …a…
(16)             …v…
(17)             …exchange(i,j); …
(18)         end{partition};
(19)     begin…end{quicksort};
(20) begin … end {sort}.
```

调用情况：

sort→quicksort

…

quicksort的display表



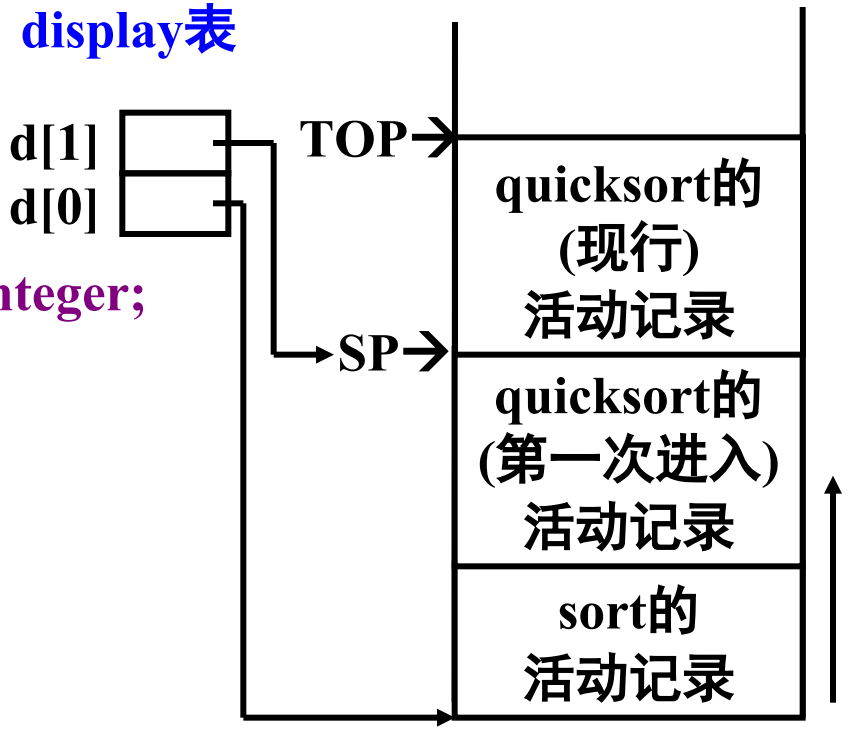
```
(1)  program sort(input, output)
(2)      var a:array [0..10] of integer;
(3)      x: integer;
(4)      procedure readarray;
(5)          var i:integer;
(6)      begin …a… end {readarray};
(7)      procedure exchange(i,j: integer)
(8)      begin
(9)          x:=a[i]; a[i]:=a[j]; a[j]:=x;
(10)     end{exchange};
(11)     procedure quicksort(m,n:integer)
(12)         var k, v: integer;
(13)         function partition(y,z:integer):integer;
(14)             var i,j:integer;
(15)         begin …a…
(16)             …v…
(17)             …exchange(i,j); …
(18)         end{partition};
(19)     begin…end{quicksort};
(20) begin … end {sort}.
```

调用情况：

sort→quicksort

→quicksort …

quicksort的（现行）display表



```
(1)  program sort(input, output)
(2)      var a:array [0..10] of integer;
(3)      x: integer;
(4)      procedure readarray;
(5)          var i:integer;
(6)      begin …a… end {readarray};
(7)      procedure exchange(i,j: integer)
(8)      begin
(9)          x:=a[i]; a[i]:=a[j]; a[j]:=x;
(10)     end{exchange};
(11)     procedure quicksort(m,n:integer)
(12)         var k, v: integer;
(13)         function partition(y,z:integer):integer;
(14)             var i,j:integer;
(15)         begin …a…
(16)             …v…
(17)             …exchange(i,j); …
(18)         end{partition};
(19)     begin…end{quicksort};
(20) begin … end {sort}.
