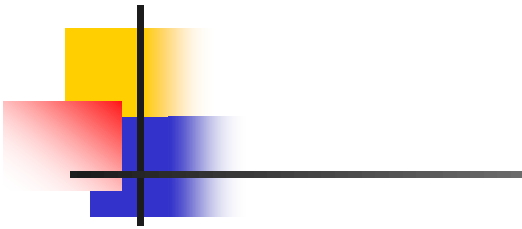
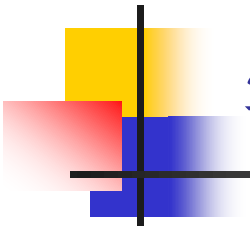


第七章 运行时环境



学习内容

- 源语言语义
- 存储组织
- 存储分配策略
- 对非局部名字的访问
- 参数传递
- 符号表
- 动态内存分配

7.1 源语言相关问题

- 7.1.1 过程（函数），procedure——静态
  活动，activation——动态
- 过程定义，procedure definition
- 过程名，procedure name
- 过程体，procedure body
- 函数，function
- 调用，call
- 形式参数，formal parameters
- 实际参数，actual parameters

Pascal源码示例

```
program sort(input, output);
    var a : array [0..10] of integer;
    procedure readarray;
        var i : integer;
        begin
            for i := 1 to 9 do read(a[i]);
        end;
    function partition(y, z : integer) : integer;
        var i, j, x, v : integer;
        begin …
        end;
```

过程名

过程定义

过程体

函数定义

形式参数

Pascal源码示例（续）

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

调用

实际参数

Pascal源码示例（续）

```
begin
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1, 9);
end.
```

7.1.2 活动树（activation tree）

- 控制流
  - 顺序性：程序的执行→一组操作步骤序列，在每个步骤，控制流开始于程序的特定位置
  - 过程的执行：起始于过程体开始，最终控制权返回到过程调用点之后→活动树表示
- 活动：一次执行（execution）
- 过程P的生存期，lifetime：
  - 过程体执行的第一步操作到最后一步操作的操作序列，包括P调用其他过程的时间

过程的活动

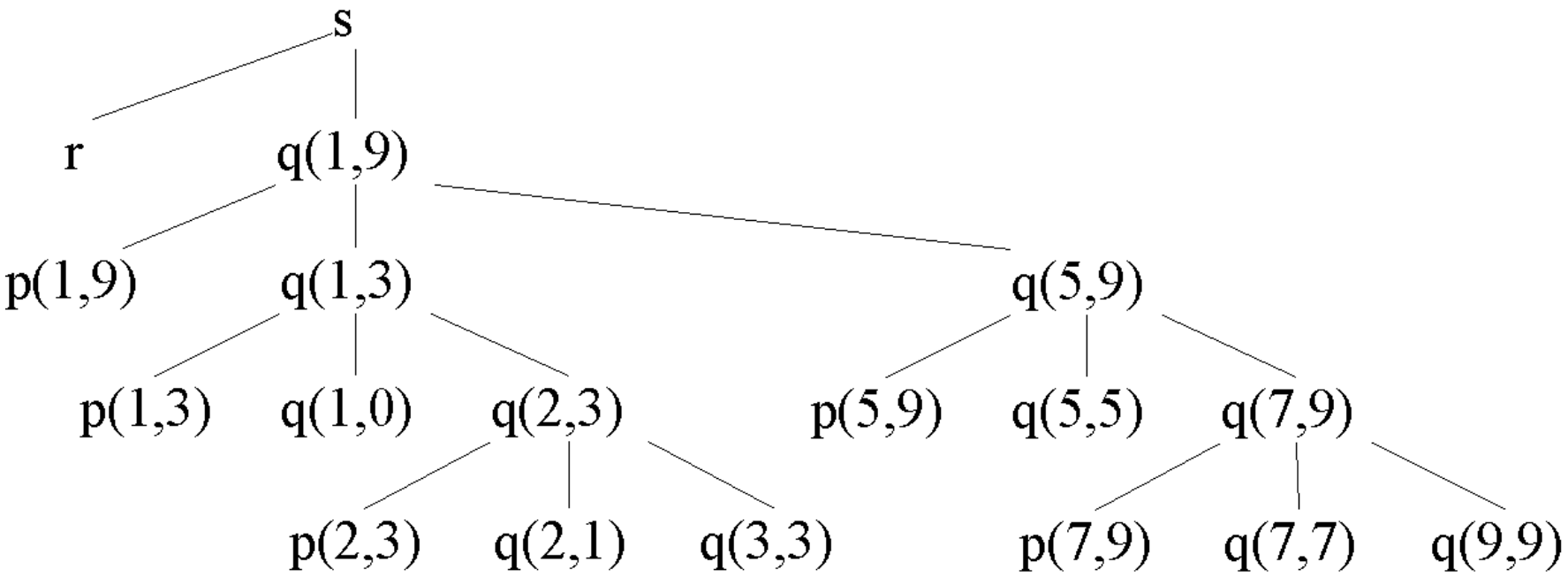
- p调用q，控制最终会返回p——每次控制流从p转移到q，总会返回到p
- 过程的生命期或者不重叠、或者嵌套——b在a退出之前进入，则必在a之前退出
  - 利用输出语句演示嵌套

execution begin…
enter readarray
leave readarray
enter quicksort(1, 9)
enter partition(1, 9)
leave partition(1, 9)
enter quicksort(1, 3)
…
leave quicksort(1, 3)
enter quicksort(5, 9)
…
leave quicksort(5, 9)
leave quicksort(1, 9)
execution terminated.

递归

- 过程p的某个活动尚未结束，可以开始它的一个新的活动
- 间接递归
- 活动树
  1. 结点——过程的活动
  2. 根结点——主程序的活动
  3. a是b的父结点←→控制流从a到b
  4. a在b的左边←→a的生存期在b之前

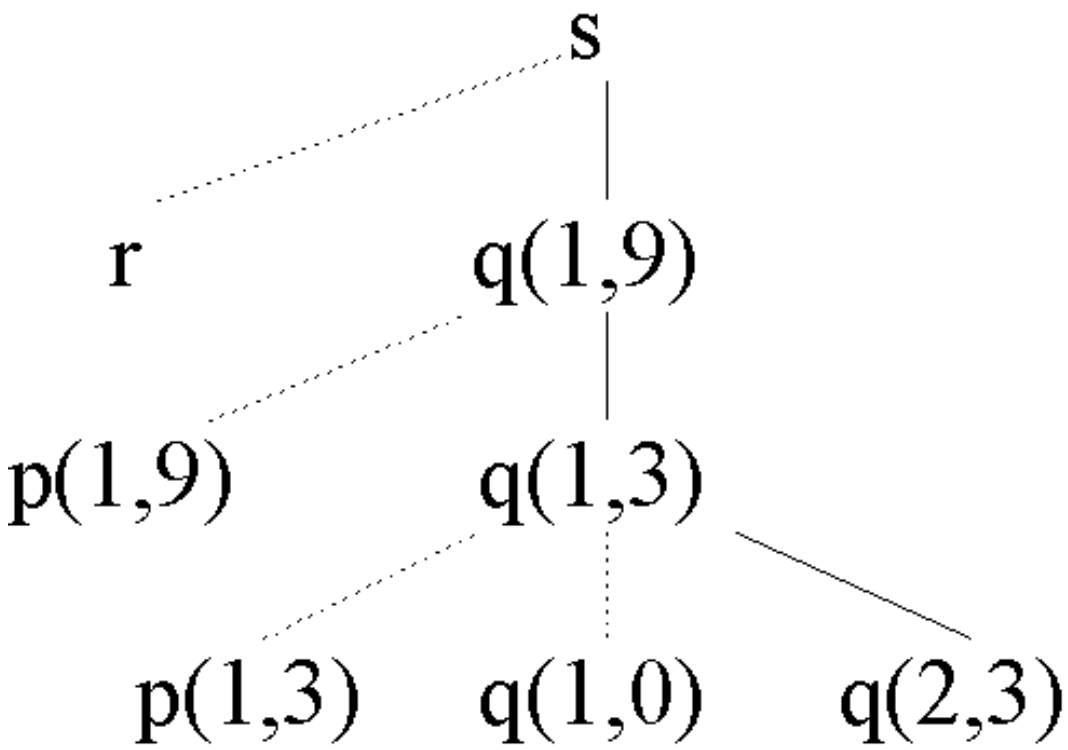
例7.1

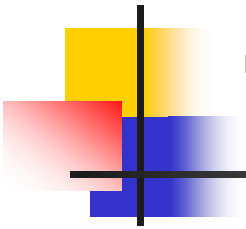

s
r
q(1,9)
p(1,9)
q(1,3)
q(5,9)
p(1,3)
q(1,0)
q(2,3)
p(5,9)
q(5,5)
q(7,9)
p(2,3)
q(2,1)
q(3,3)
p(7,9)
q(7,7)
q(9,9)

7.1.3 控制栈（control stack）

- 活动树深度优先遍历→控制流变化
- 控制栈
  - 当活动开始，对应结点入栈
  - 结束后，将其弹出
  - 当前栈内容——到根结点的路径

例7.2



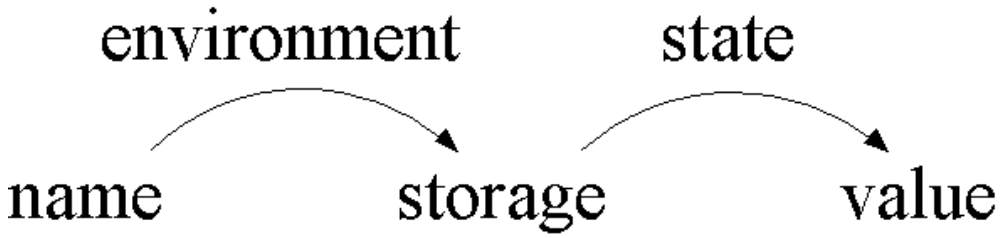


7.1.4 作用域

- 声明（declaration）：规定名字的含义
- 显式声明、隐式声明（Fortran）
- 同一名字在程序不同位置的不同声明
- 作用域（scope）：声明起作用的范围
  - 局部的（local）、非局部的（nonlocal）
- 作用域规则
  - 确定声明起作用的范围
  - 名字对应哪个声明
- 利用符号表

