

第7章 运行环境



LI Wensheng, SCS, BUPT

知识点：活动记录、控制栈
栈式存储分配
非局部名字的访问
参数传递方式

运行环境

7.1 程序运行时的存储组织

7.2 存储分配策略

7.3 非局部名字的访问

7.4 参数传递机制

小 结

7.1 程序运行时的存储组织

- 程序运行时刻的环境，即讨论运行中程序的信息是怎样存储和访问的。
 - ◆ 讨论名字和数据对象之间的关系，把静态程序正文与运行时刻的动作联系起来，源程序正文中相同的名字可以在目标程序中指示不同的数据对象
- 数据对象的空间分配和释放由运行时的支持程序包管理
 - ◆ 由一些和产生的目标代码一起装入的例行程序组成，它的设计受过程语义的影响
- 运行时刻数据对象的表示形式由它的类型来决定
 - ◆ 基本的数据类型可以用目标机器中等价的数据对象来表示
 - ◆ 复杂的数据类型一般用基本数据类型的组合来表示
- 存储组织与管理
 - ◆ 早期的计算机上，这个存储管理工作是由程序员自己来完成
 - ◆ 有了高级语言之后，程序中使用的存储单元都由标识符来表示，它们对应的内存地址由编译程序在编译时或由其生成的目标程序在运行时进行分配。
- 存储的组织及管理是编译程序要完成的一个复杂而又十分重要的工作。

本节内容

概念：过程与活动

7.1.1 程序运行空间的划分

7.1.2 活动记录与控制栈

7.1.3 名字的作用域及名字绑定

概念：过程与活动

- 与程序的执行密切相关的概念。
- 过程的定义
 - ◆ 一个声明语句
 - ◆ 把一个标识符（过程名）和一个语句（过程体）联系起来
- 过程的分类
 - ◆ 过程：没有返回值的函数
 - ◆ 函数：有返回值的函数
 - ◆ 也可以把函数、一个完整的程序看作过程
- 过程引用：过程名出现在一个可执行语句中
- 参数：
 - ◆ 形参、实参

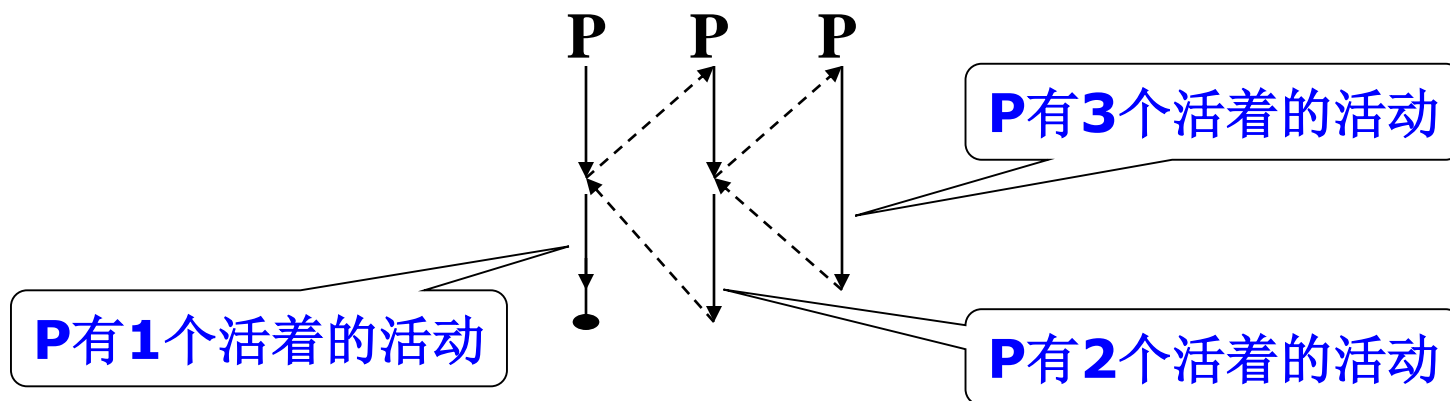
活动

■ 活动

- ◆ 一个过程的每次执行称为它的一次活动。
- ◆ 如果一个过程在执行中，则称它的这次活动是**活着的**。

■ 过程与活动

- ◆ 过程是一个静态概念，活动是一个动态概念。
- ◆ 过程与活动之间可以是**1:1**或者**1:m**的关系。
- ◆ 递归过程，同一时刻可能有若干个活动是活着的。
- ◆ 每个活动都有自己独立的存储空间/数据空间。



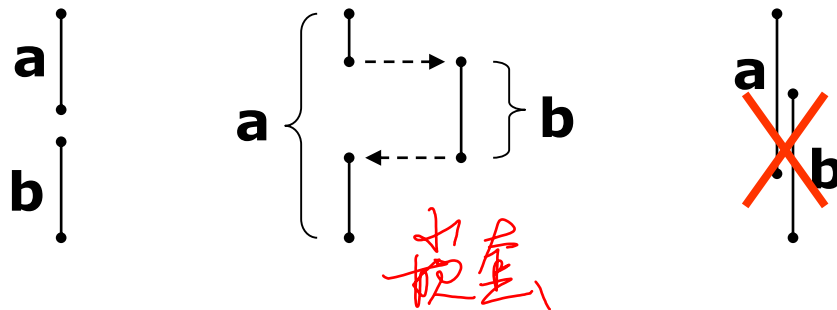
活动的生存期

■ 程序执行时, 过程之间的控制流

- ◆ 是连续的。
- ◆ 过程的每一次执行都是从过程体的起点开始, 最后控制返回到直接跟随本次调用点的位置。

■ 活动的生存期

- ◆ 过程体的执行中, 第一步和最后一步之间的一系列步骤的执行时间。
- ◆ 过程**P**的一次活动的生存期, 包括执行过程**P**所调用的过程的时间, 以及这些过程所调用的过程的时间。
- ◆ 如果**a**和**b**是过程的活动, 那么它们的生存期要么是不重叠, 要么是嵌套的。

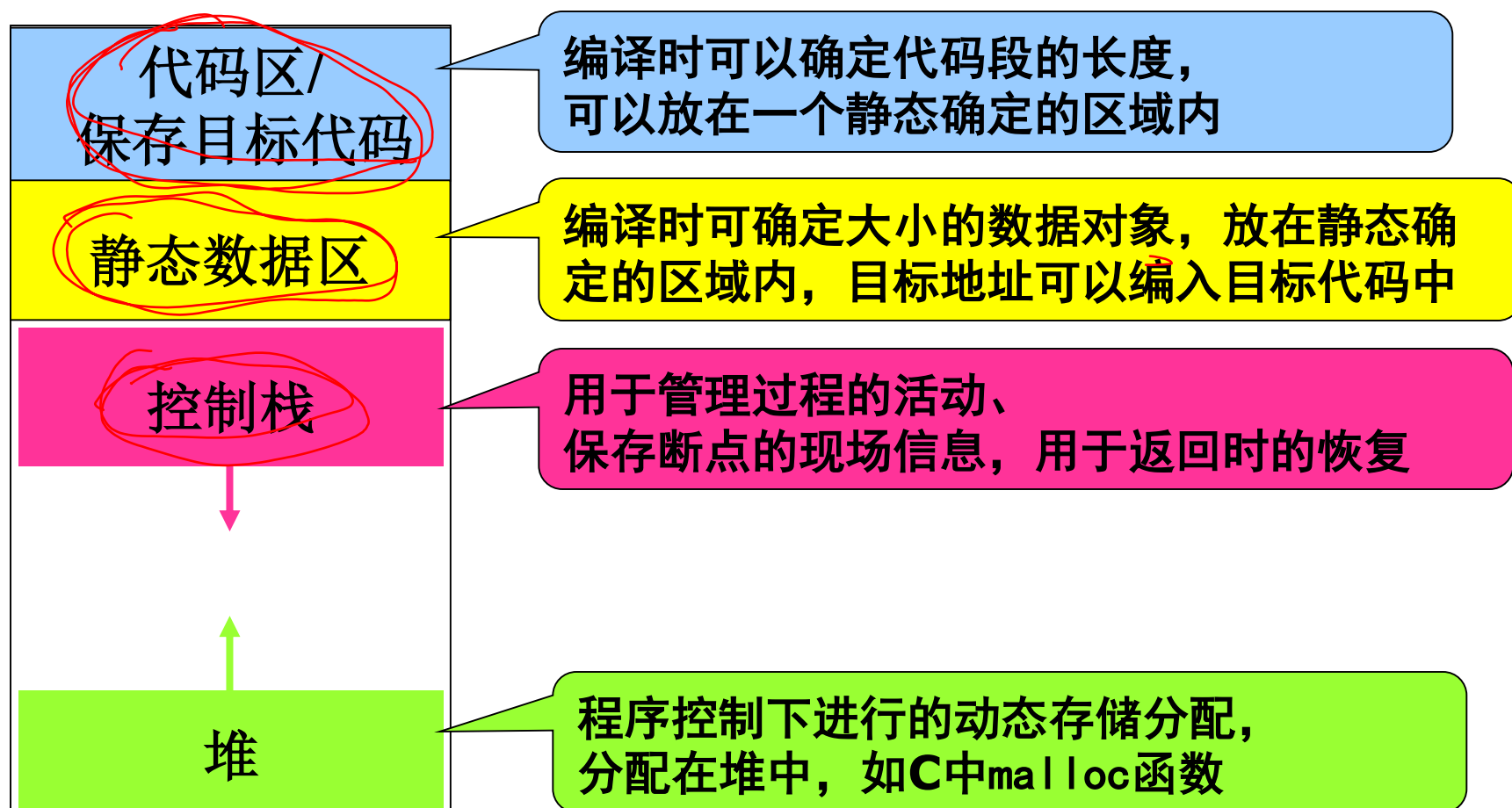


■ 递归过程:

- ◆ 如果一个过程, 它的不同活动的生存期可以嵌套, 则这个过程是递归的。
- ◆ 直接递归、间接递归。

7.1.1 程序运行空间的划分

- 编译程序在编译源程序时，向操作系统申请一块内存区域，以便被编译程序在其中运行。



7.1.2 活动记录与控制栈

- 允许过程递归调用的语言（如Pascal、C等），过程块的活动空间不能静态分配。
 - ◆ 可能：递归过程的前一次活动结束之前又开始了一次新的活动。如程序：

```
int f(int n) {  
    if (n==0) return 1;  
    else return n*f(n-1);  
}
```
 - ◆ 在某段时间内，一个递归过程的多次活动可能同时活着。
 - ◆ 为保证程序的正确执行，需要为过程的每次活动分配不同的存储空间。
 - ◆ 通常采用栈式存储分配策略来实现。
- 活动记录：一个连续的存储块
 - ◆ 记录过程在一次执行中所需要的信息。

活动记录的内容与组织

返回值	本活动返回给调用过程的值
实参区域	调用过程提供给本活动的实参值
控制链	指向调用过程的活动记录的指针，用于本活动结束时的恢复
访问链	指向直接外围过程的最近一次活动的活动记录的指针，用于对非局部名字的访问
机器状态域	保存断点的现场信息，寄存器、PSW等
局部数据区	在本次活动中，为过程中定义的局部变量分配的存储空间
临时数据区	存放中间计算结果

根据确定每个域所需空间大小的时间早晚安排其位置。

(1) 早：中间 晚：两头 (2) 用于通信：前面 自己用的：后面

控制栈与活动记录

- **控制栈：**
 - ◆ 程序运行空间中的存储区域
 - ◆ 以栈的形式组织和管理
 - ◆ 保存当前活着的活动的活动记录
- **控制栈记录活动的生存踪迹及活动的运行环境。**
- **栈空间的分配和回收，即活动记录的入栈和出栈（像Pascal、C）**
 - ◆ 当一个过程被调用时，被调用过程的一次新的活动被激活，在栈顶为该活动创建一个新的活动记录来保存其环境信息；
 - ◆ 当活动结束，控制从被调用过程返回时，释放该活动记录，使调用过程的活动记录成为栈顶活动记录，即恢复调用过程的执行环境。