```

调用情况：

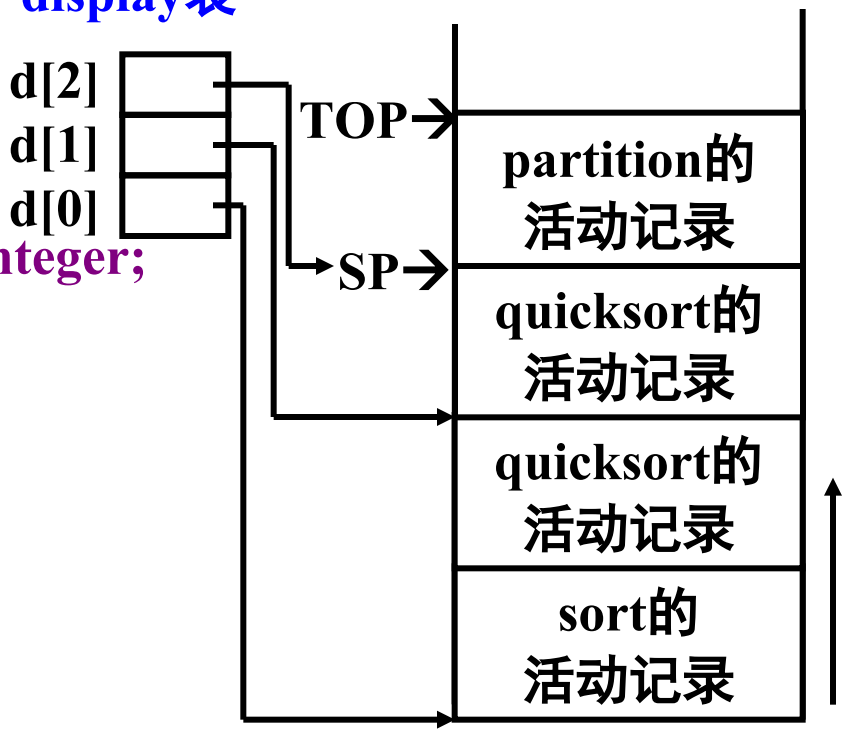
sort→quicksort

→quicksort

→partition …

partition的

display表



```
(1)  program sort(input, output)
(2)      var a:array [0..10] of integer;
(3)      x: integer;
(4)      procedure readarray;
(5)          var i:integer;
(6)      begin …a… end {readarray};
(7)      procedure exchange(i,j: integer)
(8)      begin
(9)          x:=a[i]; a[i]:=a[j]; a[j]:=x;
(10)     end{exchange};
(11)     procedure quicksort(m,n:integer)
(12)         var k, v: integer;
(13)         function partition(y,z:integer):integer;
(14)             var i,j:integer;
(15)         begin …a…
(16)             …v…
(17)             …exchange(i,j); …
(18)         end{partition};
(19)     begin…end{quicksort};
(20) begin … end {sort}.
```

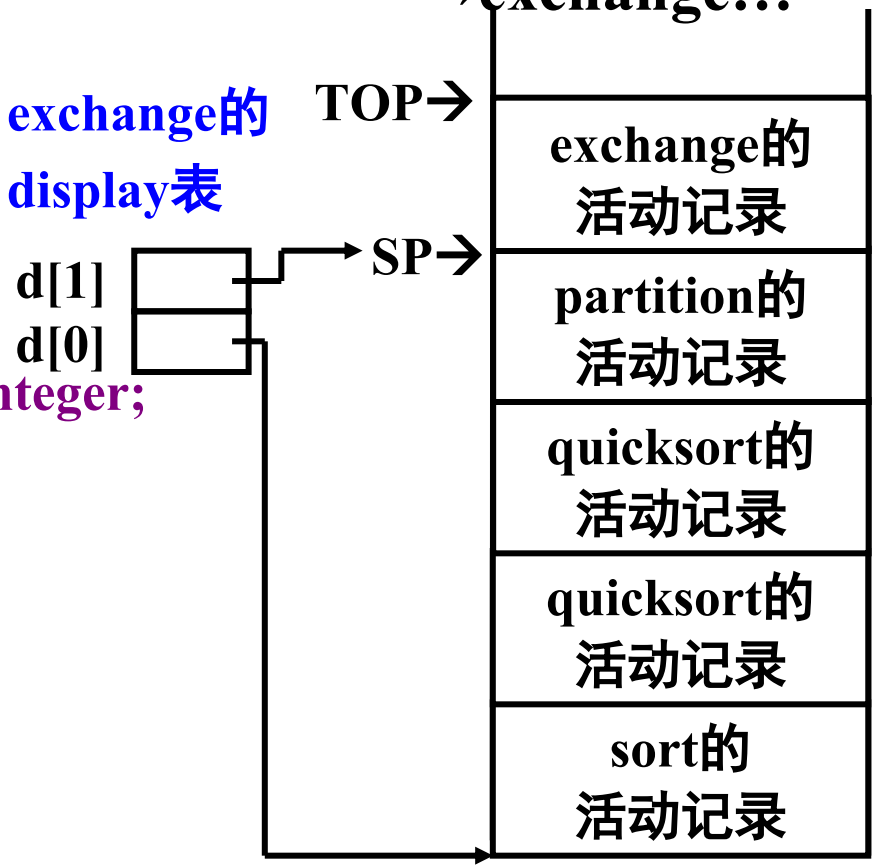