7.1.5 名字的绑定（binding）

- 名字只声明一次，运行时也可能指向不同数据对象
- 环境（environment）：函数
  - 名字→存储位置，名字→l-value
- 状态（state）：函数
  - 存储位置→存储数据，l-value→r-value



环境和状态

- 赋值语句改变状态，但不改变环境
- 对某个环境，若名字x对应存储位置s→x绑定到s，s是x的一个绑定
- 绑定——声明的动态概念

静态概念	动态概念
过程的定义	过程的活动
名字的声明	名字的绑定
声明的作用域	绑定的生存期

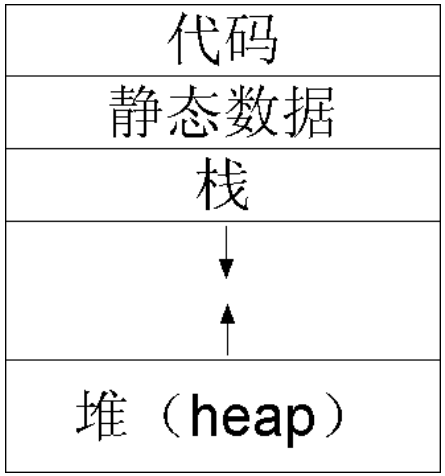
- Pascal过程的局部名字，不同活动→不同存储位置

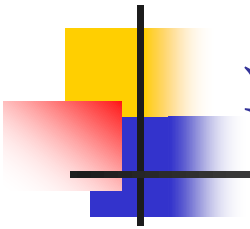
7.1.6 影响存储组织的几个问题

1. 过程可否递归？
2. 过程活动结束时，局部名字值如何处理
3. 过程可否访问非局部名字？
4. 过程被调用时，参数如何传递
5. 过程可否作为参数
6. 过程可否作为返回类型
7. 程序可否动态控制存储分配
8. 存储是否必须显式释放？

7.2 存储组织

- 7.2.1 运行时存储细化
  - 存储目标代码——大小固定
  - 存储数据对象——部分固定
  - 控制栈——保存过程的活动信息

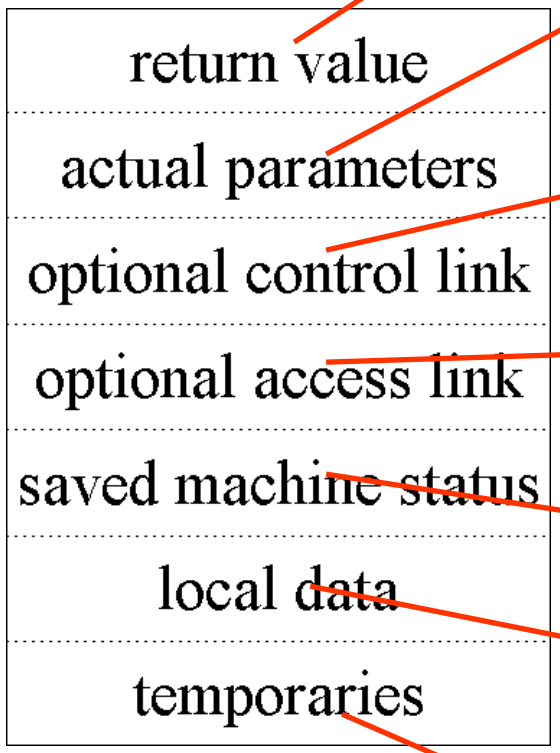




活动信息的保存

- 过程调用
  - 当前过程活动终止
  - 状态信息（寄存器、PC）→控制栈
- 调用返回
  - 从控制栈恢复状态信息
  - 停止的过程活动重新开始
- 局部数据对象→控制栈
- **堆**
  - 动态内存分配
  - 也可能保存活动信息

7.2.2 活动记录(activation record)



存放被调用函数传递给调用函数的返回结果

调用函数向被调用函数传递实际参数的区域

调用函数（父函数）活动记录的指针

用来访问那些保存在其他
活动记录中的非局部数据

函数调用前时刻机器状态相关的
一些信息：程序计数器、寄存器...

保存局部数据

保存临时数据：表达式计算中间结果...

7.2.3 局部数据布局

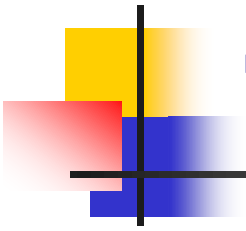
- 例7.3
- 类型→占用空间
- 对齐（align），补丁（padding），压缩（pack）

类型	大小（位）		对齐（位）	
	机器1	机器2	机器1	机器2
char	8	8	8	64
short	16	24	16	64

- char a; short b;

  机器1：4字节，a之后1字节补丁

  机器2：16字节



7.3 内存分配策略

1. 静态分配（static）
   - 在编译时（运行之前）确定所有数据的内存分布
2. 栈分配（stack）
   - 利用栈管理运行时存储
3. 堆分配（heap）
   - 允许用户在运行时动态地在堆之上分配、释放内存

7.3.1 静态分配策略

- 局限性
  1. 数据对象的大小和对其内存位置的限制，必须在编译时已知
  2. 不允许递归←
     对过程的多次活动，局部名字都绑定到相同的存储位置——局部名字的值在多次调用间会得到保持（retain）
  3. 不允许动态数据分配

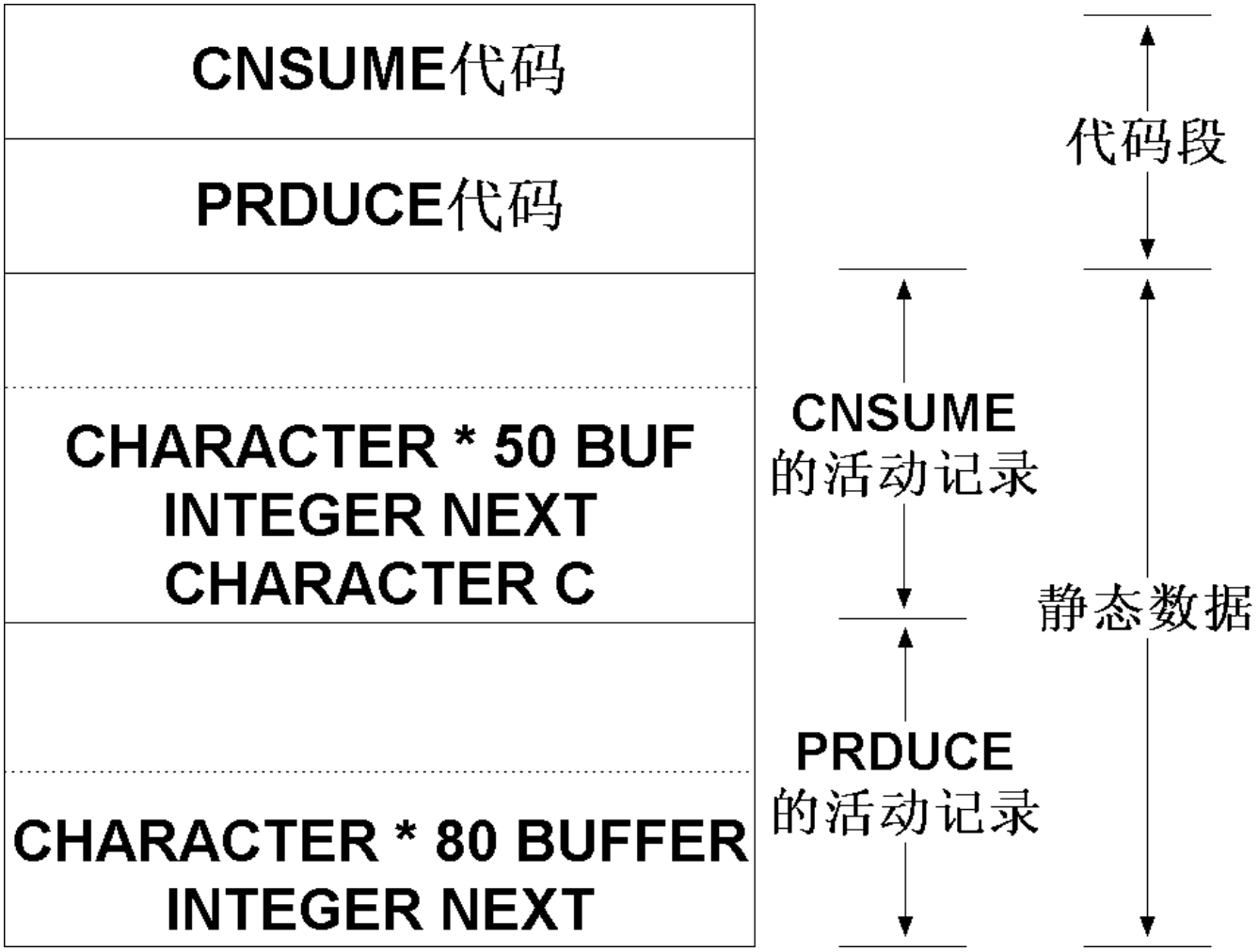
例7.4 Fortran的静态分配


```
PROGRAM CNSUN
    CHARACTER * 50 BUF
    INTEGER NEXT
    CHARACTER C, PRDUCE
    DATA NEXT /1/,  BUF /‘ ’/
6       C = PRDUCE()
        BUF(NEXT:NEXT) = C
        NEXT = NEXT + 1
        IF (C .NE. ‘ ’) GOTO 6
        WRITE (*, ‘(A)’) BUF
        END
```