局部数据的安排

■ 常识：

- ◆ 程序运行时使用连续的存储空间
- ◆ 内存可编址的最小单位是字节
- ◆ 一个机器字由若干个字节组成
- ◆ 一个名字所需存储空间的大小由其类型决定
- ◆ 需多个字节表示的数据对象，存放在连续字节的存储块中，第一个字节的地址作为它的地址

■ 局部数据的安排

- ◆ 局部数据区是在编译过程中处理声明语句时安排的
- ◆ 长度可变的数据对象，放在该区域之外

■ 数据对象的存储安排受目标机器编址限制的影响

编址限制的影响

- 多数机器都有地址对齐的要求。如：整数加法指令可能要求整数的地址能够被4整除。
- 例如，有机器M，它的每个字节都有一个地址，一个字节有8位，为字符型数据对象分配一个字节，整型数据对象分配4个字节，实型数据对象分配8个字节，整型和实型数据对象的存储地址须能够被4整除。

```
struct {  
    char c1;  
    int i;  
    char c2;  
    float x;  
}a;
```

理论：14B
实际：20B

```
struct {  
    char c1;  
    char c2;  
    int i;  
    float x;  
}a;
```

实际：16B

- 为求分配上的全局统一而多余出来的无用空间叫做填塞（padding）

7.1.3 名字的作用域及名字绑定

- 声明是一个把信息与名字联系起来的语法结构
 - ◆ 显式声明（如C中的声明：`int i`）
 - ◆ 隐含声明（如FORTRAN程序）
- 在一个程序的不同部分可能有对同一个名字的相互独立的声明。
- 一个声明起作用的程序部分称为该声明的作用域。
- 语言的作用域规则决定了当这样的名字在程序正文中出现时应该使用哪一个声明。
 - ◆ C和Pascal中的名字遵循“最近嵌套原则”
 - ◆ 名字的局部和非局部
 - ◆ 作用域是名字说明的一个性质
 - ◆ 编译过程中，名字的作用域信息记录在符号表中

名字的作用域示例

■ 下面程序的输出？

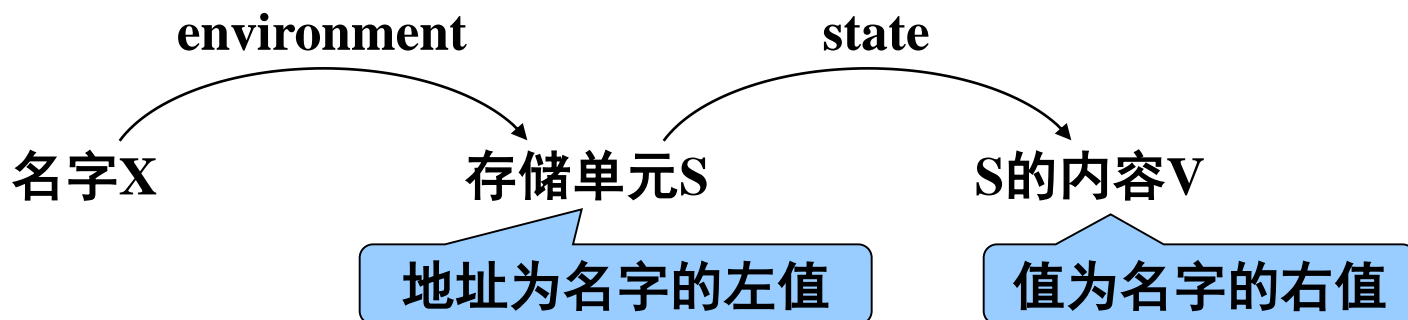
```
main()
{
    int a=0;
    int b=0;
    {
        int b=1;
        {
            int a=2;
            printf("%d %d\n", a, b);
        }
        {
            int b=3;
            printf("%d %d\n", a, b);
        }
        printf("%d %d\n", a, b);
    }
    printf("%d %d\n", a, b);
}
```

名字绑定

- 把名字映射到存储单元的过程。
- 在静态作用域规则下，名字的存储空间被安排在其声明所在过程的活动记录中。
- 名字与存储单元的对应关系
 - ◆ 不同的名字被绑定到不同的存储单元。
 - ◆ 不同作用域的同名变量，分别被绑定到不同的存储单元。
 - ◆ 即使在一个程序中每个名字只被声明一次，程序运行期间，同一个名字也可能映射到不同的存储空间（如递归过程中声明的名字）
 - ◆ 名字与存储单元的对应关系：
 - **1:1** 一个活动中的名字与其存储单元之间
 - **1:m** 一个递归过程中的名字与其存储单元之间

名字绑定（续）

- 程序运行期间，名字的值有左右之分。
 - ◆ 左值指的是其存储空间的地址
 - ◆ 右值指的是其存储空间的内容。
- 两个函数
 - ◆ Environment（环境）：代表名字映射到存储单元的函数
 - ◆ State（状态）：代表从存储单元映射到它保存的值的函数
 - ◆ 环境与状态的区别：赋值改变状态，但不改变环境



- 当environment把一个存储单元S与一个名字X联系起来时，称X受限于S，这个联系本身称为X的一个绑定。
- S的大小取决于X的类型

静态和动态概念的对应

静态概念

过程的定义

名字的声明

声明的作用域

动态对应

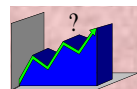
过程的活动

名字的绑定

绑定的生存期

7.2 存储分配策略

- 运行时刻存储空间的划分：
- 除目标代码外，其余三种数据空间采用的存储分配策略是不同的。
 - ◆ 静态存储分配：编译时对所有数据对象分配存储空间。
 - ◆ 栈式存储分配：运行时把存储空间组织成栈进行管理，数据对象分配在栈中。
 - ◆ 堆式存储分配：运行时把存储空间组织成堆进行管理。



7.2.1 静态存储分配

7.2.2 栈式存储分配

7.2.3 堆式存储分配

7.2.1 静态存储分配

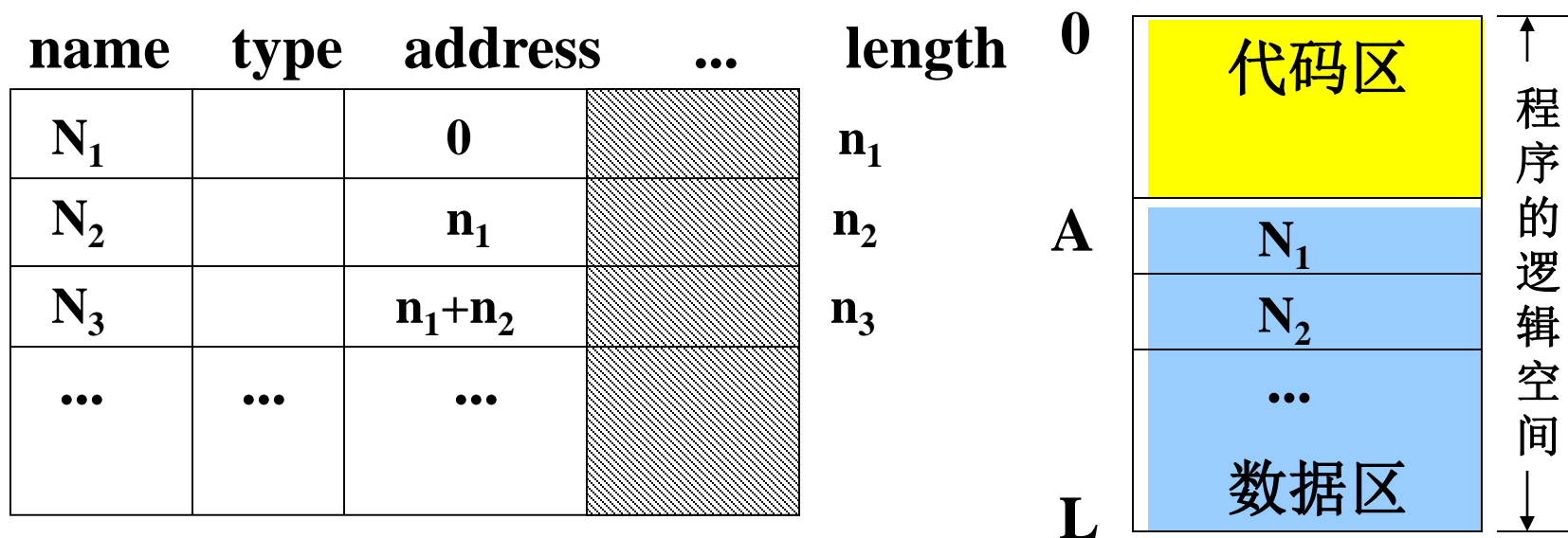
- 条件：源程序中声明的各种数据对象所需存储空间的大小在编译时都可以确定。
- 存储分配：编译时，为他们分配**固定的**存储空间。
 - ◆ 从名字的类型确定该名字所需的存储空间
 - ◆ 这个存储空间的地址用相对于该过程活动记录一端的偏移表示
 - ◆ 最后确定活动记录在目标程序中的位置，如相对于目标代码的位置
 - ◆ 这些都确定后，每个活动记录的位置，以及每个名字在活动记录中的位置就都确定了
 - ◆ 编译时可以在目标代码中填入所要操作的数据对象的地址
- 地址绑定：程序装入内存时进行。
- 运行期间：名字的左值保持不变。
 - ◆ 过程每次被激活，同一名字都使用相同的存储空间。
 - ◆ 允许局部名字的值在活动结束后被保留下来。
 - ◆ 当控制再次进入时，局部名字的值即上次离开时的值。

静态存储分配策略对源语言的限制

- 数据对象的大小和它们在内存中的位置必须在编译时都能够确定。
- 不允许过程递归调用
 - ◆ 因为使用静态存储分配，一个过程中声明的局部名字在该过程的所有活动中都结合到同一个地址。
- 不能建立动态数据结构
 - ◆ 因为没有在运行时进行存储分配的机制。

静态存储分配策略的实现

- 编译程序处理声明语句时，每遇到一个变量名就创建一个符号表条目，填入相应的属性，包括名字、类型、存储地址等。
- 每个变量所需存储空间的大小由其类型确定，并且在编译时是已知的。
- 根据名字出现的先后顺序，连续分配空间。



Fortran程序示例



PROGRAM cmain

```
CHARACTER *50 buff
```

```
INTEGER next
```

```
CHARACTER c, prc
```

```
DATA next /1/, buff /' '/
```

```
6      c=prc()  
      buff (next:next)=c  
      next=next+1  
      IF (c .NE. ' ') GOTO 6  
      WRITE (*, '(A)') buff  
      END
```

输入： welcome to beijing
输出： welcome

CHARACTER FUNCTION prc()

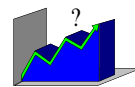
```
CHARACTER *80 buffer
```

```
INTEGER next
```