调用情况：

sort→quicksort

→quicksort

→partition

→exchange…



Display 表的维护

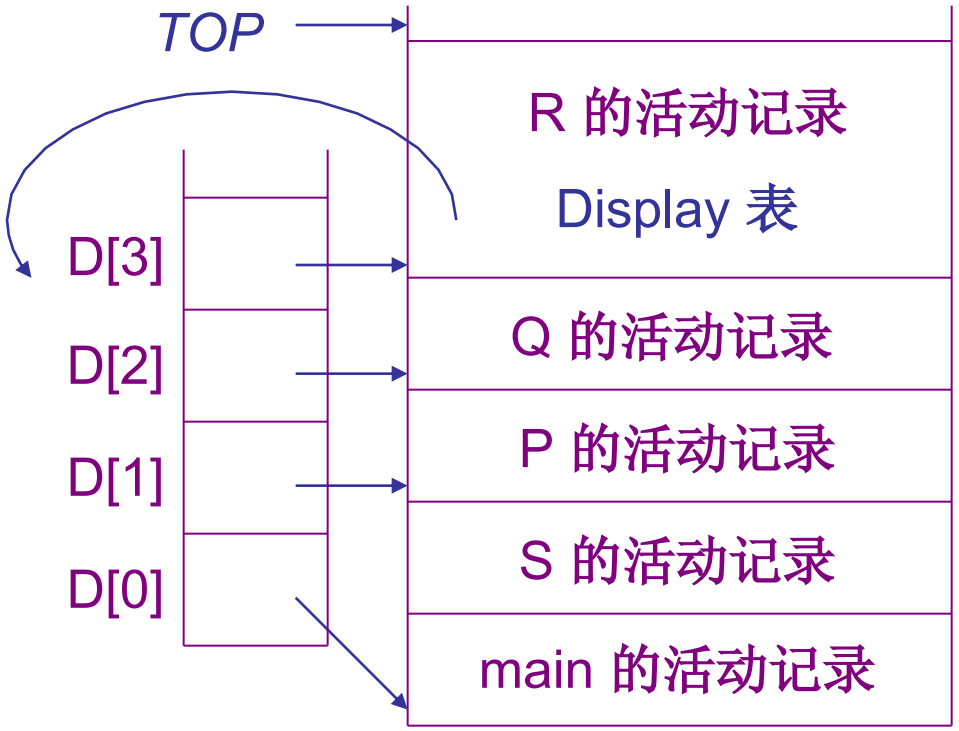
方法一　极端的方法是把整个 Display 表存入活动记录（若过程为第 *n* 层，则需要保存 D[0] ~D[*n*] ）

一个过程（处于第 *n* 层）被调用时，从调用过程的 Display 表中自下向上抄录 *n* 个 TOP 值，再加上本层的 TOP 值

方法二　只在活动记录保存一个的 Display 表项，在静态存储区或专用寄存器中维护全局 Display 表

Display 表的维护举例

活动记录中保存完整的Display 表



调用层次：Main→S→P→Q→R

```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
      begin
        … R; …
      end;  /*R*/
    begin
      … R; …
    end;  /*Q*/
  begin
    … Q; …
  end;  /*P*/
procedure S;
  begin
    … P; …
  end;  /*S*/
begin
  …  S; …
end.  /*main*/
```

Display 表的维护举例

只保存一个Display 表项

calls …	P	Q	R	P’	Q’	R’
D[3]	_	_	R	R	R	R’
D[2]	_	Q	Q	Q	Q’	Q’
D[1]	P	P	P	P’	P’	P’
D[0]	Main	Main	Main	Main	Main	Main
saved …	S	_	_	P	Q	R

