



School of Computer Science & Technology
Harbin Institute of Technology

第8章 符号表管理

重点： 符号表的作用,符号表的组织结构,符号表与作用域。

难点： 符号表的组织结构及其性能评价。



第8章 符号表管理

8.1 符号表的作用

8.2 符号表中存放的信息

8.3 符号表的组织结构

8.4 符号表与作用域

8.5 本章小结



8.1 符号表的作用

- **编译的各个阶段都会用到符号表中登记的信息**
 - **协助进行语义检查和中间代码生成**
 - **在目标代码生成阶段，当需要为名字分配地址时，符号表中的信息将是地址分配的主要依据**
 - **编译器用符号表来记录、收集和查找出现在源程序中的各种名字及其语义信息。**



8.1 符号表的作用

- 符号表是以**名字为关键字**来记录其信息的数据结构
 - 支持的两个最基本操作是**添加**表项和**查找**表项，这两个操作必须是高效的



8.2 符号表中存放的信息

- **记录源程序中出现的各种名字及其属性信息是符号表的首要任务。**
- **同一个名字在一段程序中应该表示同一个对象，即同一个符号表中不能出现相同的名字，因此名字可以作为符号表的关键字。**



8.2 符号表中存放的信息

- 每一个符号表表项中需要存放的基本信息就是符号的**名字**及其**属性**。

	名字	属性
符号表表项1	abc	...
符号表表项2	i	...
⋮
符号表表项 n

图8.1 符号表的基本格式



8.2.1 符号表中的名字

- **情况1：名字字段长度固定**
 - **名字项的长度大小取决于标识符允许的最大长度**
 - **不适于标识符长度变化范围较大的语言**
 - **空间浪费**



8.2.1 符号表中的名字

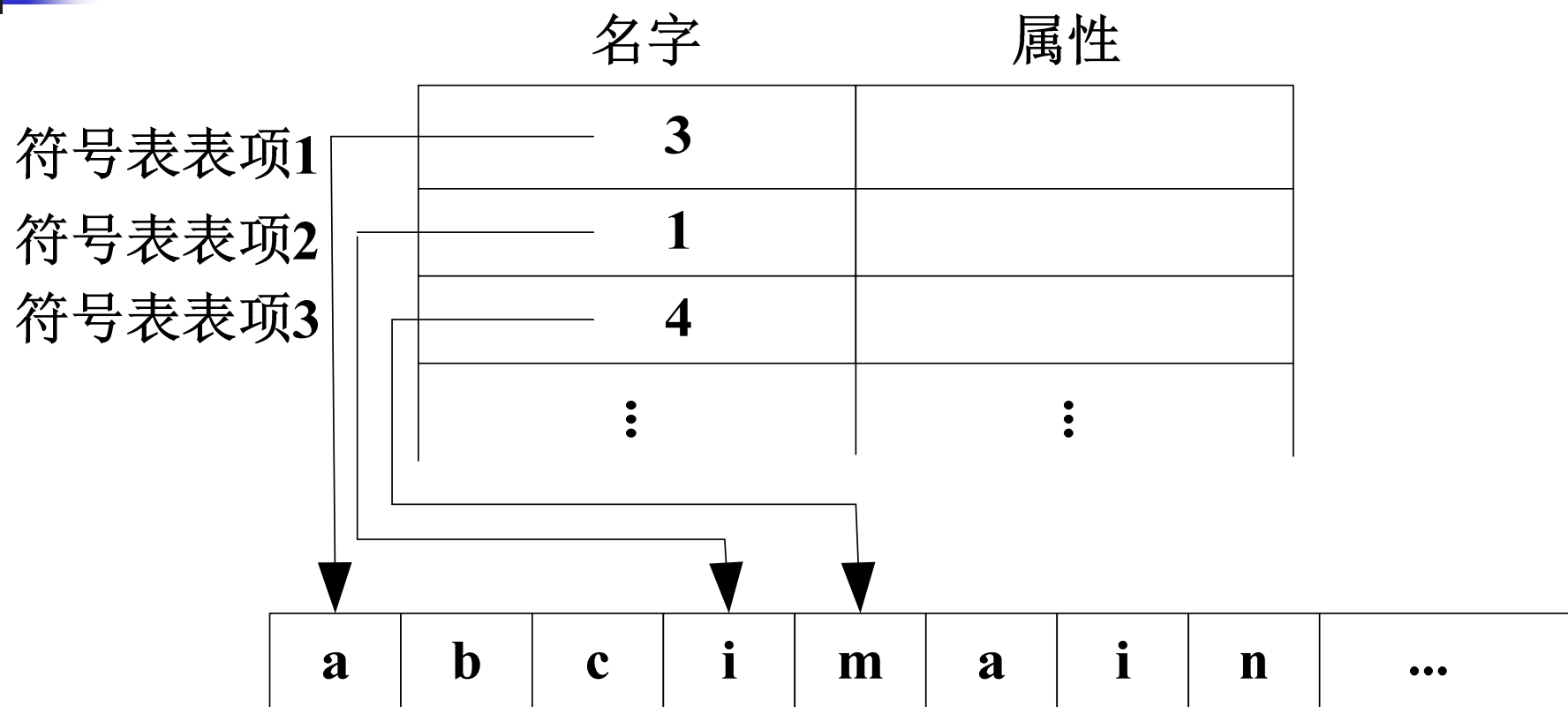
- **情况2：名字字段长度可变**
 - **标识符的长度没有限制**
 - **符号表上的操作复杂而低效**