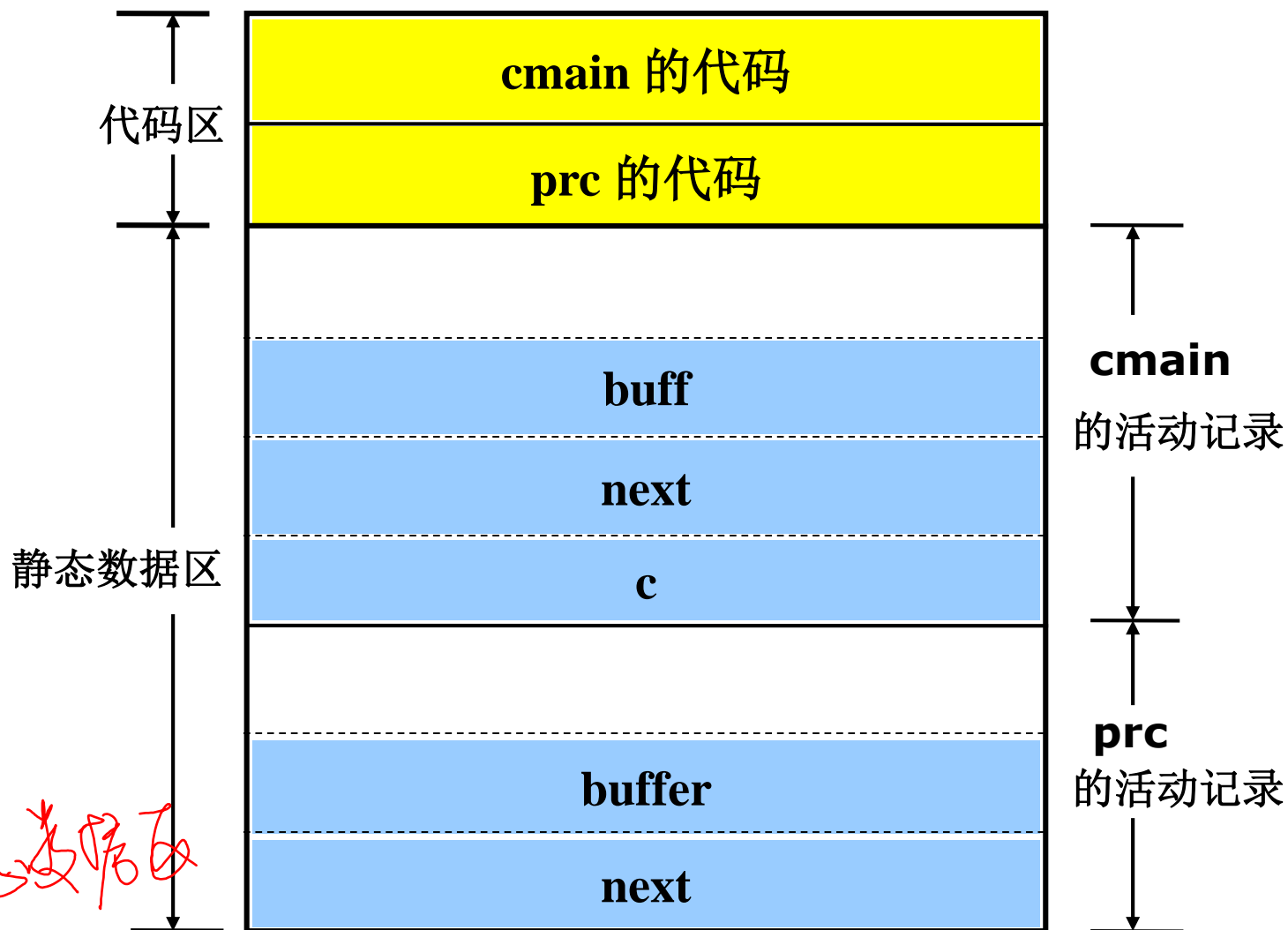
```
SAVE buffer, next
```

```
DATA next /81/
```

```
IF (next .GT. 80) THEN  
    READ (*, '(A)') buffer  
    next=1  
END IF  
prc=buffer(next:next)  
next=next+1  
END
```



存储空间分配

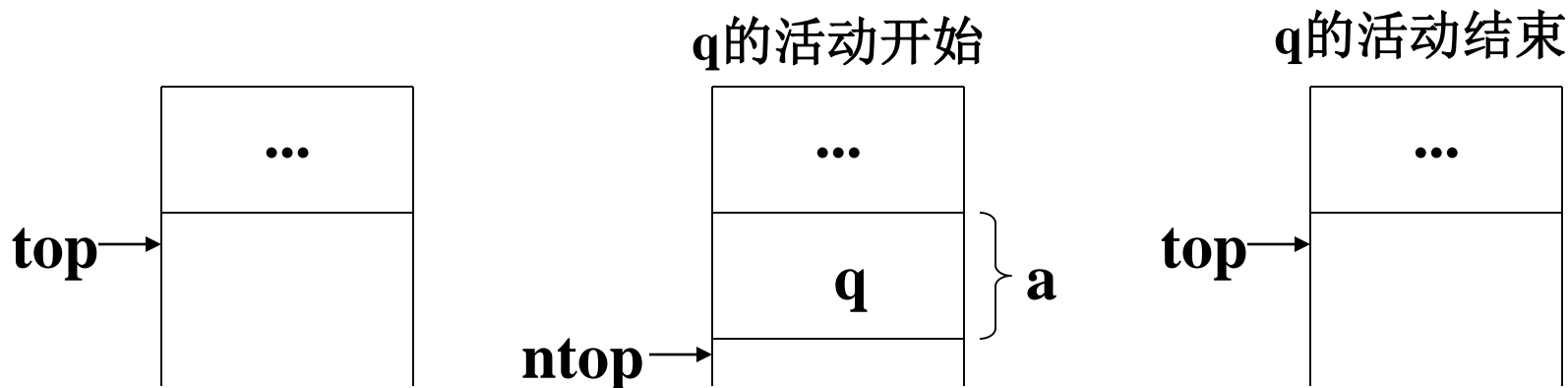


7.2.2 栈式存储分配

- 存储空间被组织成栈

- 存储管理

- ◆ 活动时开始时，与活动相应的活动记录入栈。
局部变量的存储空间分配在活动记录中，同一过程中声明的名字在不同的活动中被绑定到不同的存储空间。
- ◆ 活动结束后，活动记录出栈。
分配给局部名字的存储空间被释放。名字的值将丢失，不可再用。
- ◆ 性质：每次活动时局部量绑定到新的存储，当活动结束后，局部量的值删除



读入数据，并排序 的PASCAL程序

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

```
procedure readarray;  
var i: integer;  
begin  
    for i:=1 to 9 do read(a[i])  
end;
```

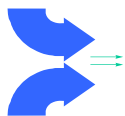
```
prcedure exchange(i, j: integer);  
begin  
    x:=a[i]; a[i]:=a[j]; a[j]:=x  
end;
```

```
procedure quicksort(m, n: integer);  
var k, v: integer;
```

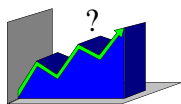
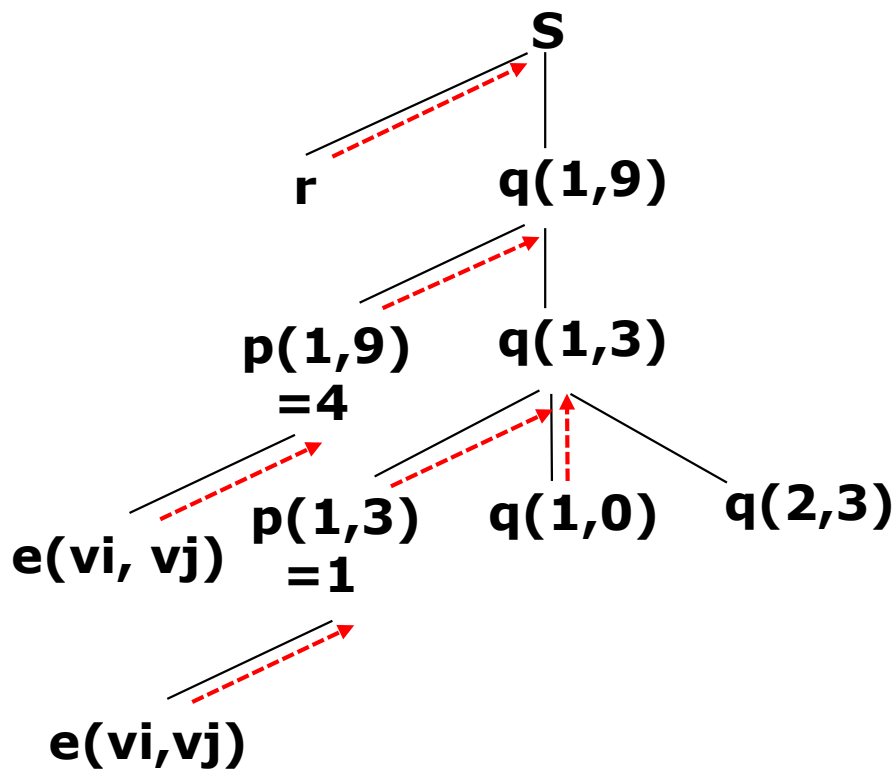
```
function partition(y, z: integer): integer;  
var i, j: integer;  
begin  
    ... a ...; //引用名字a  
    ... v ...; //引用名字v  
    exchange(i, j);  
end;
```

```
begin  
    if (n>m) then begin  
        i:=partition(m, n);  
        quicksort(m,i-1);  
        quicksort(i+1, n)  
    end  
end {quicksort};
```

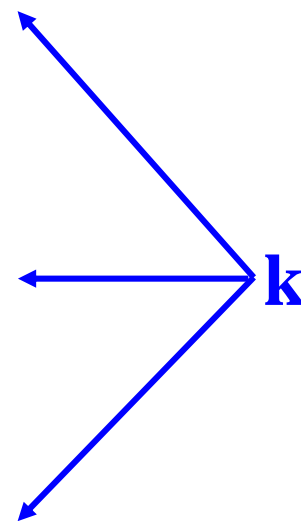
```
begin a[0]:=-999; a[10]=999;  
      readarray; quicksort(1, 9)  
end {sort}.
```



程序运行期间 控制栈的变化举例

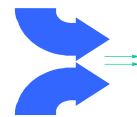


S
a : array
x : integer
q(1,9)
k : integer
v : integer
q(1,3)
k : integer
v : integer
q(2,3)
k : integer
v : integer
e(vi, vj)