例7.4 Fortran的静态分配（续）


```
CHARACTER FUNCTION PRDUCE()
    CHARACTER * 80 BUFFER
    INTEGER NEXT
    SAVE BUFFER, NEXT
    DATA NEXT /81/
        IF (NEXT .GT. 80) THEN
            READ (*, ‘(A)’) BUFFER
            NEXT = 1
        END IF
        PRDUCE = BUFFER(NEXT:NEXT)
        NEXT = NEXT + 1
        END
```

例7.4 Fortran的静态分配（续）



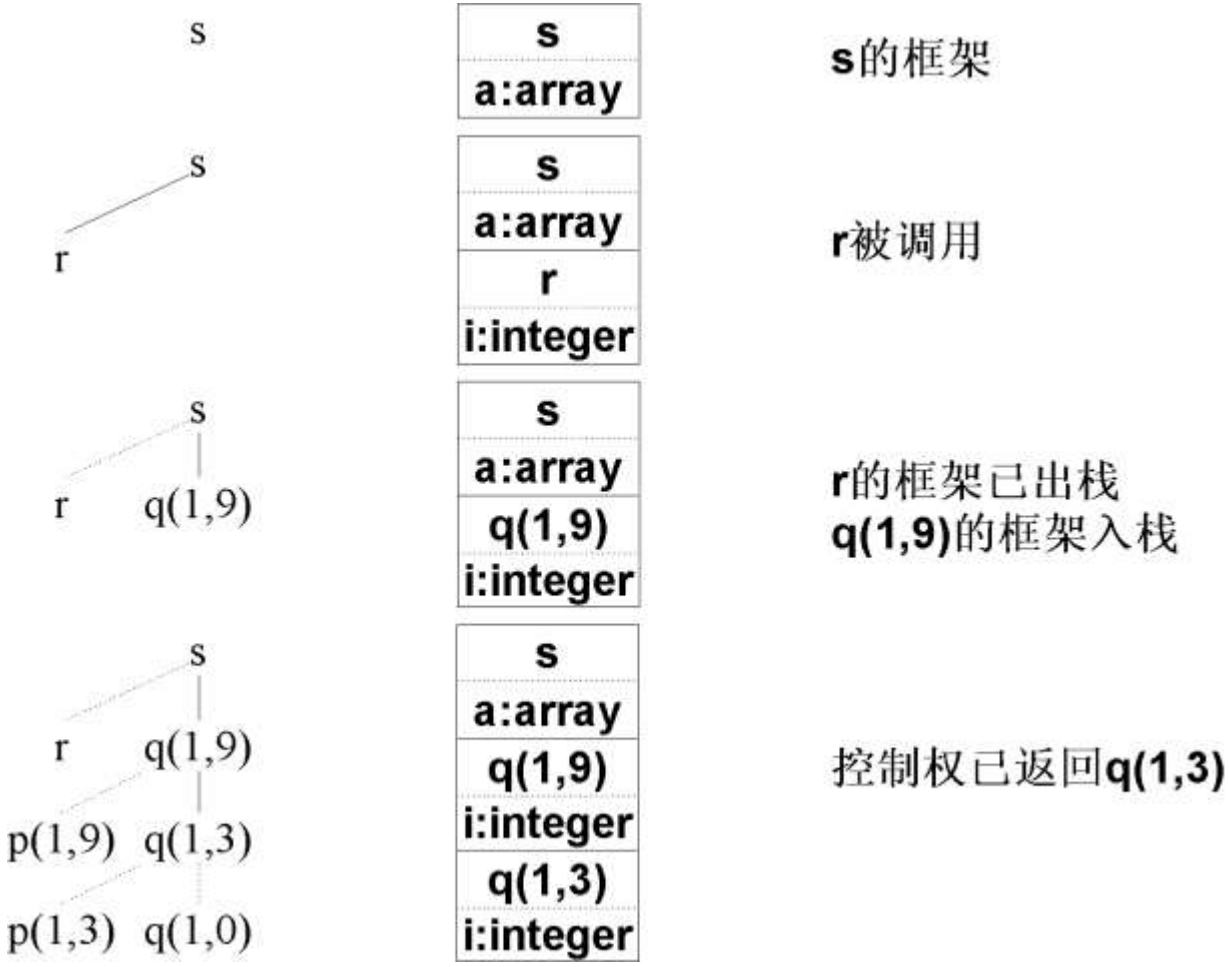
例7.4 Fortran的静态分配（续）

- 用户输入“hello world”
- PRDUCE
  - 常态：缓冲区字符→CNSUME，NEXT加1
  - 缓冲区用完（NEXT>80），重新读取用户输入
- 第一次调用：NEXT=81，用户输入→BUFFER，NEXT←1，‘h’→CNSUME，NEXT加1
- 第二次，BUFFER未变，NEXT未变=2，‘e’→CNSUME，NEXT加1
- …

7.3.2 栈分配策略

- 过程调用——push 活动记录
- 过程返回——pop 活动记录
- 每次过程活动，局部名字都绑定到新的位置
- 过程结束，局部名字空间被释放——值丢失

例7.5 快速排序执行过程



调用序列（calling sequence）

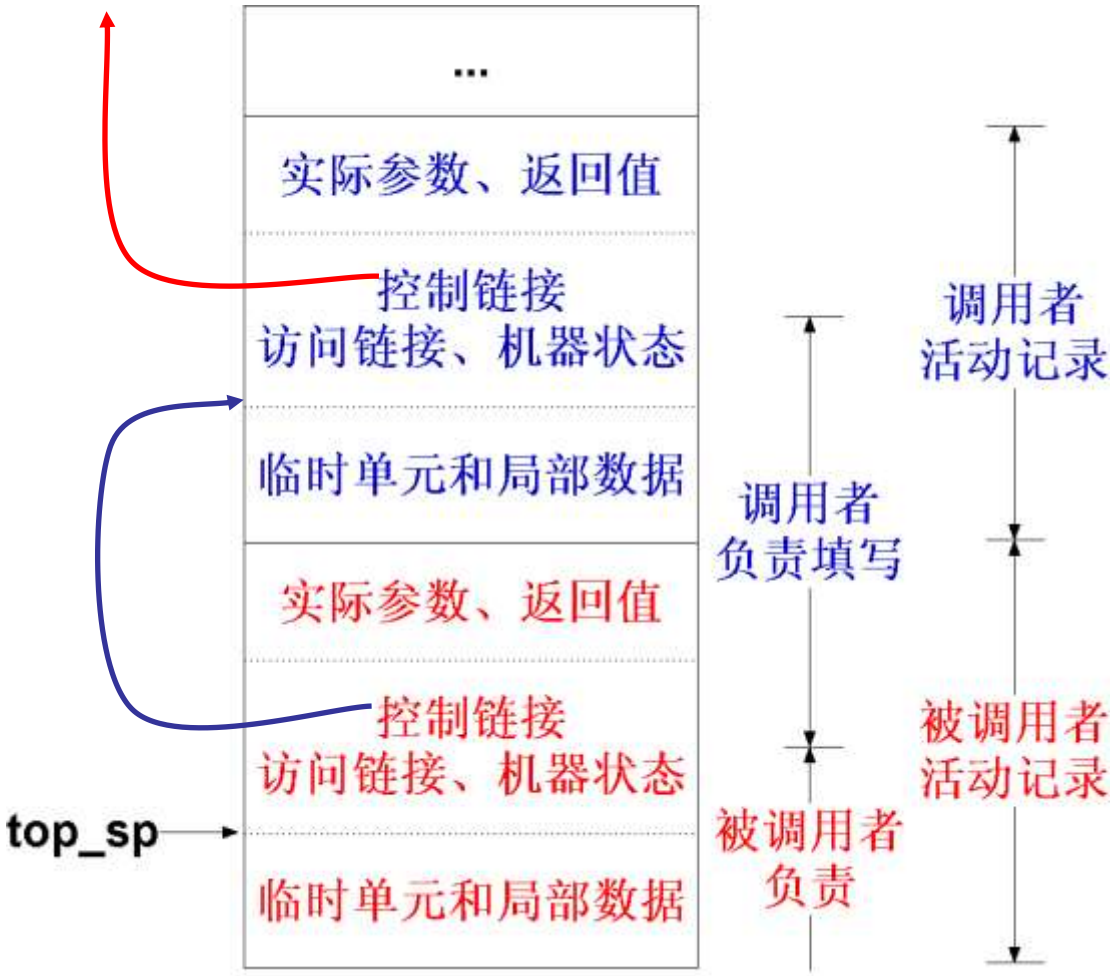
- 创建活动记录，填写相关信息的代码
- 返回序列，return sequence
- 划分到调用函数和被调用函数两部分
- 较早确定大小的信息放在活动记录中部
  - 控制链接、访问链接、机器状态信息
  - 临时单元
  - 实际参数、返回值
  - 局部数据，Pascal局部数组

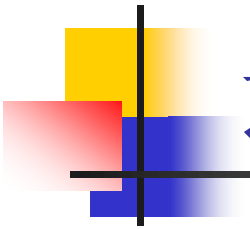
大小确定 早↓晚

调用序列（续）

1. 调用者计算实际参数
2. 调用者存储返回地址（被调用函数返回后继续运行的代码位置）和top_sp旧值，然后将top_sp调整到上图位置——要跨过调用者的局部数据和临时单元以及被调用者的实际参数和状态区
3. 被调用者保存寄存器值和其他状态信息
4. 被调用者初始化局部数据开始运行