8.2.1 符号表中的名字

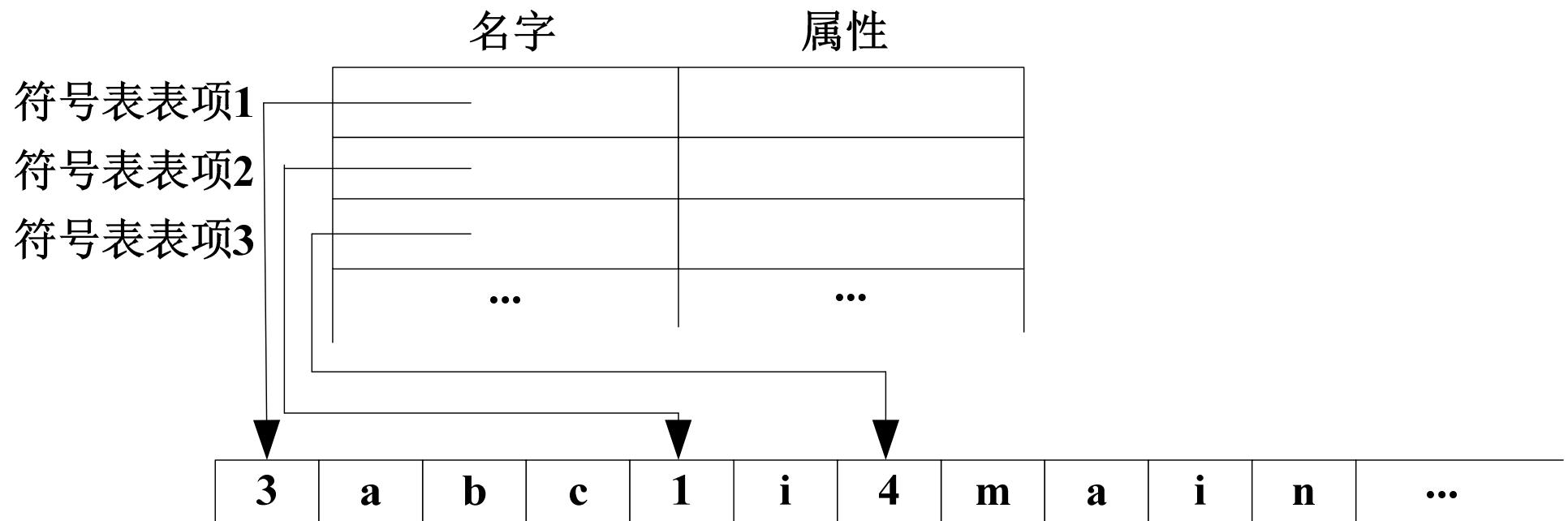
- **情况2：名字字段长度可变**
 - **引入一个单独的字符串表，将符号表中的全部标识符集放在这个字符串表**
 - **在符号表表项的名字部分只要给出相应标识符的首字符在字符串表中的位置即可**