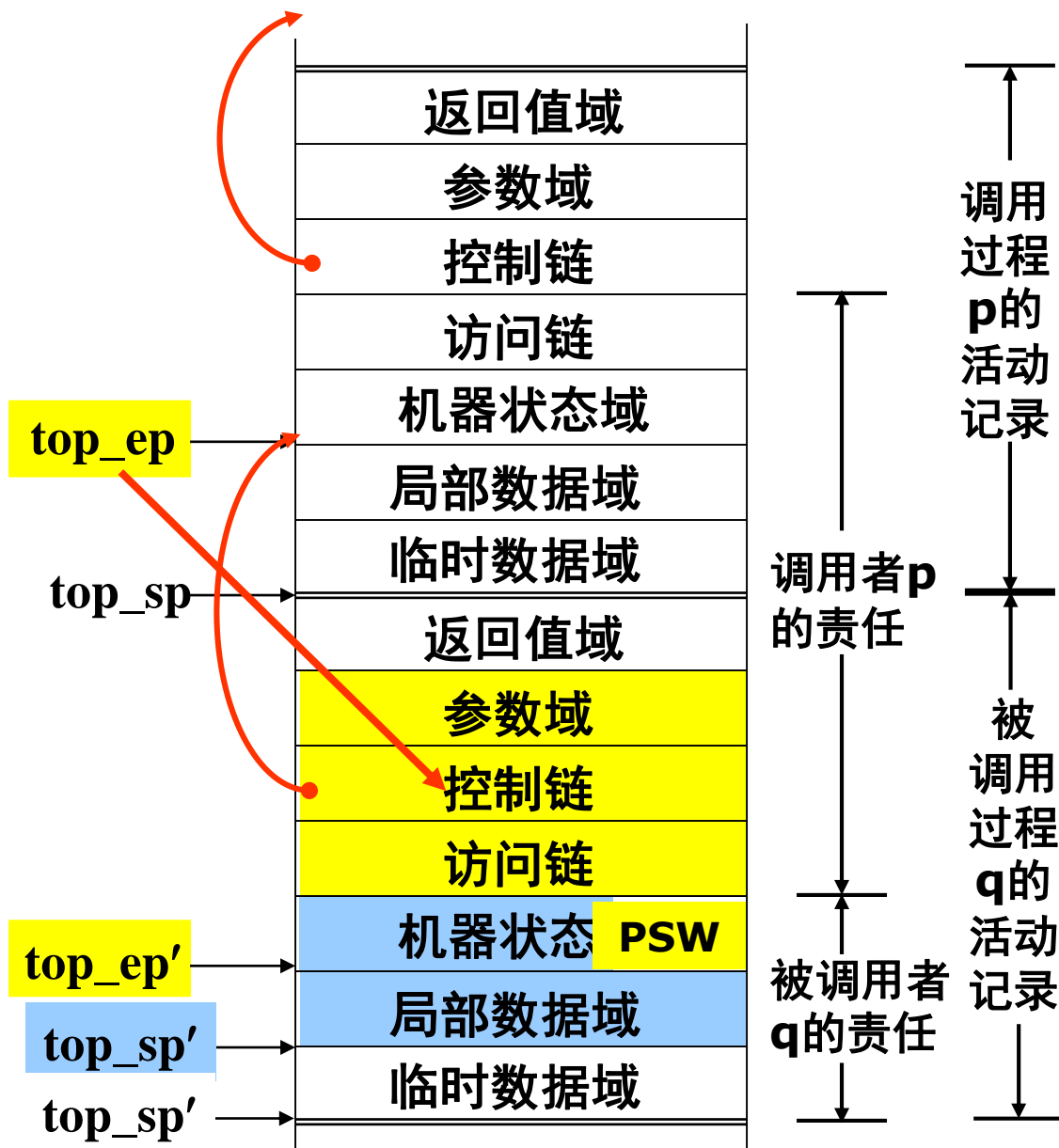


调用序列和返回序列

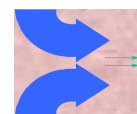
- **调用序列：**目标程序中实现控制从调用过程进入被调用过程的一段代码。
 - ◆ **功能：**实现活动记录的入栈和控制的转移。
 - ◆ **其中有调用过程和被调用过程各自需要完成任务。**
如：
 - **调用者：**准备并传递实参，为被调用者创建访问非局部名字的环境等。
 - **被调用者：**保存调用点环境、初始化局部变量等。
- **返回序列：**目标程序中实现控制从被调用过程返回到调用过程的一段代码。
 - ◆ **功能：**实现活动记录的出栈和控制的转移。
 - ◆ **其中有调用过程和被调用过程各自需要完成任务。**

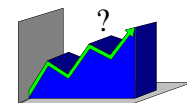


调用序列



- C1: 控制链域 + 访问链域 + 机器状态域
- C2: 返回值域 + 参数域
- $top_ep' = top_sp + C2 + C1$
- $top_sp = top_ep' - C1 - C2$





调用序列的安排

■ 参数传递

- ◆ P准备实参，参数传递入q的活动记录的参数域；

■ 控制信息设置

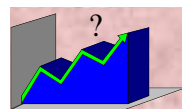
- ◆ p将返回地址写入q的活动记录的机器状态域中；
- ◆ p将当前top_ep的值写入q的活动记录的控制链域；
- ◆ p为q建立访问链；
- ◆ P计算并重置top_ep的值(指向top_ep'位置)

■ 进入q的代码(goto语句)

- ◆ q保存寄存器的值、其他机器状态信息；
- ◆ q增加top_sp的值，初始化局部变量；
- ◆ q开始执行

返回序列

- **q**把返回值写入自己活动记录的返回值域；
- **q**恢复调用点状态
 - ◆ 寄存器的值、机器状态；
 - ◆ **top_ep**的值（保存在控制链中）；
 - ◆ 计算并重置**top_sp**的值；
- 根据返回地址返回到**p**的代码中（将保存值写回程序计数器**PC**中）
- **P**把返回值取入自己的活动记录中
- **p**继续执行



调用序列与活动记录

■ 区别：

- ◆ 活动记录是一块连续的存储区域，保存一个活动所需的全部信息，与活动一一对应。
- ◆ 调用序列是一段代码，完成活动记录的入栈，实现控制从调用过程到被调用过程的转移。
- ◆ 调用序列逻辑上是一个整体，物理上被分成两部分，分属于调用过程和被调用过程。

■ 联系：

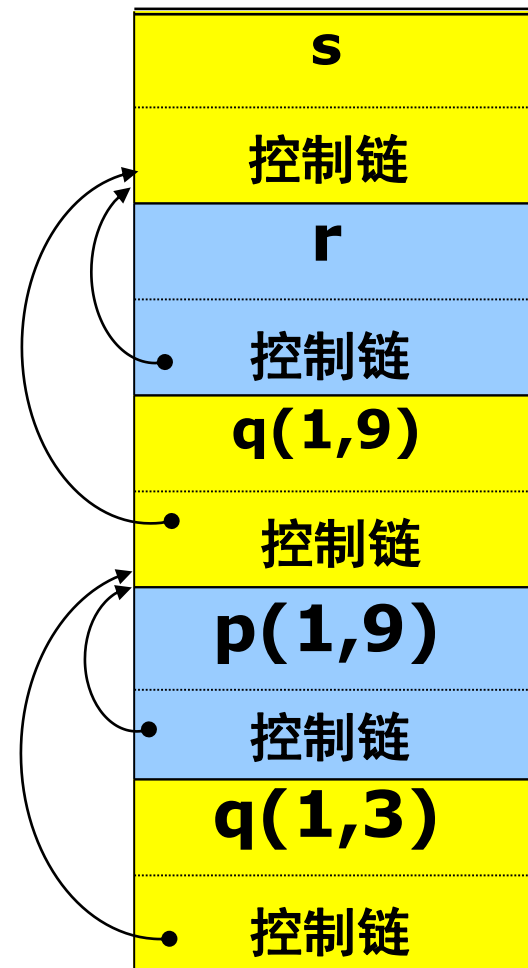
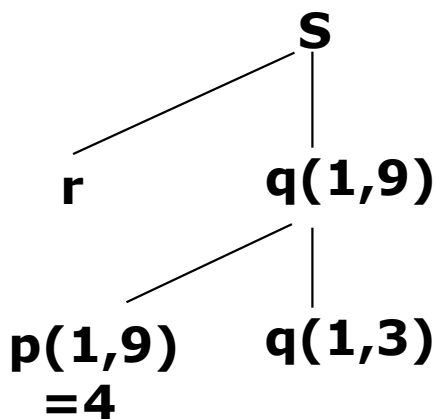
- ◆ 调用序列的实现与活动记录中内容的安排有密切关系。

7.2.3 堆式存储分配

- 如果具体的存储需求在编译时刻可以确定
——采取静态存储分配策略
 - 如果某些存储需求在编译时不能确定，但在程序执行期间，在程序的入口点上可以知道
——采用栈式存储分配策略
 - 栈式存储分配策略不能处理的存储需求：
 - ◆ 活动停止时局部名字的值必须被保存下来；
 - ◆ 被调用过程的活动生存期超过调用过程的生存期，这种语言的过程间的控制流不能用活动树正确地描述；
 - ◆ 程序中某些动态数据结构（如链表、树）的存储需求。
- 共性：**活动记录的释放不需要遵循先进后出的原则
- 堆式存储分配
 - ◆ 把连续存储分成块，
 - ◆ 当活动记录或其它对象需要时就分配，
 - ◆ 块的释放可按任意次序
 - ◆ 类似于操作系统中的段式存储分配

堆式存储分配示例

- sort程序执行期间，活动记录：

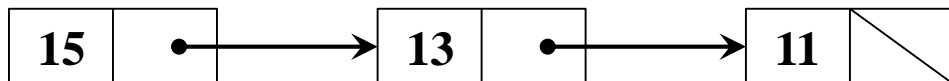


动态创建或撤消一个数据结构

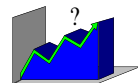
- 在处理链表、树和图等数据结构时，需要根据具体情况随机地增加或删除某些结点。

- C语言链表的例子：

```
struct node{  
    int data;  
    struct node * nextPtr;  
};
```



- ◆ 动态申请一个新结点： `newPtr=malloc(sizeof(struct node));`
- ◆ 释放占用的内存： `free(newPtr);`
- 链表所需存储空间在编译时无法确定，空间的分配和释放依赖于程序的执行，必须采取堆式存储分配策略来解决。



堆式与栈式存储分配的比较

■ 相同点：

- ◆ 动态存储分配
- ◆ 在程序执行期间进行存储分配

■ 不同点：

- ◆ 组织形式不同：栈、堆
- ◆ 释放顺序：
栈：后进先出
堆：按需、任意
- ◆ 堆中存活的活动记录不一定是邻接的

7.3 非局部名字的访问

- 如何处理对非局部名字的引用取决于**作用域规则**
 - 静态作用域规则：词法作用域规则、**最近嵌套**规则。由程序正文中名字声明的位置决定。
 - 动态作用域规则：由运行时最近的活动决定应用到一个名字上的声明。
- 对非局部名字的访问通过**访问链**实现。
- 关键：访问链如何创建、使用、维护

7.3.1 程序块

7.3.2 静态作用域规则下非局部名字的访问

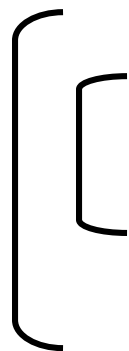
7.3.3 动态作用域规则下非局部名字的访问

7.3.1 程序块

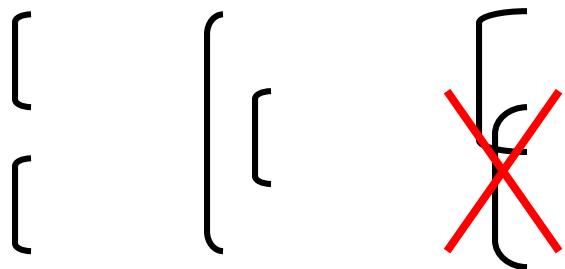
- 块：含有本身局部数据说明的复合语句
- C语言程序块的基本结构

```
{  
    声明语句  
    语句序列  
}
```

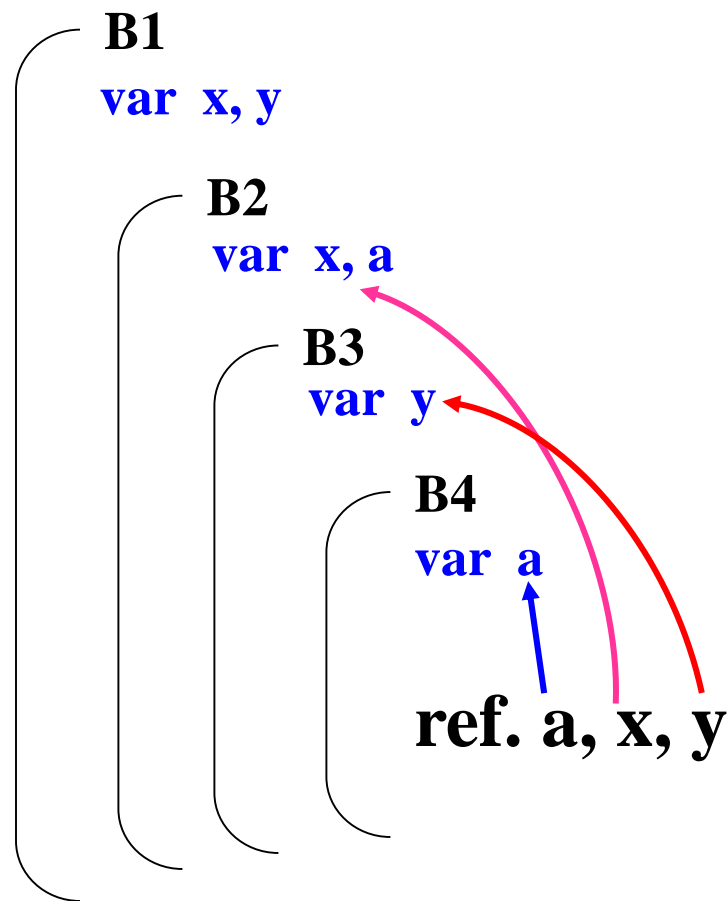
- 块可以嵌套



- 块之间的关系



- 最近嵌套规则



静态作用域示例

■ 名字的作用域

块	声 明	作 用 域
B_0	<code>int a=0</code>	B_0-B_2
B_0	<code>int b=0</code>	B_0-B_1
B_1	<code>int b=1</code>	B_1-B_3
B_2	<code>int a=2</code>	B_2
B_3	<code>int b=3</code>	B_3