调用序列（续）

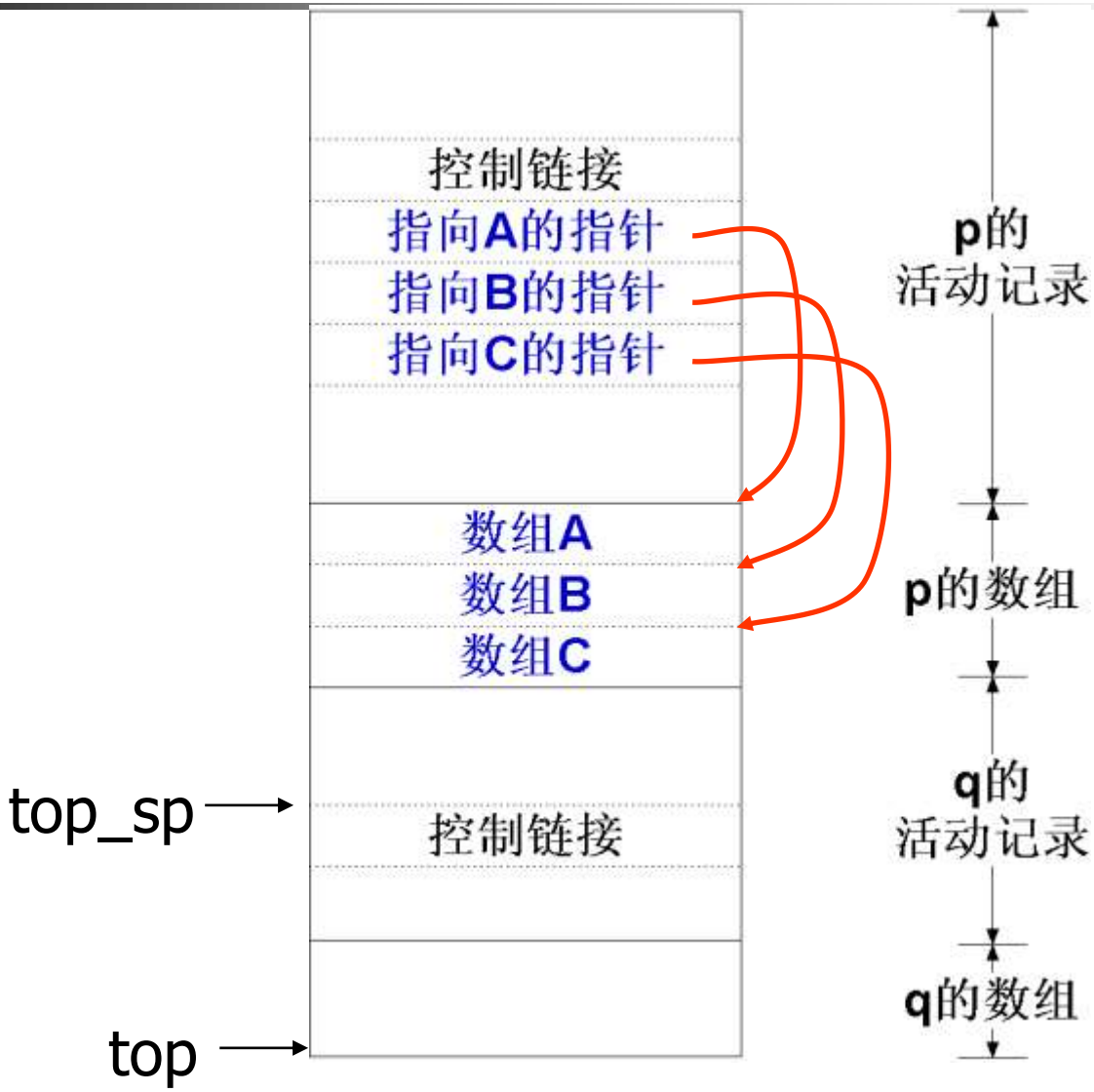

...
实际参数、返回值
控制链接
访问链接、机器状态
临时单元和局部数据
实际参数、返回值
控制链接
访问链接、机器状态
top_sp
临时单元和局部数据
调用者
负责填写
被调用者
负责
调用者
活动记录
被调用者
活动记录



返回序列

1. 被调用者设置返回值（位于栈底，与调用者活动记录相邻）
2. 利用保存的状态信息，被调用者恢复top_sp和其他寄存器，然后跳转到返回地址
3. 调用者复制返回值

可变长度数据


控制链接
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
控制链接
top_sp
top
p的
活动记录
p的数组
q的
活动记录
q的数组

7.3.3 空悬（dangling）引用

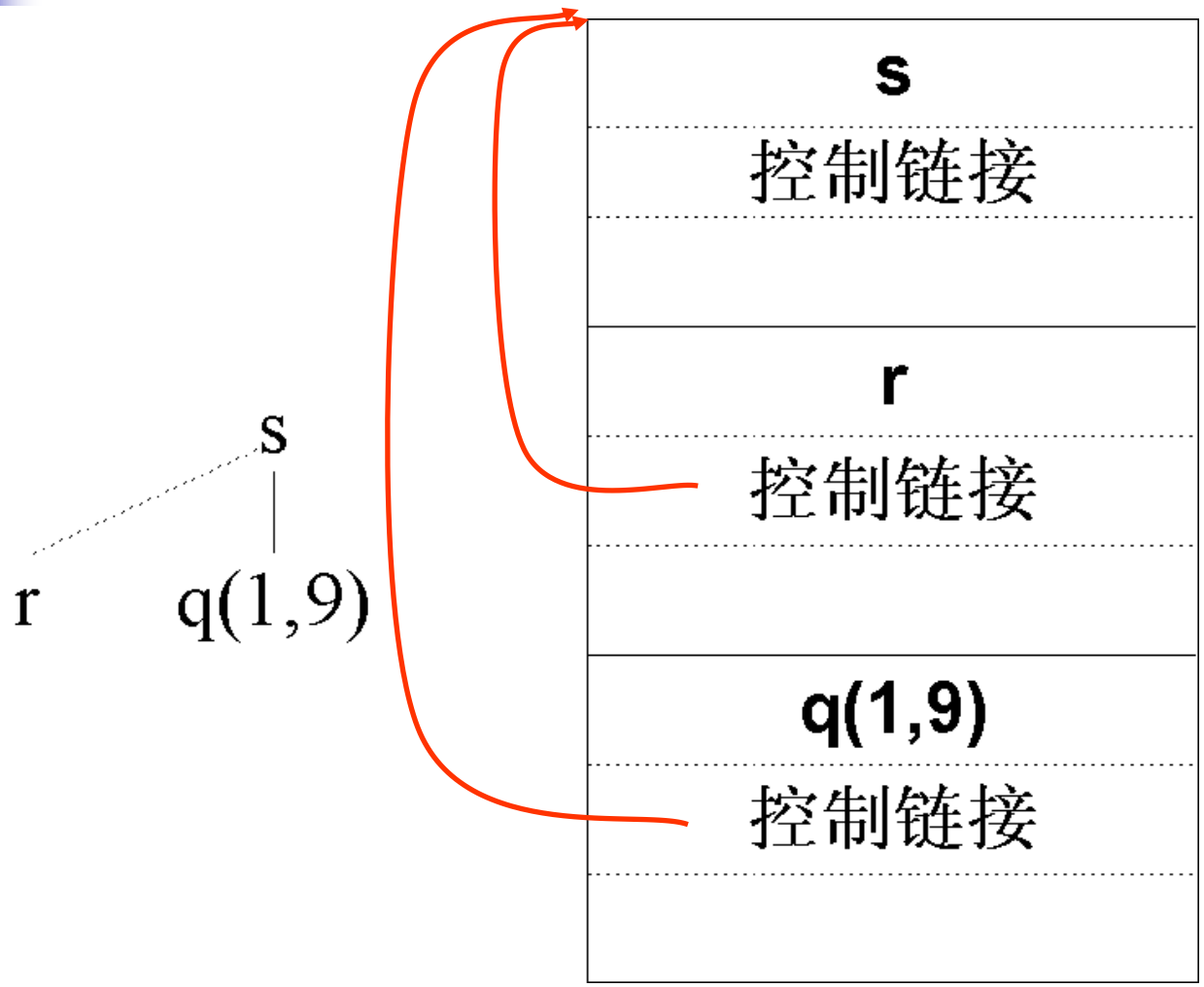
- 例7.6：
- 内存空间已释放，但还存在指向它的引用——程序员的错误！

```
int *dangle()
{
    int i = 23;
    return &i;
}
main()
{
    int *p;
    p = dangle();
}
```

7.3.4 堆分配策略

- 栈分配策略的局限，不能处理下列情况
  - 局部名字的值在活动结束后能够保持
  - 被调用函数的活动生存期比调用者长——可用活动树表示控制流的语言无法实现
- 对小的（或大小已知的）活动记录
  1. 为若干可能大小，创建空闲块链表（静态）
  2. 为大小为s的活动记录分配空间，尽量使用s’的空闲块表，s’为大于等于s的最小值。空间释放时，将块退回空闲块链表
  3. 对大块内存分配，利用堆管理器

堆分配策略（续）



r的活动记录得到保持

7.4 访问非局部名字

- 栈
- 词法/静态作用域规则

  lexical-，static-scope rule

  C、Pascal、Ada

  嵌套问题
- 动态作用域规则

  dynamic-scope rule

  Lisp、APL、Snobol

7.4.1 程序块（block）

- 语句块＋它的局部数据声明
  { 声明　语句}
- 特性：嵌套结构、不会交叉
- “最近嵌套规则”，most closely nested rule
  - 对于块B中的声明，其作用域包括B
  - 块B中未声明名字x，则B中对x的引用由某个外层块B’中x的声明所确定，B’满足：
    - 在包围B的，且有x的声明的外层块中，B’是距离B最近的那个