8.2.1 符号表中的名字



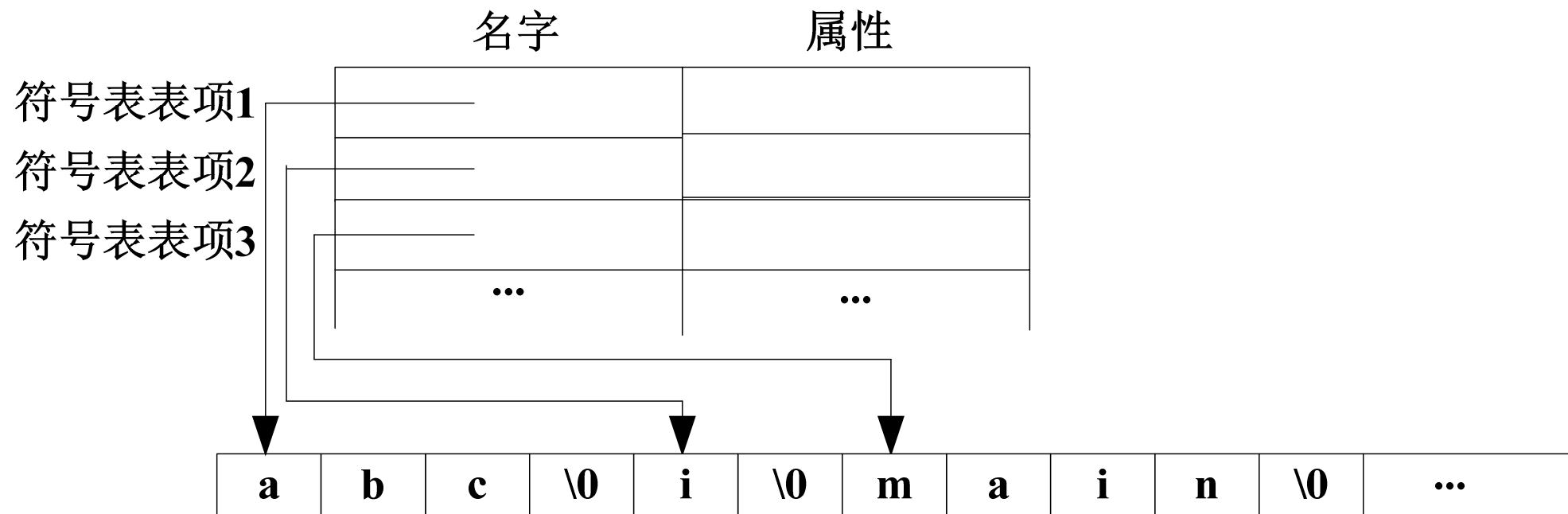
(a)标识符长度放在符号表中

8.2.1 符号表中的名字



(b) 标识符长度放在字符串中

8.2.1 符号表中的名字



(c) 用' \0' 表示标识符的结束



8.2.2 符号表中的属性

- **符号所表达的含义不同，符号表中需要存放的属性也就不同**
 - **数组名字需要存放的属性信息应该包括数组的维数、各维的维长等**
 - **函数的名字应该存放其参数个数、各参数的类型、返回值的类型等**



8.2.2 符号表中的属性

- **方法1：建立多个符号表来管理源程序中出现的各种符号，如常数表、变量表、函数表、数组表等**
 - **问题：可能出现不同种类符号重名的问题**



8.2.2 符号表中的属性

- **方法2：建立一张共用的大表来管理各种符号，需要在符号表增设一个标志来表明符号的种属**
 - **问题：不同种类符号所需存放属性信息在数量上的差异将会造成符号表的空间浪费**

8.2.2 符号表中的属性

	名字	基本属性			扩展属性
	符号种类	类型	地址	扩展属性指针	
符号表表项1	abc	变量	int	0	NULL
符号表表项2	i	变量	int	4	NULL
符号表表项3	myarray	数组	int	8	<div><div>维数</div><div>各维维长</div><div>234</div></div>
			

图8.3 多种符号共用符号表的一种实现结构

8.2.2 符号表中的属性

	名字	基本属性			
		符号种类	类型	地址	扩展属性指针
符号表表项1	swap	函数	int		
符号表表项2	a	形参	int *		
符号表表项3	b	形参	int *		NULL
	

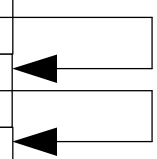


图8.4 用扩展属性链组织函数形参的符号表



8.2.3 符号的地址属性

- 如果采用静态存储分配策略，符号 x 绑定的地址等于静态分配的基址 $base$ 加上 x 的偏移量 $offset$
- 如果采用动态存储分配策略（栈式存储分配或堆式存储分配等），则符号是在程序执行过程中和地址动态绑定的。
 - 栈式存储分配时， i 的地址是以栈指针 sp 为基址加上 i 相对于活动记录起始地址的偏移量 $offset$
- 符号表中各符号的地址属性是该符号相对于第一个符号的偏移地址

8.3 符号表的组织结构

8.3.1 符号表的线性表实现

■ 用线性表实现符号表较为直观

- 数组实现：插入 n 个符号、执行 e 次查找操作的时间复杂度为 $T(n, e) = O(n(n+e))$
- 有序数组实现：插入 n 个符号、执行 e 次查找操作的时间复杂度为 $T(n, e) = e \times \log n + \sum_{i=1}^n (\log i) + \sum_{i=1}^n i / 2 \leq O(n+e) \log n + O(n^2)$
- 有序符号表结构只有在下面的情况下才能取得较好效果：和插入操作次数相比，符号表表项上的查找操作次数占绝对多数，即 $e \gg n$ 。