■ 栈式存储分配

a=0
b=0
b=1
b=3

main()

```

{
    int a=0;
    int b=0; ←
    {
        int b=1; ←
        {
            B2
            int a=2; ←
            printf("%d %d\n",a,b); 2, 1
        }
        {
            B3
            int b=3; ←
            printf("%d %d\n",a,b); 0, 3
        }
        printf("%d %d\n",a,b); 0, 1
    }
    printf("%d %d\n",a,b); 0, 0
}
    
```

7.3.2 静态作用域规则下非局部名字的访问

■ 块结构语言分为两类：

◆ 非嵌套过程语言：

- 过程定义不允许嵌套，即一个过程定义不能出现在另一个过程定义之中。
- 语言代表：C语言

◆ 嵌套过程语言

- 过程定义允许嵌套
- 语言代表：Pascal语言

■ 程序运行时刻数据空间的分配有差别

■ 对非局部名字访问的实现方式不同

1. 非嵌套过程

- 过程定义不允许嵌套
 - ◆ 独立的
 - ◆ 顺序的
- 声明语句的位置
 - ◆ 过程/函数内部
 - ◆ 所有过程之外
- 在一个过程中引用的名字
 - ◆ 局部的
 - ◆ 全局的

```
int a[11];  
int x;
```

```
void readarray( )  
{ int i; ... a ... }
```

```
void exchange(int i, int j)  
{ ... a ...; ... x ... }
```

```
int partition(int y, int z) {  
    int i, j;  
    ... a ...;  
    exchange(i, j);  
    ...  
}
```

```
void quicksort(int m, int n) {  
    int i;  
    ...  
    i=partition(m, n);  
    quicksort(m, i-1);  
    quicksort(i+1, n);  
}
```

```
void main ( )  
{ readarray(); quicksort(0,10); }
```

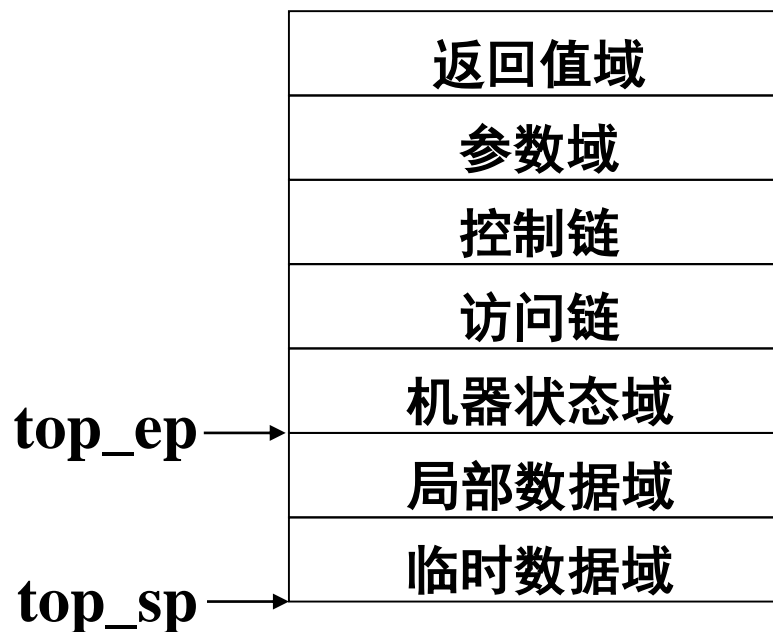
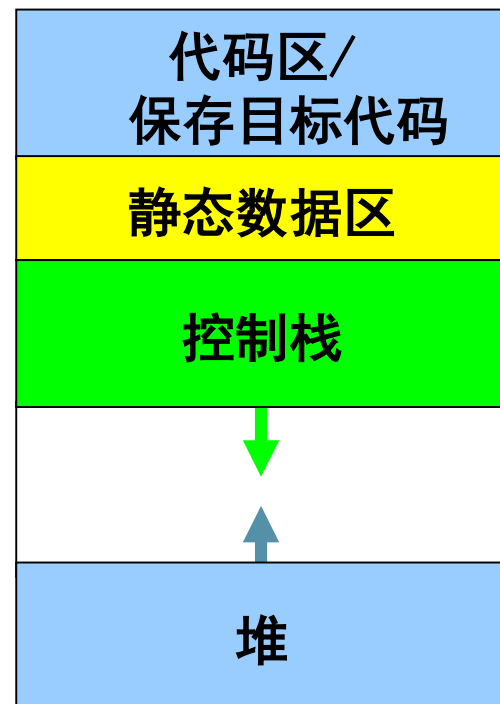
变量的存储分配

■ 全局变量

- ◆ 静态地进行分配
- ◆ 分配在静态数据区中
- ◆ 编译时知道它们的位置，可以将全局变量对应的存储单元的地址编入目标代码中

■ 局部变量

- ◆ 动态地进行分配
- ◆ 分配在活动记录中
- ◆ 栈式存储分配
- ◆ 通过对栈环境指针top_ep的偏移访问当前活动记录中的局部名字



2. 嵌套过程

- 过程定义允许嵌套
- 最近嵌套规则

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

```
  procedure readarray;  
    var i: integer;  
    begin  
      for i:=1 to 9 do read(a[i])  
    end;
```

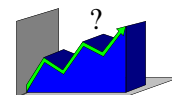
```
  prcedure exchange(i, j: integer);  
    begin  
      x:=a[i]; a[i]:=a[j]; a[j]:=x  
    end;
```

```
  procedure quicksort(m, n: integer);  
    var k, v: integer;
```

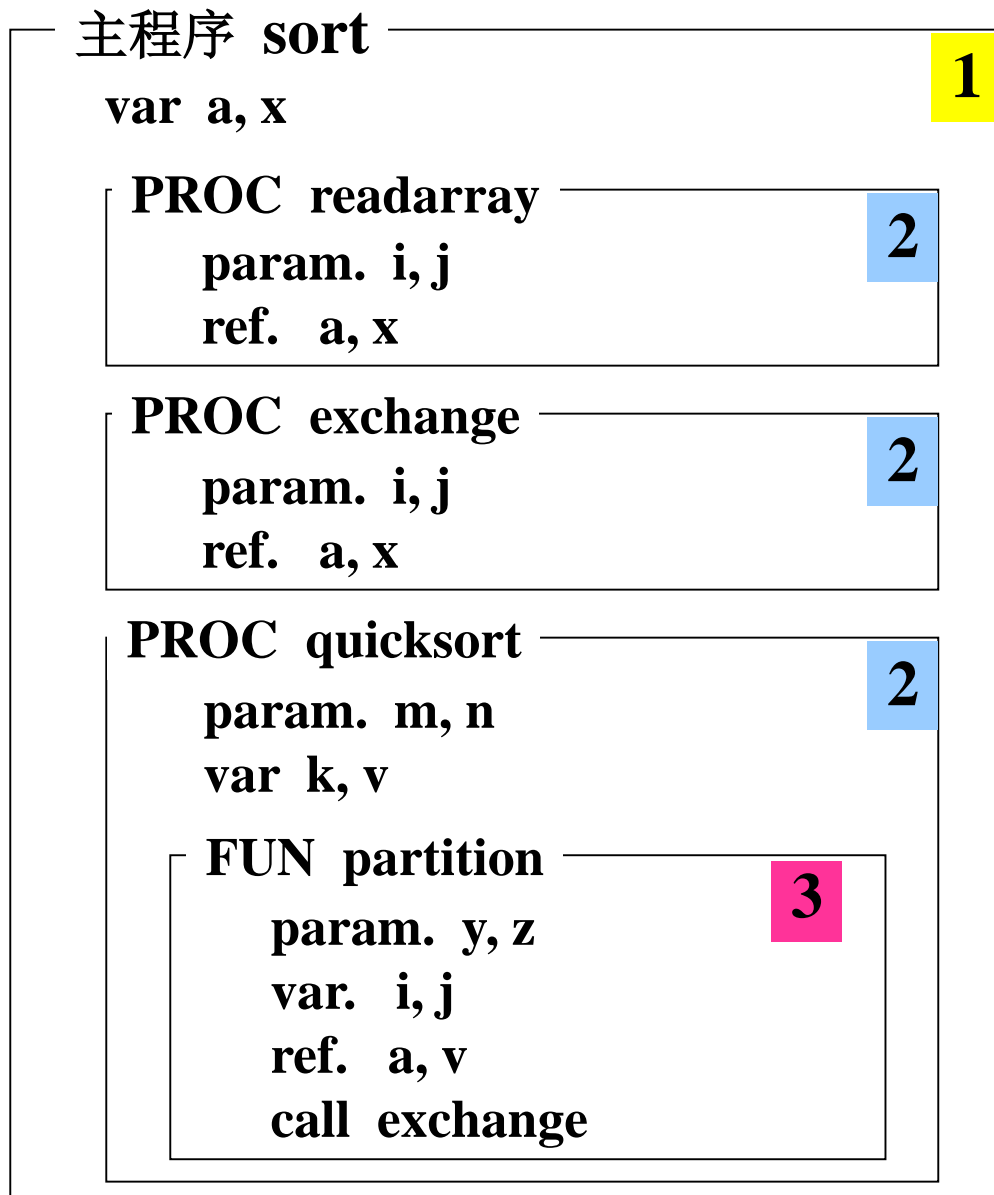
```
  function partition(y, z: integer): integer;  
    var i, j: integer;  
    begin  
      ... a ...; //引用名字a  
      ... v ...; //引用名字v  
      exchange(i, j);  
    end;
```

```
  begin  
    if (n>m) then begin  
      i:=partition(m, n);  
      quicksort(m,i-1);  
      quicksort(i+1, n)  
    end  
  end {quicksort};
```

```
begin a[0]:=-999; a[10]=999;  
      readarray; quicksort(1, 9)  
end {sort}.
```