嵌套块作用域示例

```
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);
}
```

B0 B1 B2 B3

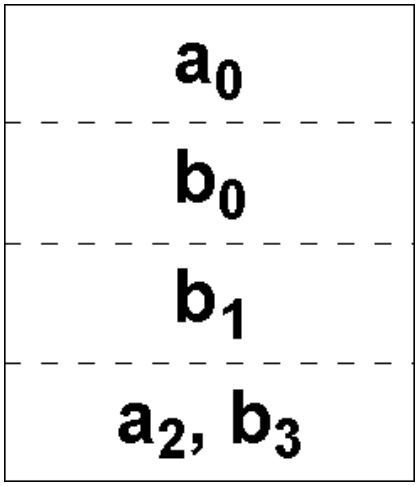
声明	作用域
int a = 0;	B_0-B_2
int b = 0;	B_0-B_1
int b = 1;	B_1-B_3
int a = 2;	B_2
int b = 3;	B_3

运行结果

2	1
0	3
0	1
0	0

实现方法

- 栈实现，将块看作过程，进入块时为局部变量分配空间，退出时释放
- 为过程中所有块一起分配内存



不在上机大作业范围内

7.4.2 无过程嵌套的静态作用域

- C
- 局部名字——栈
- 任何过程之外的名字——静态分配
- 非局部——全局概念
- 过程可作为参数和返回值←非局部名字都可用静态地址访问到

过程作为参数——无嵌套情况

```
program pass(input, output);
    var m : integer;
    function f(n : integer) : integer;
        begin   f := m + n end;
    function g(n : integer) : integer;
        begin   g := m * n end;
    procedure b(function h(n : integer) : integer);
        begin   write(h(2)) end;
    begin
        m := 0;
        b(f); b(g); writeln
    end.
```

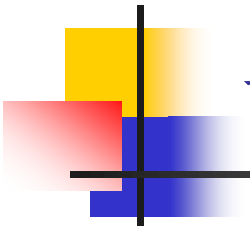
运行结果：

2　　0

7.4.3 包含过程嵌套的静态作用域

- Pascal，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 … 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…  end {partition};
            begin … end {quicksort};
    begin … end {sort}.
```

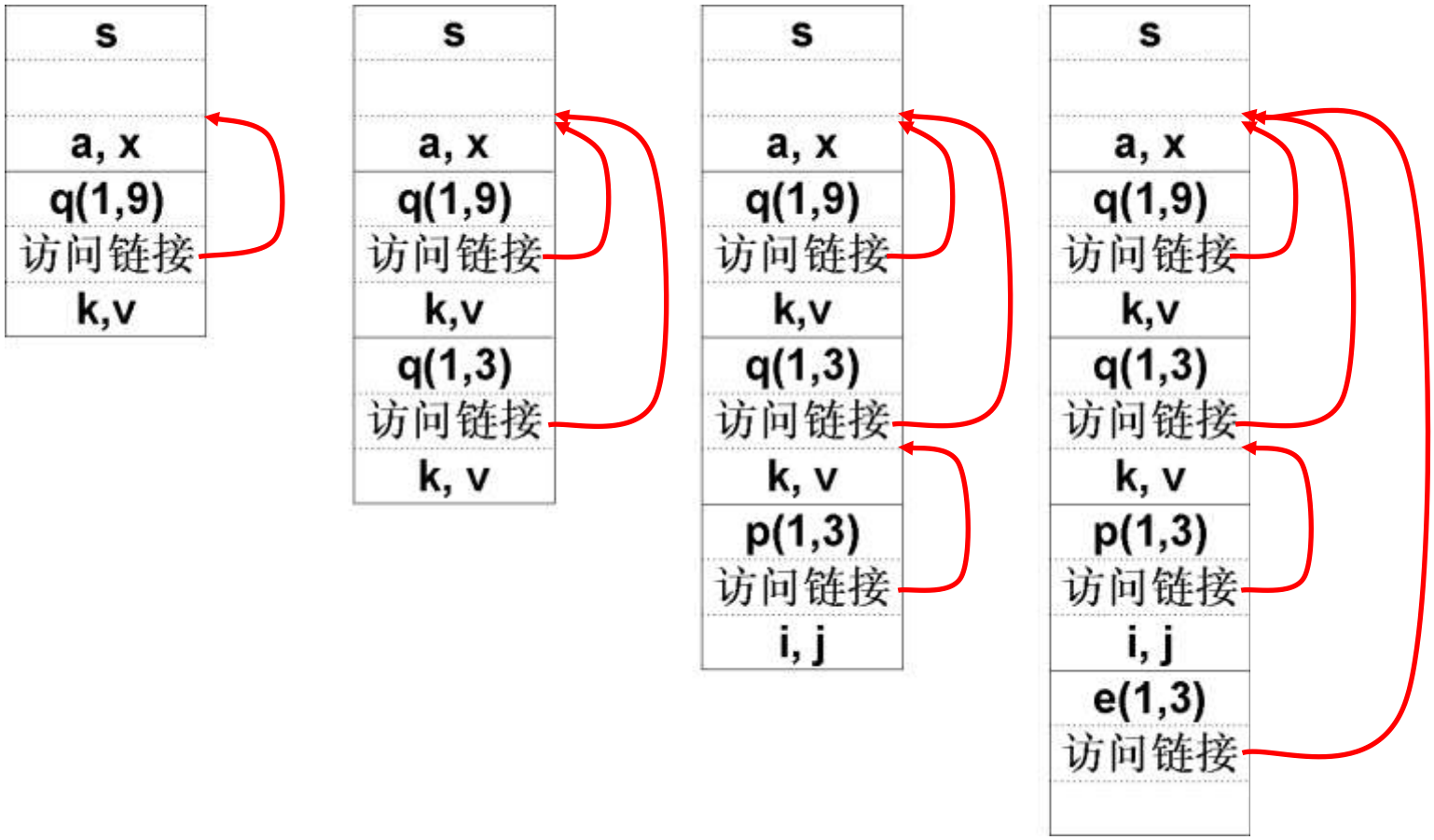


嵌套深度（nesting depth）

sort	1
readarray	2
exchange	2
quicksort	2
partition	3

访问链接

- p直接嵌套在q内→p的访问链接指向q的最近一次活动记录内的访问链接



使用访问链接

- 过程p（嵌套深度n_p）访问非局部数据a（嵌套深度n_a），$n_a <= n_p$，访问方法为：
  1. 当控制流到达p，其活动记录位于栈顶。从它的访问链接开始，遍历$n_p - n_a$（此值在编译阶段即可获得）个访问链接
  2. 此时即可到达a所在过程的活动记录，由其访问链接地址即可得到a的地址

设置访问链接

- 调用序列代码的一部分，假定过程p（嵌套深度n_p）调用过程x（嵌套深度n_x）

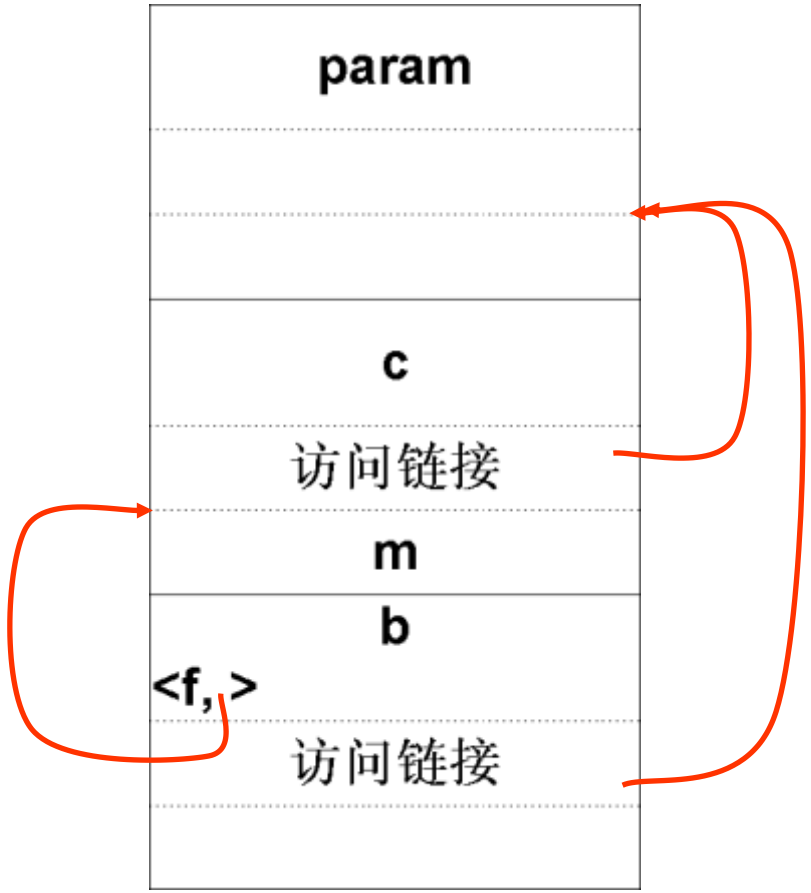
1. $n_p<n_x$
   ➔x在p内部声明，应将x的访问链接指向p的访问链接
2. $n_p>=n_x$
   ➔当前包含p和x，深度为1, 2, …, n_x-1的过程必然是相同
   从p开始，遍历n_p-n_x+1个访问链接，到达同时包含p、x的最近的过程的活动记录
   x的访问链接应指向此记录的访问链接

过程作为参数的情况