调用层次：Main→S→P→Q→R→P’→Q’→R’

```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
      begin
        … P; …
      end;  /*R*/
    begin
      … R; …
    end;  /*Q*/
  begin
    … Q; …
  end;  /*P*/
procedure S;
  begin
    … P; …
  end;  /*S*/
begin
  …  S; …
end.  /*main*/
```

9.2.2.2 采用静态链

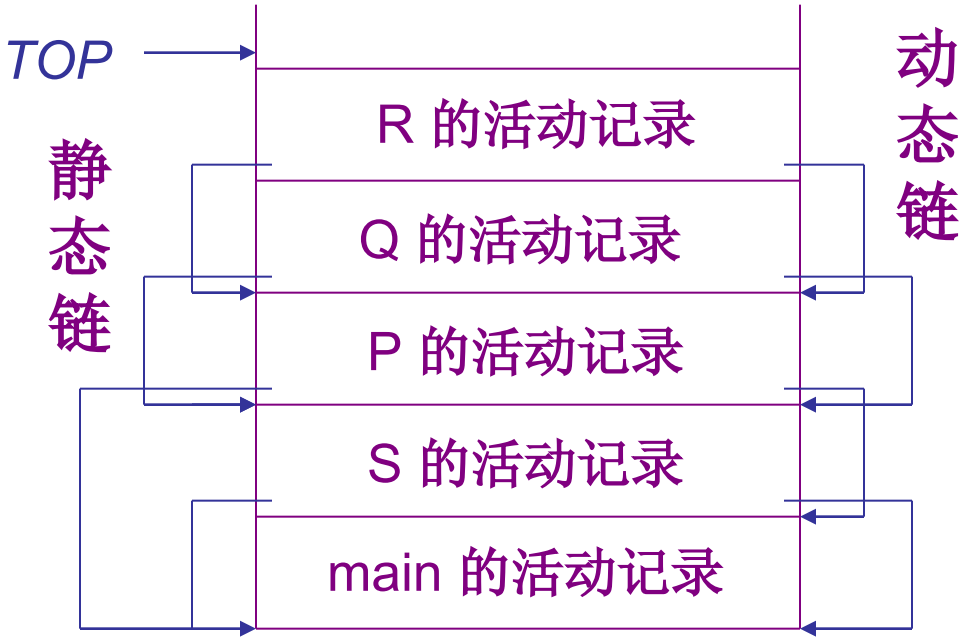
Display 表的方法要用到多个存储单元或多个寄存器，有时并不情愿这样做，一种可选的方法是采用静态链

所有活动记录都增加一个静态链（也称为访问链）的域，指向定义该过程的直接外过程（或主程序）运行时最新的活动记录

在过程返回时当前 AR 要被撤销，为回卷（*unwind*）到调用过程的AR（恢复FP）需要用到动态链（也称为控制链）域，指向调用该过程的活动记录的基址

采用静态链的方法举例

过程 R 被第一次激活后运行栈的情况



调用层次：Main→S→P→Q→R

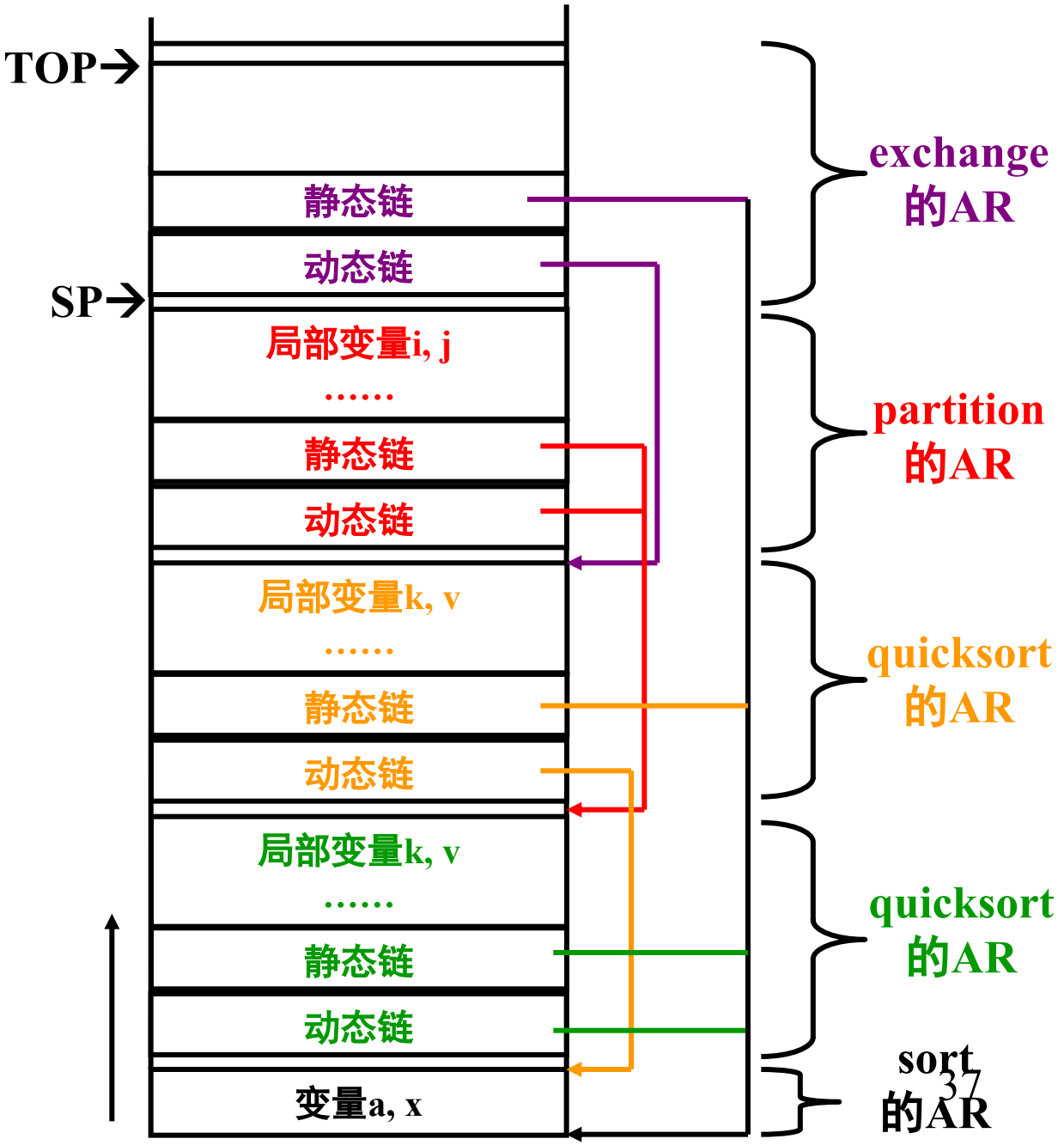
```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
      begin
        … R; …
      end;  /*R*/
    begin
      … R; …
    end;  /*Q*/
  begin
    … Q; …
  end;  /*P*/