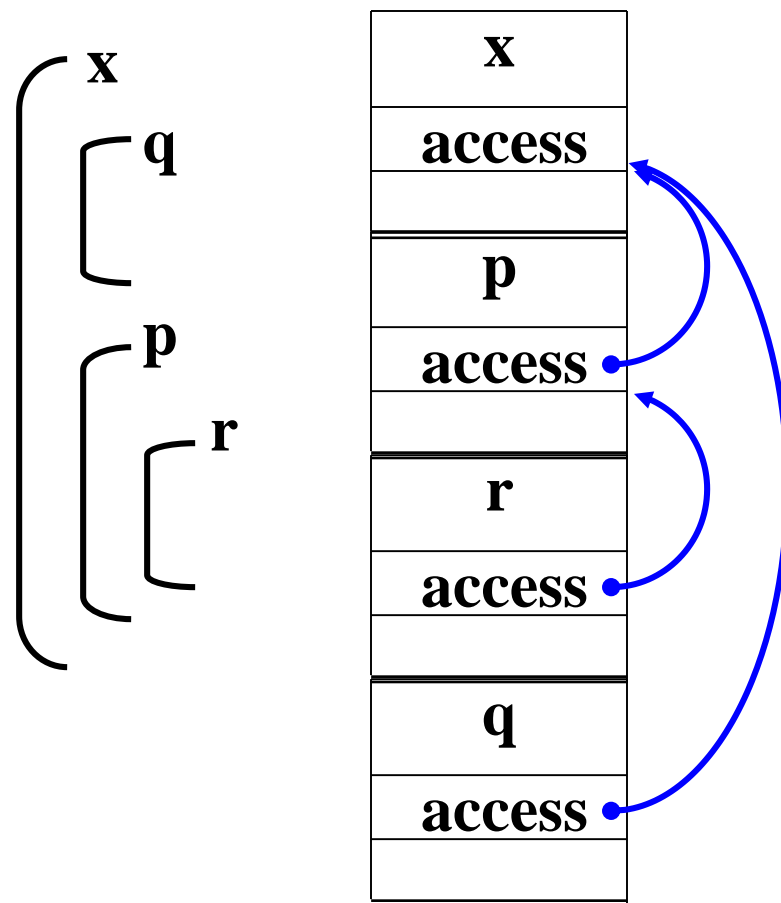
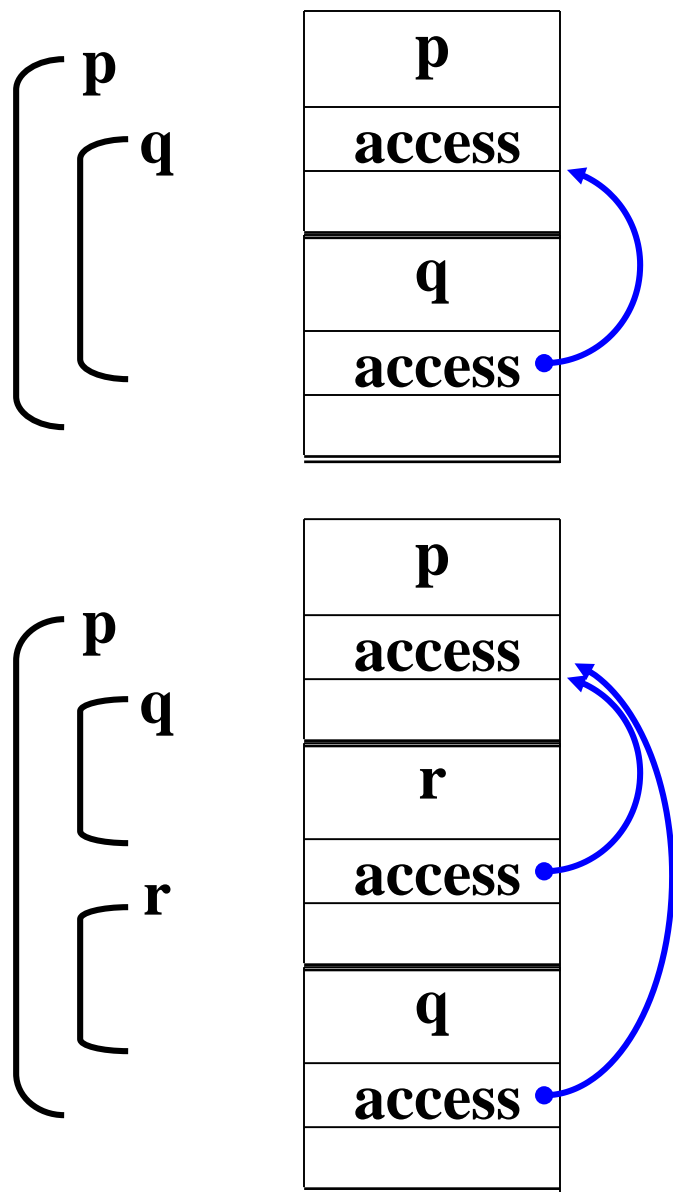
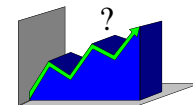
过程及名字的嵌套关系



- 嵌套深度
 - ◆ 主程序：1
 - ◆ 每进入一个过程，深度加1
- 名字的嵌套深度
 - 声明时所在过程的嵌套深度
- 最近嵌套规则同样适用于过程名。

访问链

- 实现嵌套过程的静态作用域规则
- 通过访问链可以实现对非局部名字的访问



被调用过程活动记录的访问链指向其直接外层过程的最新活动的活动记录！

访问链的使用

- 过程 p 引用非局部名字 a ，嵌套深度分别为 n_p ， n_a
 - ◆ $n_a < n_p$ (非局部名字 a 一定在过程 p 的外围过程中定义)
- 当控制处于 p 中时， p 的活动记录在栈顶
- 访问链的使用
 - ◆ 从栈顶活动记录出发，沿访问链前进 $n_p - n_a$ 步
 - ◆ 到达 a 的声明所在过程的最新活动记录
 - ◆ 在该活动记录中，相对于数据区起点偏移某个固定值即 a 的存储位置
- 例子
- 符号表中，变量名字的目标地址：
 - \langle 嵌套深度，偏移量 \rangle
- p 中非局部名字 a 的地址：
 - $\langle n_p - n_a, a$ 在活动记录中相对于数据区起点的偏移量 \rangle

该值在编译时可以计算出来

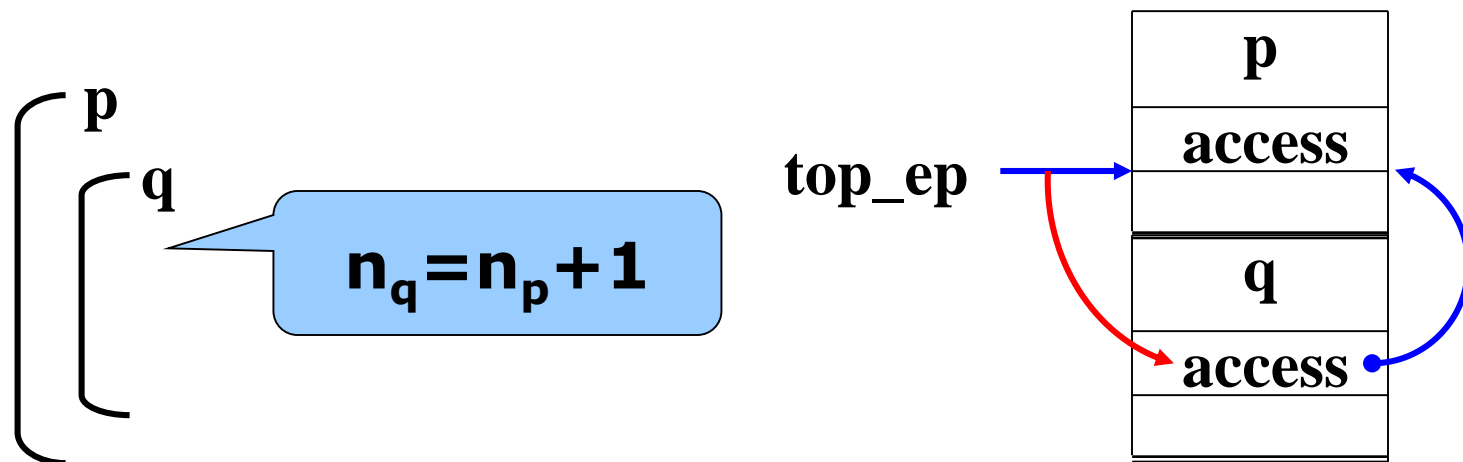
该值在编译时可以确定

访问链的建立

- 调用序列中，由调用过程 p 为被调用过程 q 创建访问链。
- 过程 p 和 q 的嵌套深度分别为 n_p 和 n_q
- q 的活动记录中访问链的建立方式依赖于 q 与 p 的关系
 - ◆ q 直接嵌套在 p 中
 - $n_q = n_p + 1$
 - ◆ q 不嵌套在 p 中
 - $n_q = n_p$
 - $n_q < n_p$

q嵌套在p中—— $n_q = n_p + 1$

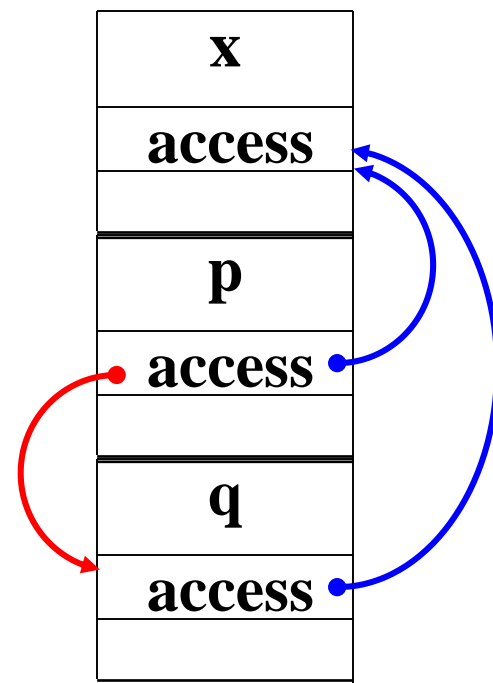
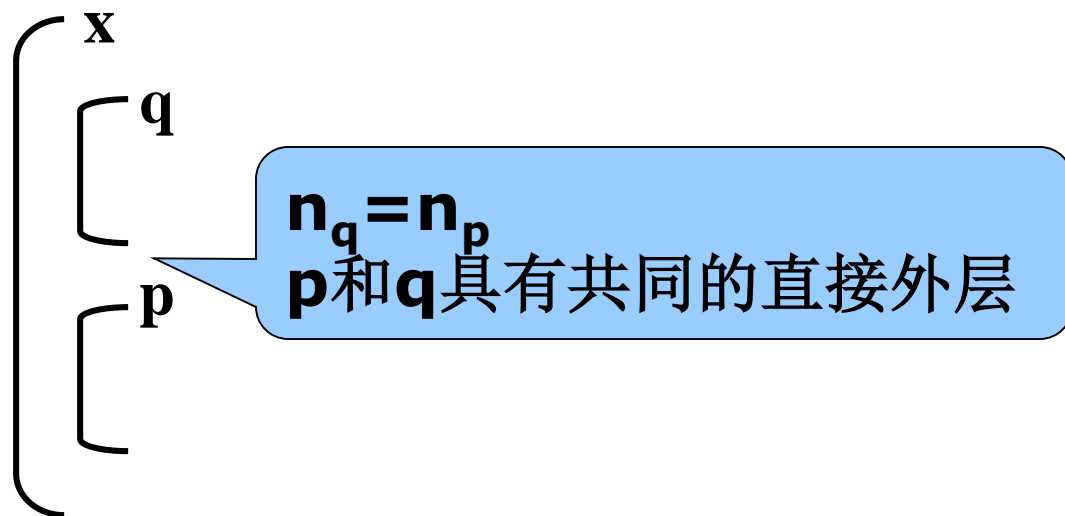
- 静态文本中q与p的关系：



- q 活动记录中的访问链指向栈中刚好在其前面的 p 的活动记录。
- 调用序列中， p 把**top-ep**的值写入 q 的活动记录的访问链域即可。

q不嵌套在p中—— $n_q = n_p$

- 静态文本中q与p的关系：

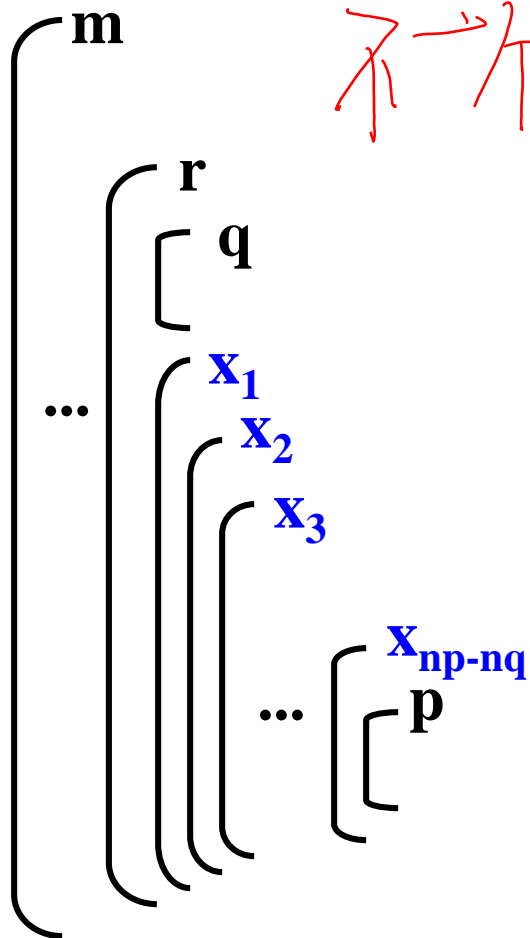


- q的活动记录的访问链与p的活动记录的访问链一样，都指向栈中刚好在p前面的x的活动记录。
- 调用序列中，p把自己的访问链域的值复制到q的活动记录的访问链域即可。