```
program param(input, output);
    procedure b(function h(n : integer) : integer);
            begin writeln(h(2)) end { b };
    procedure c;
            var m : integer;
            function f(n : integer) : integer;
                    begin f := m + n end { f };
            begin m := 0; b(f) end { c };
    begin
            c
    end.
```

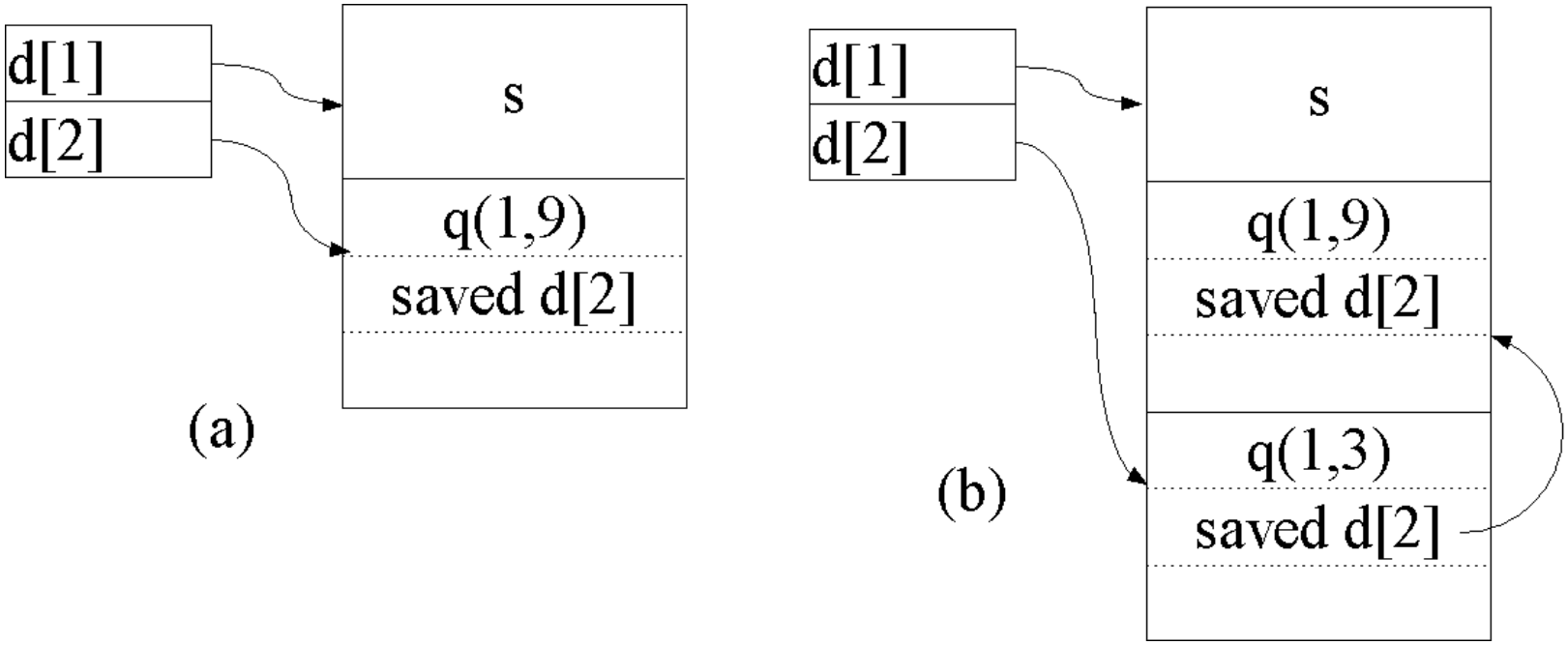
过程作为参数的情况（续）

- 当过程作为参数传递时，对应的访问链接也一起传递

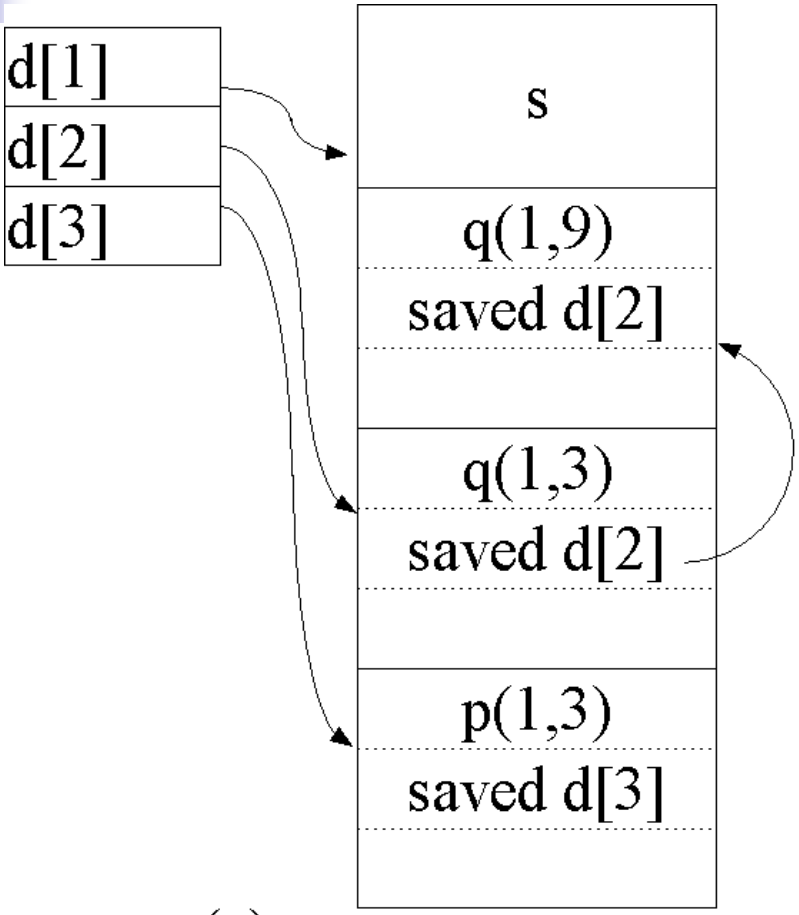


Display表

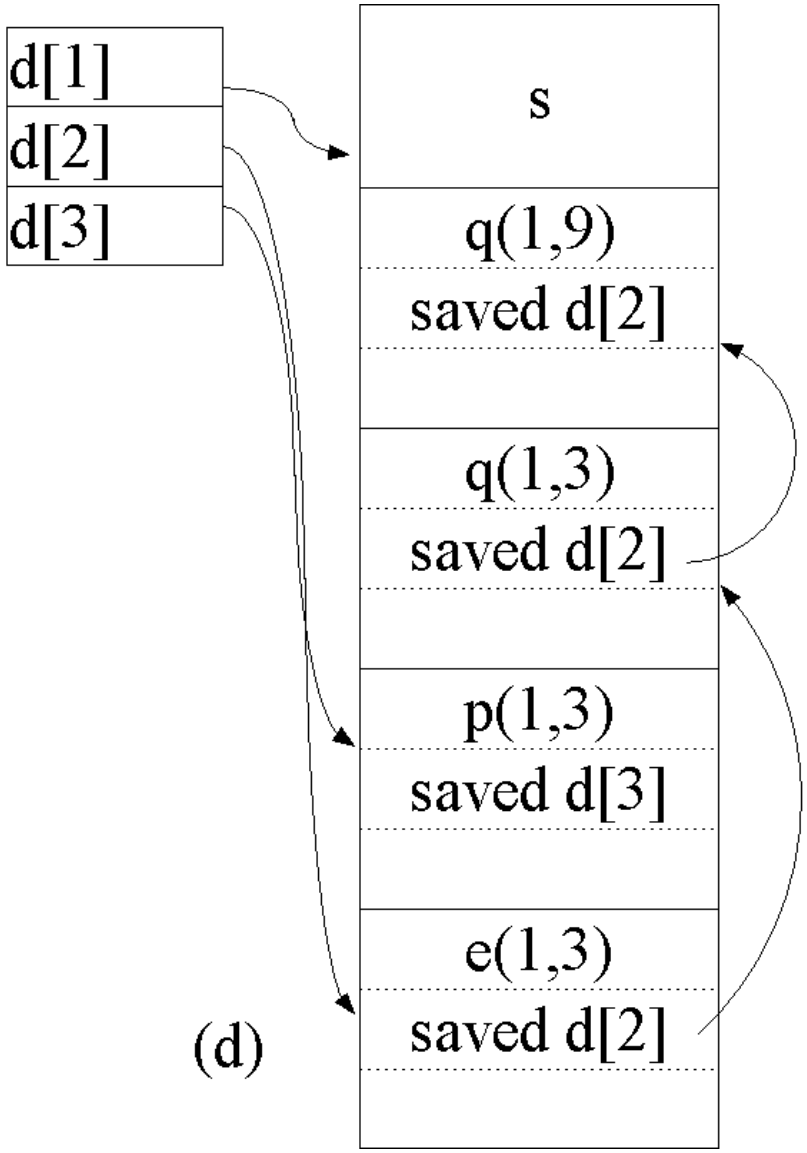
- 一维数组d[i]：将嵌套深度为i的过程的访问链接串起来



Display表（续）



(c)



(d)

display表的设置

- 当设置嵌套深度为i的过程的活动记录
  - 旧的d[i]值保存入新的活动记录
  - 设置d[i]指向新的活动记录
- 正确性：假定嵌套深度j的过程调用深度i的过程
  - j<i：➔i=j+1，d[1]~d[j]无需改变，d[i]指向新活动记录，上面(a)图
  - j>=i：➔包含两个过程，嵌套深度1,2,…,i-1的过程必然是相同的，将旧d[i]保存到新活动记录，并设置它指向新活动记录即可，上面(d)图

7.4.4 动态作用域

- ~~嵌套关系~~，调用关系决定名字绑定

```
program dynamic(input, output);
    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.
```

静态绑定结果

0.250 0.250
0.250 0.250

动态绑定结果

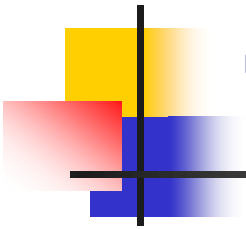
0.250 0.125
0.250 0.125

此处调用show
r绑定到

此处调用show
r绑定到

实现方法

1. 深入访问，deep access
   - ~~访问链接~~，控制链接
   - “深入”控制栈
2. 浅访问，shallow access
   - 名字的当前绑定值保存在静态区域
   - 过程p新的活动开始，它的局部名字n接管为n分配的静态区域
   - n的原值可保存在p的活动记录中，当活动结束时恢复



7.5 参数传递

- 7.5.1 传值方式，call-by-value
  - 形参与局部名字同样处理→存储位置在被调用函数的活动记录中
  - 调用函数计算实参值，将其右值放置到形参的存储位置
  - 对形参的操作不会影响实参的值

传值方式例