procedure S;
  begin
    … P; …
  end;  /*S*/
begin
  …  S; …
end.  /*main*/
```

存取链指向包含该过程的直接外层过程的最新活动记录的起始位置。

如果该程序的某执行顺序为：

sort→quicksort
→quicksort
→partition
→exchange

过程定义的嵌套情况：
sort
readarray
exchange
quicksort
partition



9.2.3 嵌套程序块的非局部量访问

- 一些语言（如 C 语言）支持嵌套的块，在这些块的内部也允许声明局部变量，同样要解决依嵌套层次规则进行非局部量使用（访问）的问题

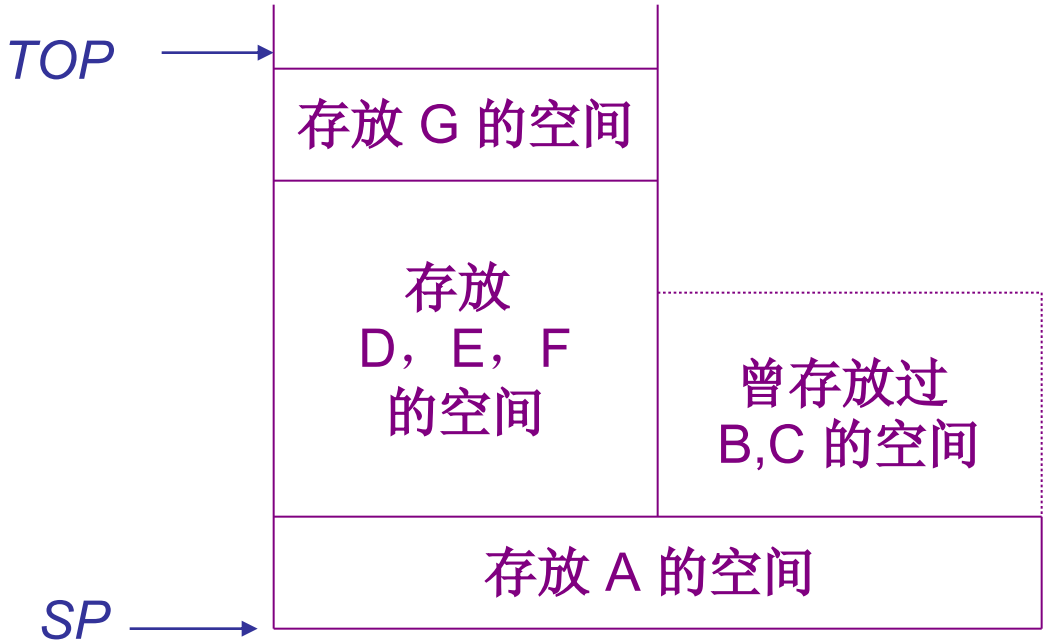
  方法一　将每个块看作为内嵌的无参过程，为它创建一个新的活动记录，称为块级活动记录

  该方法代价很高

  方法二　由于每个块中变量的相对位置在编译时就能确定下来，因此可以不创建块级活动记录，仅需要过程级的活动记录就可解决问题（见下例）

采用过程级活动记录的方法举例

运行至/*here*/时p的活动记录形如：




```
int p（）
{
    int A;
    …
    {
        int B,C;
        …
    }
    {
        int D,E,F;
        …
        {
            int G;
            …  /*here*/
        }
    }
}
```

9.2.4 动态作用域规则和静态作用域规则

- 静态作用域规则：通过观察程序本身就可以确定一个声明的作用域
- 动态作用域规则：只有在程序执行时才能确定程序某处所使用的名字的声明位置。