复制

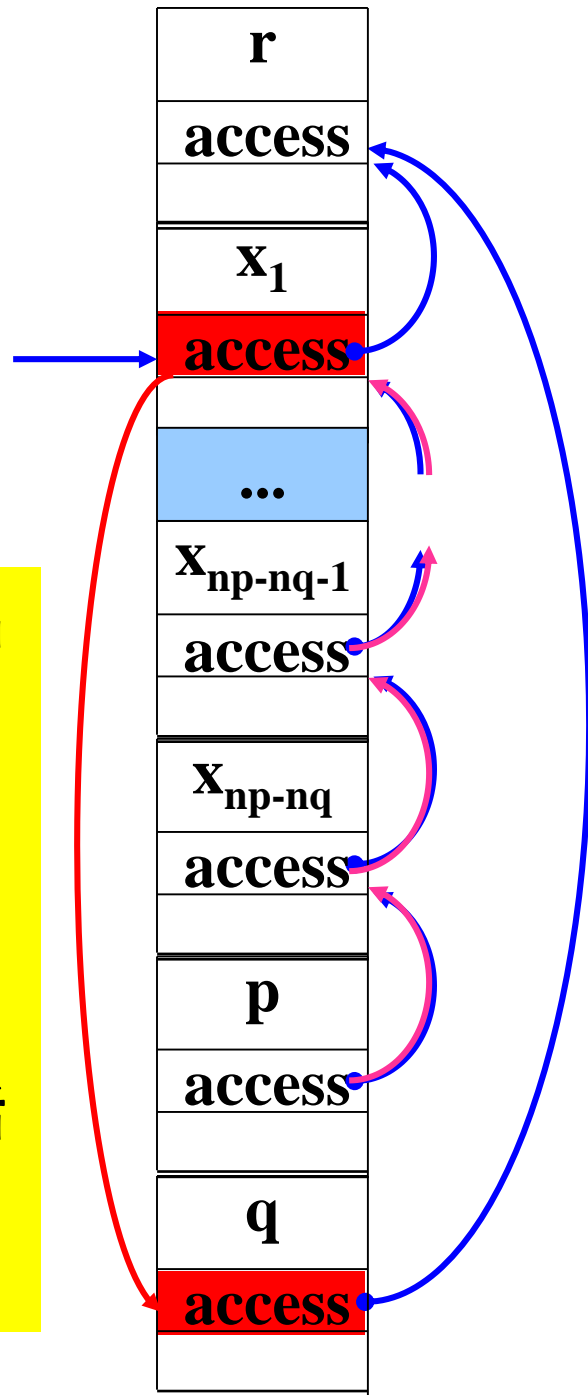
q不嵌套在p中—— $n_q < n_p$

- 静态文本中q与p的关系: p与q有相同的嵌套深度为 $1, 2, \dots, n_q - 1$ 的外围过程



不是一个平方不定式

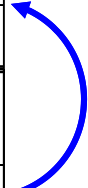
- 从p的活动记录出发，沿访问链前进 $n_p - n_q$ 步；
- 找到 x_1 的活动记录
- 把 x_1 的访问链域的值复制到q的活动记录中访问链域。



访问链举例

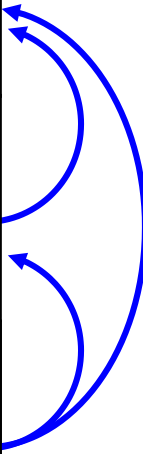
(a)

s
access
a, x
q(1,9)
access
k, v



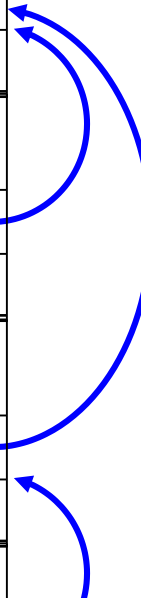
(b)

s
access
a, x
q(1,9)
access
k, v
q(1,9)
access
i, j



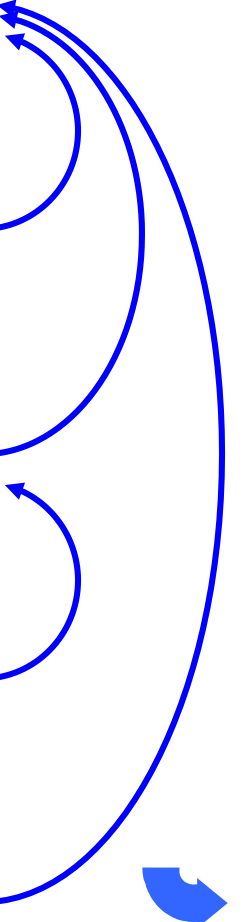
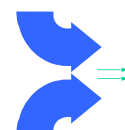
(c)

s
access
a, x
q(1,9)
access
k, v
q(1,3)
access
k, v
p(1,3)
access
i, j



(d)

s
access
a, x
q(1,9)
access
k, v
q(1,3)
access
k, v
p(1,3)
access
i, j
e(vi, vj)
access

display表

- 目的：为了提高访问非局部名字的速度
- display表（简称d表）：
 - ◆ 指针数组d
 - ◆ 每一个指针指向一个活动记录
 - ◆ $d[i]$ 指向嵌套深度为 i 的过程的最新活动的活动记录
 - ◆ 前 $i-1$ 个元素指向按静态规则包围过程P的那些过程的最新的活动记录
- d表的组织
 - ◆ 全程数组
 - ◆ 元素个数在编译时刻根据过程的最大嵌套深度确定
 - ◆ 静态存储分配
 - ◆ 控制栈中，具有相同嵌套深度 j 的各活动记录，从靠近栈顶的最新活动记录开始，通过访问链链成一个链表
 - ◆ $d[j]$ 是该链表的头指针。

Display表的维护

■ d表的维护

- ◆ 调用序列和返回序列的功能之一。
- ◆ 通过追踪访问链可以完成对d表的更新。

■ 假设：过程p（嵌套深度为 n_p ）调用过程q（嵌套深度为 n_q ）

■ 过程调用时，调用序列中

- ◆ 将q的最新活动记录压入栈顶，并将其插入到 $d[n_q]$ 所指链表中，使其成为该链表的首结点。

■ 活动结束时，返回序列中

- ◆ 删除 $d[n_q]$ 链表的首结点。

d[3]	
d[2]	
d[1]	

S
null
a, x
q(1,9)
null
k, v

d[3]	
d[2]	
d[1]	

S
null
a, x
q(1,9)
null
k, v
q(1,9)
save d[2]
k, v

d[3]	
d[2]	
d[1]	

S
null
a, x
q(1,9)
null
k, v
q(1,3)
save d[2]
k, v
p(1,3)
null
i, j

d[3]	
d[2]	
d[1]	

S
null
a, x
q(1,9)
null
k, v
q(1,3)
save d[2]
k, v
p(1,3)
null
i, j
e(vi, vj)
save d[2]

d表应用举例

7.4 参数传递机制

■ 过程之间传递数据的方式：

- ◆ 非局部名字
- ◆ 参数
- ◆ 如：

```
procedure exchange(i, j: integer);  
  var x: integer;  
  begin  
    x:=a[i]; a[i]:=a[j]; a[j]:=x  
  end;
```

■ 表达式代表的含义

- ◆ 如：a[i]:=a[j]; a:=b;
- ◆ a[i]、a 代表存储单元的地址，即左值。 *a[i]:=a[j] 左值*
- ◆ a[j], b 代表存储单元的内容，即右值。 *a:=b 右值*

■ 形参和实参联系的四方法

- ◆ 了解参数传递方法的重要性：程序的结果依赖于所用的方法
- ◆ 不同的方法出自对表达式所表示的东西做的不同解释
- ◆ 参数传递的区别：基于参数是代表右值、左值还是实参名本身（正文）

■ 参数传递机制：

传值调用、引用调用、复制恢复、传名调用。

7.4.1 传值调用 (call-by-value)

- 最一般、最简单的参数传递方法。
- 先计算出实参的值，然后将其右值传递给被调用过程。参数值在被调用过程执行时如同常数。
- 用相应的实参的值替代过程体中出现的所有形参。
- 如： `int max(int x, int y) { return x>y? x : y ; }`
 - ◆ `max(5, 3+4):`
 - ◆ 将`x`替换为`5`，`y`替换为`7`，得到：`5>7? 5:7`。
- 函数式语言中唯一的参数传递机制。
C++、Pascal语言的内置机制。
C语言、Java语言唯一的参数传递机制。
- 在这些语言中，参数被看作是过程的局部变量，初值由调用过程提供的实参给出。
- 在过程中，参数和局部变量一样可以被赋值，但其结果不影响过程体之外的变量的值。

传值调用的实现

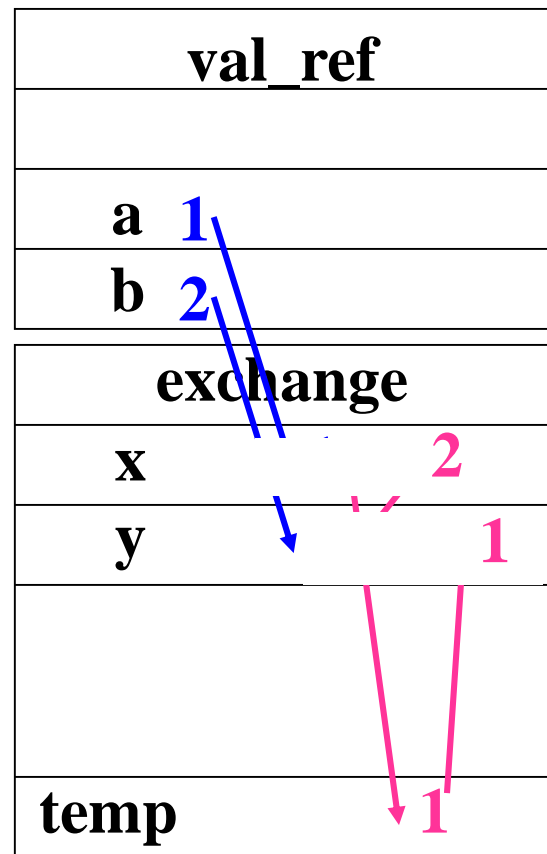
- 把形参当作过程的局部名字看待，形参的存储单元分配在被调用过程的活动记录中（即参数域）。
- 调用过程先对实参求值，发生过程调用时，由调用序列把实参的**右值**写入被调用过程活动记录的参数域中。（调用序列的功能之一）
- 被调用过程执行时，对形参的操作在其活动记录的参数域上进行。

传值调用示例

传值调用，参数的使用会不会影响过程体外变量的值？

```
program val_ref(input,output);  
  var a, b: integer;  
  procedure exchange(x, y: integer);  
    var temp: integer;  
  begin  
    temp:=x;  
    x:=y;  
    y:=temp  
  end;  
begin  
  a:=1; b:=2;  
  exchange(a, b);  
  writeln('a=', a);  
  writeln('b=', b)  
end.
```

过程体



执行结果
a=1
b=2

参数是指针类型的情况

- 传值调用并不意味着参数的使用一定不会影响过程体外变量的值。

- 如果参数的类型为指针，参数的值就是一个地址

- ◆ 通过它可以改变过程体外部的内存值。如，有C语言函数

```
void init_ptr(int *p)
```

```
{ *p=3; }
```

- ◆ 对参数p的直接赋值不会改变过程体外的实参的值。如：

```
void init_ptr(int *p)
```

```
{ p=(int *) malloc(sizeof(int)); }
```

- 在一些语言中，某些值是隐式指针

- ◆ 如C语言中的数组是隐式指针（指向数组空间的起始位置）

- ◆ 可以使用数组参数来改变存储在数组中的值。如：

```
void init_array_0(int p[])
```

```
{ p[0]=0;}
```

7.4.2 引用调用 (call-by-reference)