```
void swap(int x, int y)
{   int temp;
    temp = x; x = y; y = temp;  }
void main()
{   int a = 1, b = 2;
    swap(a, b);
    printf("a is now %d, b is
            now %d\n", a, b); }
```

结果

a is now 1, b is now 2

调用者 存储空间
…
a：1
b：2
…

swap 存储空间
…
x：2
y：1
temp：1
…

通过传递指针改变实参值

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x; *x = *y; *y = temp;
}
void main()
{
    int a = 1, b = 2;
    swap(&a, &b);
    printf("a is now %d, b is now %d\n", a, b);
}
```

显式传递地址

结果

a is now 2, b is now 1

7.5.2 传地址方式

- call-by-reference，call-by-address，call-by-location，隐含传递地址
  - 实参是名字或表达式，且具有左值，则其左值将被传递给形参
  - 实参无左值，如a+b或2，则为其分配新的空间，将此空间的地址传递给形参
  - 形参与实参——同一个数据对象
  - 引用形参——用传递来的指针进行间接引用

例7.7

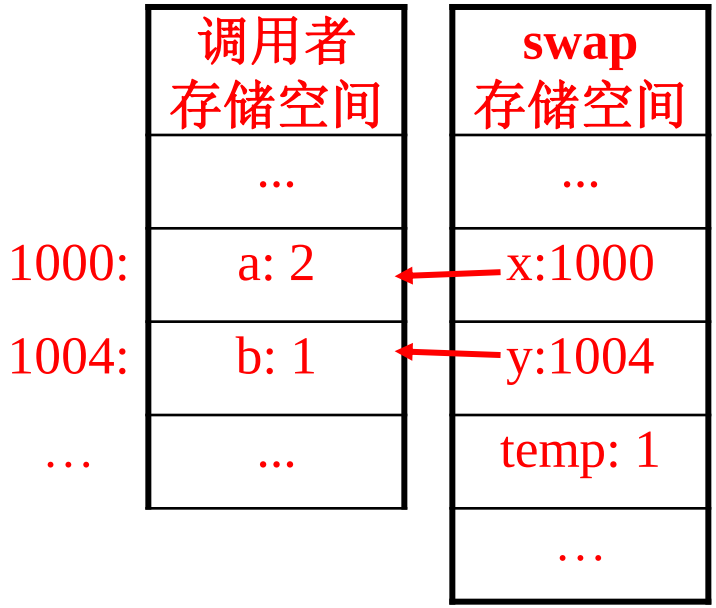
```
program reference(input, output)
var a, b : integer;
procedure swap(var x, y : integer);
    var temp : integer;
    begin
        temp := x; x := y; y := temp;
    end;
begin
    a := 1; b := 2;
    swap(a, b);
    writeln('a =', a); writeln('b =', b);
end.
```

结果

a = 2

b = 1

传地址方式



例7.7（续）

- 若调用swap(i, a[i])
  1. 将i和a[i]的左值复制到swap的活动记录中，假定保存在arg1和arg2两个位置——分别对应形参x和y
  2. temp:=x——将arg1指向位置的值（i的初值，假定为I_0）赋予temp
  3. x:=y——将arg2指向位置的值赋予arg1指向的位置——i := a[I_0]
  4. y:=temp——将temp的值赋予arg2指向的位置——a[I_0]:=I_0

7.5.3 复制—恢复方式

- copy-restore，传值和传地址的混合，copy-in，value-result
  1. 进入被调用函数之前，计算实参值，将其右值传送到被调用函数（传值）。另外，若实参有左值的话，还要求出左值
  2. 当调用返回时，将形参当前的右值复制回实参左值指向位置。

例

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

传地址方式

a、x指向相同内存地址，先被赋值为2，然后赋值为0，因此最终a的值为0

复制—恢复方式

a、x指向不同内存地址，unsafe结束后，a=0，x=2，此时进行恢复，x→a，最终a的值为2

7.5.4 传名方式(call-by-name)

- Algol，拷贝规则，copy-rule
  1. 过程像宏一样处理
     - 被调用——用过程体程序文本替换过程名
     - 实参的程序文本替换形参
     - 宏扩展，marco-expansion
       内联扩展，in-line expansion
  2. 被调用过程内的局部名字保证与调用过程中的名字不混淆——可认为这些局部名字在宏扩展前改名为独一无二的名字
  3. 实参可能被加以括号，以保证完整性

例7.8

…

swap(i, a[i]);

…

- 在传名方式下，实际上相当于如下代码

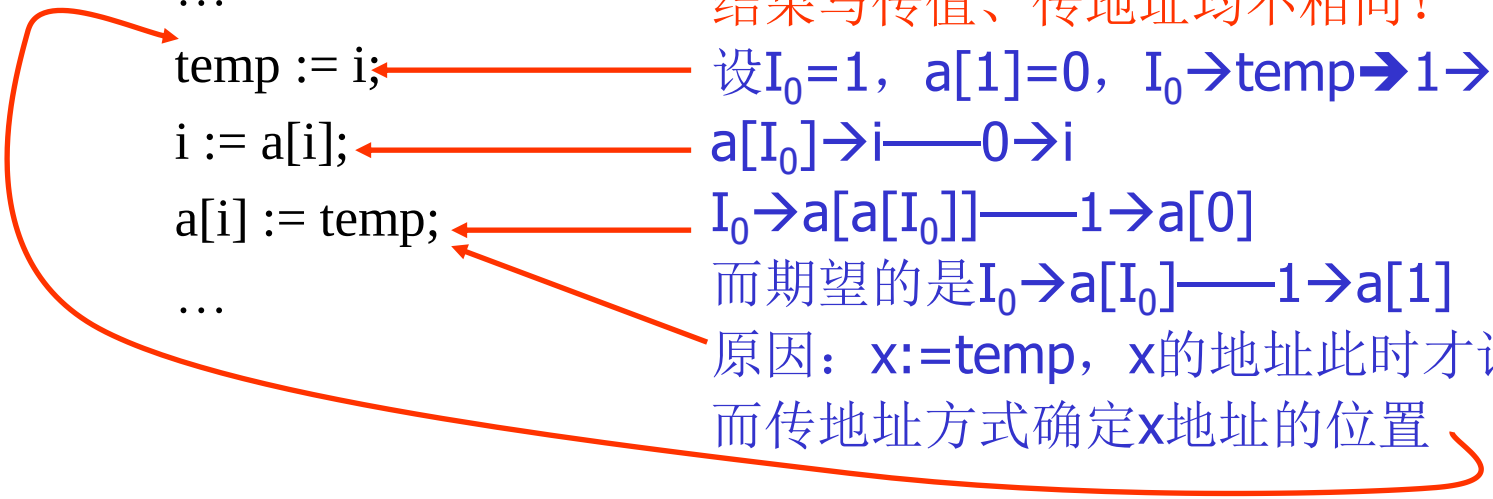
…

temp := i;

i := a[i];

a[i] := temp;

…



结果与传值、传地址均不相同！

设I_0=1，a[1]=0，I_0→temp➔1→temp

a[I_0]→i——0→i

I_0→a[a[I_0]]——1→a[0]

而期望的是I_0→a[I_0]——1→a[1]

原因：x:=temp，x的地址此时才计算，

而传地址方式确定x地址的位置

例7.9 内联函数

x := f(A) + f(B);

➔

t_1 := A;

t_2 := B;

t_3 := f(t_1);

t_4 := f(t_2);

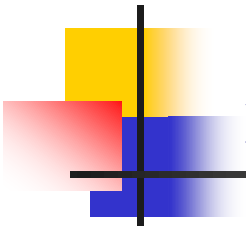
x := t_3 + t_4;

7.6 符号表

- 保存作用域和绑定信息
- 基本操作
  - 添加新标识符——insert
  - 查找标识符——lookup
- 符号表表项——记录
  - 一致性
  - 某些信息保存在外部，用指针指向
  - 两个主要组成部分
    - 名字：定长字符串；统一保存，指针指向
    - 属性

实现方式一：线性表

- 数组方式
  - 固定大小，可能浪费，也可能不足
- 链表方式
  - 动态内存分配，根据输入大小确定
- 时间复杂度
  - 查找操作：与表大小线性关系
  - 插入操作
    - 一个名字允许多个表项（不检查）：常数时间
    - 检查重复表项：线性
  - n次插入，e次查找：$cn(n+e)$，$O(n^2)$



实现方式二：hash表

- 开地址法：“链表数组”
  - 长度m的数组，每个元素为指向一个符号表项链表的指针
  - hash函数：H(名字)→{0, 1, …, m-1}
  - H(名字)=i→名字存入第i个链表
  - 若hash函数设计良好，每个链表长度n/m
  - n次插入，e次查找：cn(n+e)/m
  - hash函数的计算应该很高效

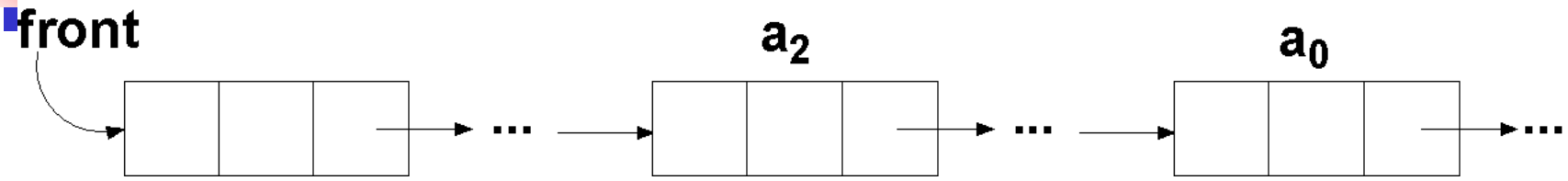
hash函数的选择

- 原则
  - 计算简单
  - 使表项均匀分布
- 常用算法
  - 字符→整数➔名字→整数序列$c_1, \ldots, c_k$
  - 对整数序列进行数学运算→单一整数结果
  - 对m取模
- 如：$h_0=0$，$h_i=65599*h_{i-1}+c_i$，$H=h_k\%m$