```
var  r:real
procedure    show;
    begin   write(r:5:3)  end;
procedure small;
    var   r:real;
    begin   r:=0.125;  show  end;
begin
     r:=0.25;
    show; small; writeln;
    show; small; writeln;
end.
```

lexical scope

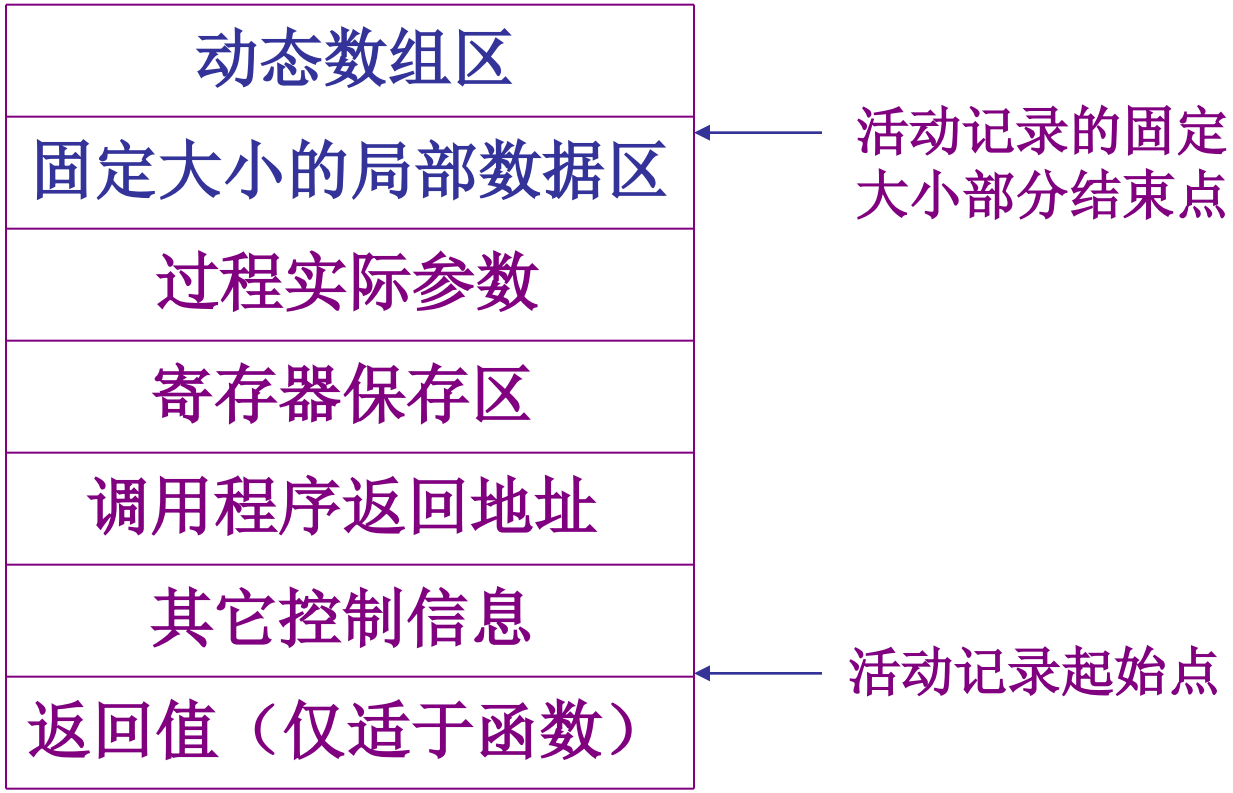
0.250 0.250
0.250 0.250

dynamic scope

0.250 0.125
0.250 0.125

9.3 过程调用

- 活动记录中与过程/函数调用相关的信息
  - 典型的活动记录形式举例



调用起始阶段

- 调用代码序列，由调用过程和被调用过程分担执行，主要工作有：
  a）参数传递
  b）为被调用过程的活动记录分配栈上的存储空间
  c）保存旧栈帧基址
  d）保存调用过程的返回地址
  e）保存其他控制信息
  f）保存寄存器信息
  g）建立新栈帧基址
  h）建立新栈顶
  i）转移控制，启动被调用程序的执行

调用收尾阶段

- 返回代码序列，由被调用过程执行，主要工作有：
  a）如果被调用过程是函数，需要返回一个值。函数返回值可以放入专门的寄存器，也可以入栈。
  b）恢复所有被调用过程保存的寄存器
  c）弹出被调用过程的栈帧，恢复旧栈帧
  d）将控制返回给调用过程
  最后，由调用过程保存的寄存器由调用过程负责恢复

- 最常见的参数传递方式
  - 传值　**call-by-value**

    传递的是实际参数的右值（*r-value*）