- 原则上要求：实参必须是已经分配了存储空间的变量。
- 调用过程把实参存储单元的地址（即一个指向实参存储单元的指针）传递给被调用过程的相应形参。
- 被调用过程执行时，通过形参间接地引用实参。
 - ◆ 可以把形参看成是实参的别名，对形参的任何引用都是对相应实参的引用。
- FORTRAN语言唯一的参数传递机制。
- Pascal语言，通过在形参前加关键字var来指定采用引用调用机制。

```
procedure inc_1(var x: integer);  
begin x:=x+1 end;
```

形参前加关键字var指定
引用调用

- C++中，通过在形参的类型关键字后加符号‘&’指明采用引用调用机制，如：

```
void inc_1(int &x) { x++ ; }
```

引用调用（续）

- C语言只有传值调用机制，但可以通过传递引用或显式指针来实现引用调用的效果。
- C语言使用 ‘&’ 指示变量的地址，使用操作 ‘*’ 撤销引用指针。
- 如：

```
int a;
```

```
void inc_1(int *x)
```

```
// C 模拟引用调用
```

```
{ (*x)++; }
```

```
.....
```

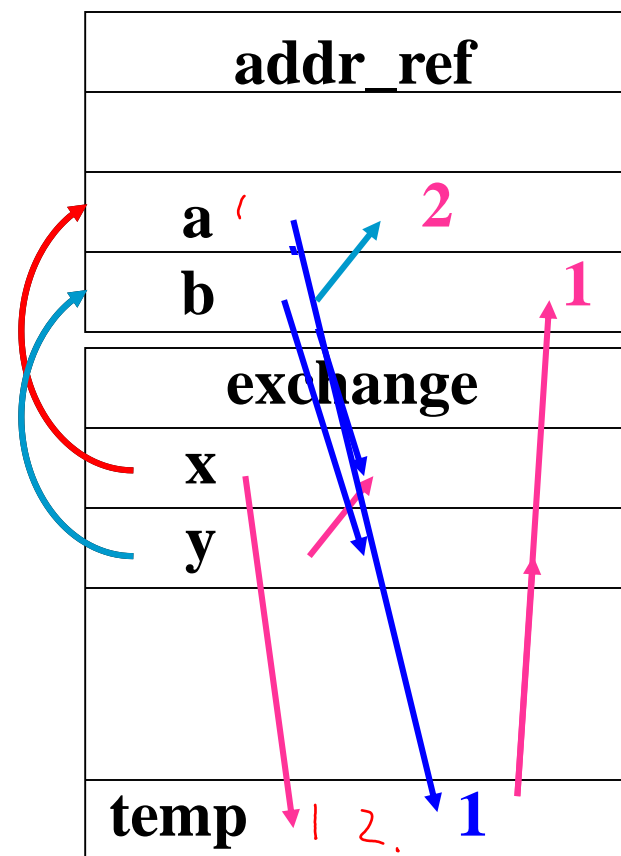
```
inc_1(&a);
```

引用调用的实现

- 调用过程对实参求值。
- 如果实参是具有左值的名字或表达式，则传递这个左值本身。
- 如果实参是一个没有左值的表达式（如 $a+b$ 或 2 等），则为其申请一临时数据空间，计算出的表达式的值并存入该单元，然后传递这个存储单元的地址。
- 把实参的左值写入被调用过程活动记录中相应形参的存储单元中。
- 被调用过程执行时，通过形参间接地引用实参。

引用调用示例

```
program addr_ref(input, output);  
  var a, b: integer;  
  procedure exchange(var x, y: integer);  
    var temp: integer;  
    begin  
      temp:=x;  
      x:=y;  
      y:=temp  
    end;  
  begin  
    a:=1; b:=2;  
    exchange(a,b);  
    writeln('a=',a); writeln('b=',b)  
  end.
```



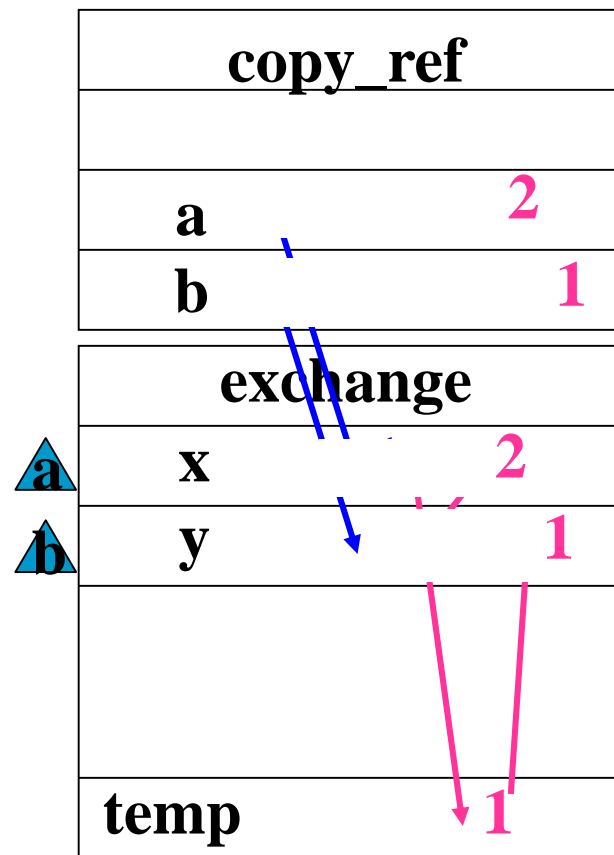
执行结果
a=2
b=1

7.4.3 复制恢复 (copy-restore)

- 传值调用和引用调用的一种混合形式
- copy-in/copy-out
- 复制恢复机制的实现
 - ◆ 过程调用时，调用过程对实参求值，将实参的右值传递给被调用过程，写入其活动记录的参数域中 (copy in)，并记录与形参相应的实参的左值。
 - ◆ 被调用过程执行时，对形参的操作在自己的活动记录参数域空间上进行。
 - ◆ 控制返回时，被调用过程根据所记录的实参的左值把形参的当前右值复制到相应实参的存储空间中 (copy out)。当然，只有具有左值的那些实参的值被复制出来。
- 特征：
 - ◆ 调用时是把实参的右值复制入被调用过程活动记录的形参单元
 - ◆ 返回时是把形参的结果从其活动记录复制出来，复制到调用过程活动记录实参左值确定的单元

复制恢复调用示例

```
program copy_ref(input, output);  
  var a, b: integer;  
  procedure exchange(var x, y: integer);  
    var temp: integer;  
    begin  
      temp:=x;  
      x:=y;  
      y:=temp  
    end;  
  begin  
    a:=1; b:=2;  
    exchange(a,b);  
    writeln('a=',a); writeln('b=',b)  
  end.
```



执行结果

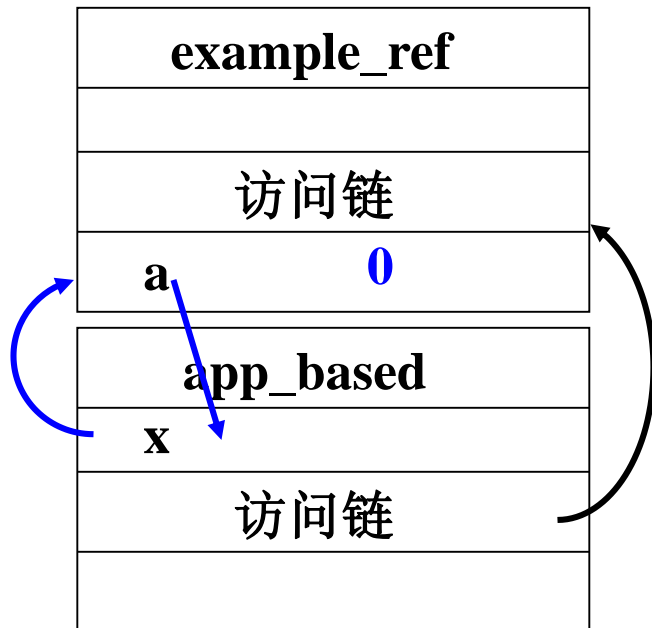
a=2

b=1

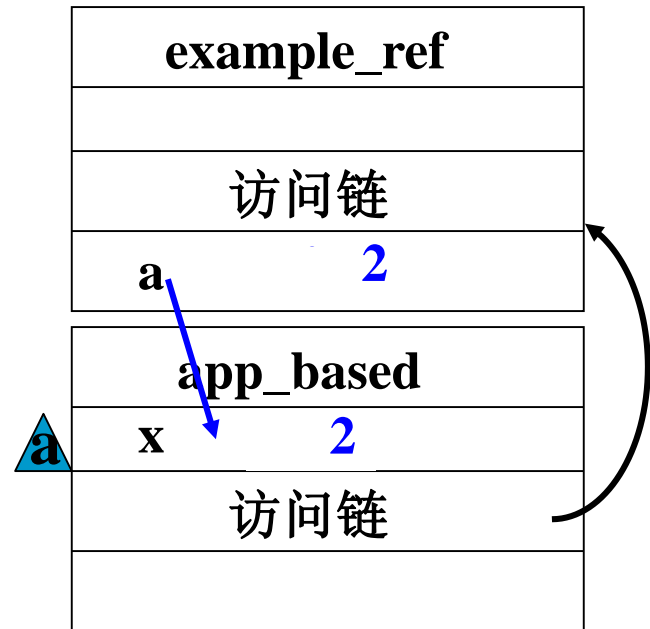
引用调用与复制恢复的区别示例

```
program example_ref(input,output);  
  var a: integer;  
  procedure app_based(var x: integer);  
    begin x:=2; a:=0 end;  
begin a:=1; app_based(a); writeln('a=', a) end.
```

■ 引用调用



■ 复制恢复



复制恢复机制

- 如被调用过程以多种方式访问实参存储单元的话，引用调用和复制恢复的执行结果不一样。
- 复制恢复机制没有规定的问题在不同的语言 and 实现上是有区别的
 - ◆ 按什么顺序把形参的当前值复制回实参？
 - ◆ 实参的地址是不是仅在过程的入口处计算？
 - ◆ 实参的地址需不需要存储？
 - ◆ 实参的地址在过程的出口处需不需要重新计算？

7.4.4 传名调用 (call-by-name)

- Algol 60语言所定义的一种特殊的参数传递方式。
- 传名调用机制
 - ◆ 把过程当作“宏”处理，即在调用出现的地方，用被调用过程的过程体替换调用语句，并用实参的名字替换相应的形参。这种文字替换称为宏扩展。
 - ◆ 用实参的字面形式替换相应的形参
 - ◆ 被调用过程中的局部名字不能与调用过程中的名字重名。
 - 可以考虑在做宏扩展之前，对被调用过程中的每一个名字都系统地重新命名，即给以一个不同的新名字。
 - ◆ 为保持实参的完整性，可以用括号把实参的名字括起来。

传名调用示例

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

打展.

- `exchange(a, b);`
将被展开为:

```
temp:=a;  
a:=b;  
b:=temp;
```

- `exchange(i, a[i]);`
将被展开为:

```
temp:=i;  
i:=a[i];  
a[i]:=temp;
```

局部展开

小结

■ 基本概念

- ◆ 静态文本、活动
- ◆ 活动的生存期、活动树、控制栈、活动记录
- ◆ 运行时内存的划分
- ◆ 声明的作用域、名字的结合

■ 存储分配策略

- ◆ 静态存储分配
- ◆ 栈式存储分配
 - 控制链、访问链
 - 调用序列、返回序列
- ◆ 堆式存储分配

小结 (续)

再执行

■ 非局部名字的访问

- ◆ 静态作用域规则
- ◆ 非嵌套过程的作用域规则、存储分配策略
- ◆ 嵌套过程的作用域规则
 - 嵌套深度
 - 访问链的指向
 - 访问链的使用
 - 访问链的建立

小结 (续)

■ 参数传递

- ◆ 传值调用
- ◆ 引用调用
- ◆ 复制恢复
 - 引用调用与复制恢复的不同
- ◆ 传名调用