作用域的处理

- 每个作用域一个符号表
  - 对应活动记录，附着于语法树结点
- 处理作用域——为过程、块编号
  - 符号表项——保存名字和过程号
  - 名字匹配——名字字符串&过程编号
  - 符号表操作
    - lookup：查找最近创建的表项
    - insert：创建新表项
    - delete：删除最近创建的表项

链表实现方式



- front：最近创建的符号表项
- insert：创建表项，插入到insert之前
- lookup：从front开始遍历链表
- delete
  - 无需保存过程编号
  - 同一过程名字相邻，保存每个过程位置即可

hash表实现方式

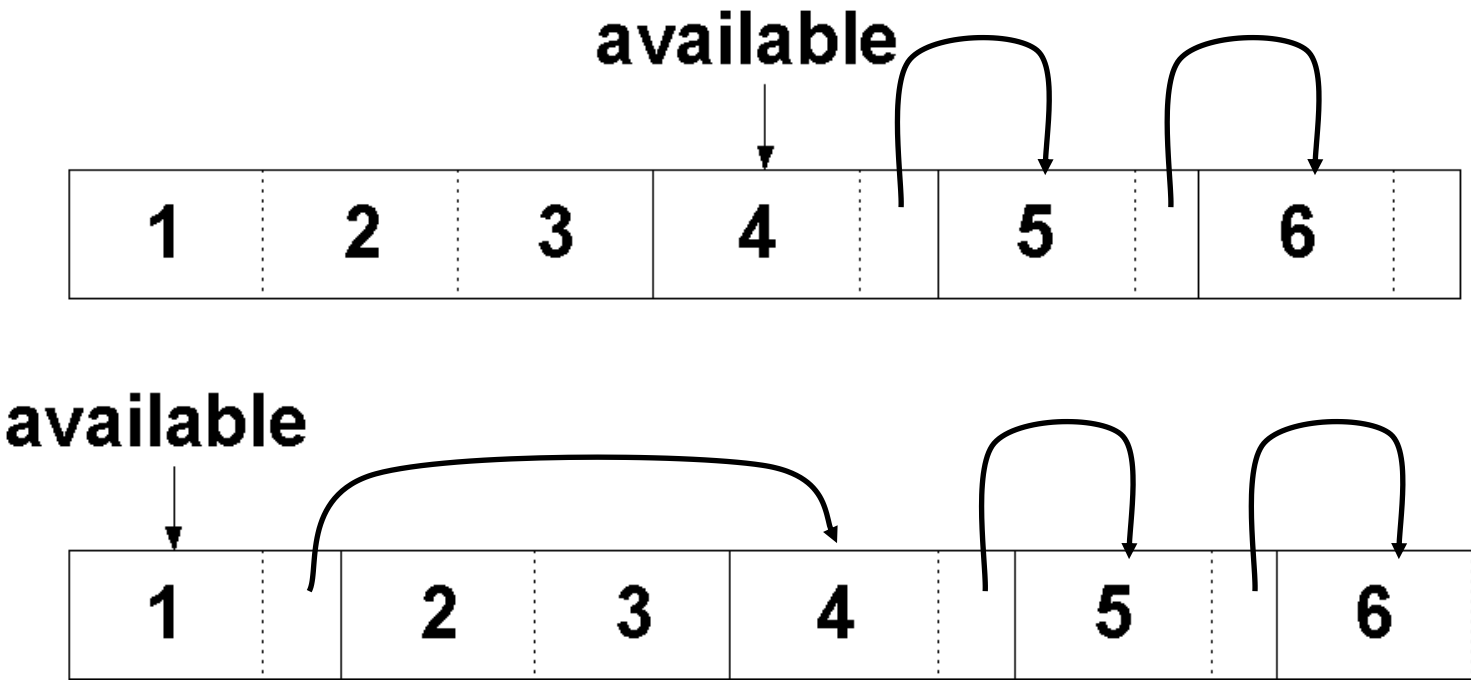
- lookup：相同名字总是存入同一链表，在对应链表搜索相同过程编号者即可
- delete
  - 无需遍历hash表所有链表
  - 作用域指针，scope link，相同作用域的所有表项串接起来
  - 指针设置问题

7.7 高级语言的动态内存分配

- 堆
- 显式方式：new, delete
- 隐式方式：中间计算结果
- 垃圾内存（garbage）
  - 已分配内存未被任何指针引用
  - 垃圾收集，garbage collection
- 空悬引用（dangling reference）
  - 指针指向无效区域

7.8 动态内存分配的处理

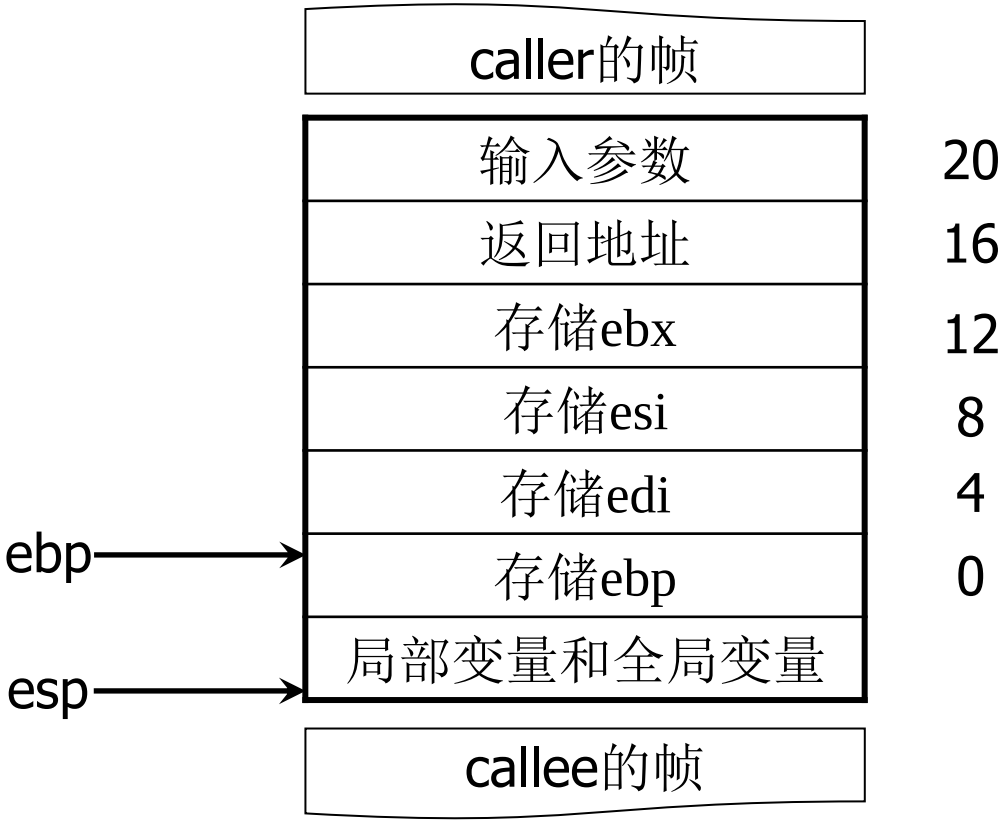
- 显式方式，定长块分配



隐式方式的内存释放问题

- 记录指向内存块的所有指针
- 处理两个问题
  1. 块边界
  2. 辨别块是否被使用
     a) 引用计数法，reference count
        记录引用块的指针数目
        计数=0➔可以释放
        性能影响较大
     b) 标记技术，marking technique
        暂停程序，检查所有指针
        引用的内存块标记为used
        未引用的内存块标记未unused，可释放

lcc的运行时环境处理



lcc的运行时环境处理

- 调用序列

```
print("%s: \n", f->x.name);
print("push ebx\n");
print("push esi\n");
print("push edi\n");
print("push ebp\n");
print("mov ebp,esp\n");
```

lcc的运行时环境处理

- 处理参数

```
offset = 16 + 4;
for (i = 0; callee[i]; i++) {
    Symbol p = callee[i];
    Symbol q = caller[i];
    p->x.offset = q->x.offset;
    p->x.name = q->x.name = stringf("%d", p->x.offset);
    p->sclass = q->sclass = AUTO;
    offset += roundup(q->type->size, 4);
}
```

lcc的运行时环境处理

- 返回序列

```
print(“mov esp,ebp\n”);
print(“pop ebp\n”);
print(“pop edi\n”);
print(“pop esi\n”);
print(“pop ebx\n”);
print(“ret\n”);
```

lcc的运行时环境处理

- 数据定义
  - 静态局部符号保证唯一名字

```
if (p->scope >= LOCAL && p->sclass == STATIC)
    p->x.name = stringf("L%d", genlabel(1));
```

  - 生成的符号→合法汇编名

```
if (p->generated)
    p->x.name = stringf("L%s", p->name);
```

  - 全局符号加下划线前缀

```
if (p->scope == GLOBAL || p->sclass == EXTERN)
    p->x.name = stringf("_%s", p->name);
...
```

lcc的运行时环境处理

- 数据定义
  - 空间分配

```
case C: print(“db %d\n”, v.uc); return;
case S: print(“dw %d\n”, v.ss); return;
case I: print(“dd %d\n”, v.i); return;
case U: print(“dd 0%xH\n”, v.u); return;
case P: print(“dd 0%xH\n”, v.p); return;
…
```