  - 传地址　**call-by-reference（-address, -location）**

    传递的是实际参数的左值（*l-value*）

– 注　表达式的左值代表存储该表达式值的地址
　　表达式的右值代表该表达式的值

传值举例

调用swap(a,b) 过程将不会影响a和b的值，其结果等价于执行下列语句序列：

```
x :=a;
y :=b;
temp :=x;
x :=y;
y :=temp
```

```
procedure  swap(x,y:integer);
    var  temp:integer;
    begin
        temp:=x;
        x:=y;
        y:=temp
    end;
```

实现传值的方法

形式参数当作过程的局部变量处理，即在被调过程的活动记录中开辟了形参的存储空间，这些存储位置用以存放实参

调用过程计算实参的值，将其放于对应的存储空间

被调用过程执行时，就像使用局部变量一样使用这些形式单元

传地址举例

调用swap(a,b) 过程将交换 a 和 b 的值

```
procedure  swap(var  x,y:integer);
        var  temp:integer;
        begin
                temp:=x;
                x:=y;
                y:=temp
        end;
```

实现传地址的方法

- 把实在参数的地址传递给相应的形参，即调用过程把一个指向实参的存储地址的指针传递给被调用过程相应的形参；
- 若实在参数是一个名字，或具有左值的表达式，则传递左值若实在参数是无左值的表达式，则计算该表达式的值，放入一存储单元，传此存储单元地址