8.3.2 符号表的散列表实现

- 散列表可以提高`lookup`操作的效率，同时也可以提高`insert`操作的效率，许多实际编译器的符号表实现中均采用了散列技术

开散列表表头数组

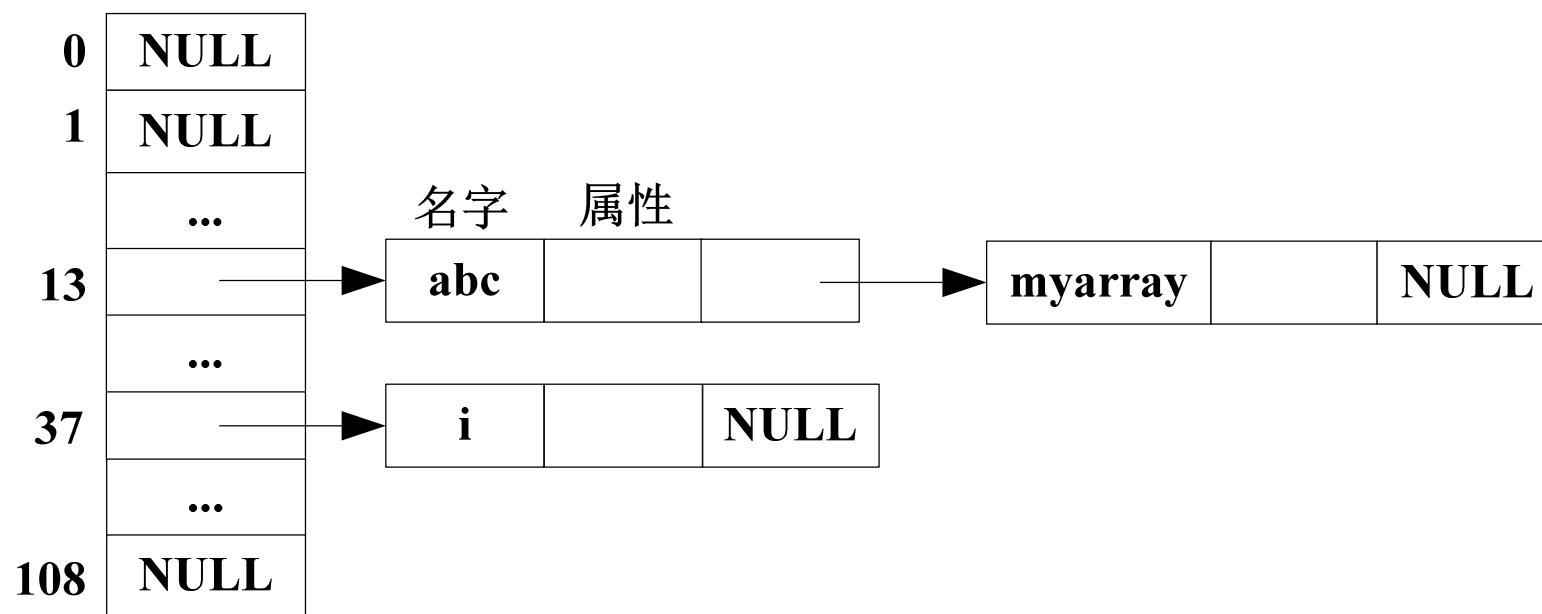


图8.6 一个符号表的散列表实现

8.3.2 符号表的散列表实现

- 插入 n 个符号，查找 e 个符号的 $lookup$ 操作和 $insert$ 操作的时间复杂度与 m (哈希桶数)有关，记为 $T(n,e,m)$ ， $T(n,e,m) \approx n(n+e)/m$
- 空间复杂度 $S(n,m) = O(n)$
- 散列函数应在满足 $\sum_{j=1}^m b_j = n$ 的前提下，使 $\sum_{j=1}^m b_j \times (b_j + 1) / 2$ 达到最小， b_j 表示哈希桶对应的外散列表长度

8.4 符号表与作用域

```
int main()
{
    int abc;
    abc = 1;
    {
        int abc;
        abc = 2;
        printf("abc is %d\n", abc);
    }
    printf("abc is %d\n", abc);
}
```

运行结果为:

abc is 2

abc is 1

说明abc在不同的范围内有效。
这个有效范围就是符号的作用域

8.4.1 程序块结构的符号表

变量的作用域满足最近嵌套原则

```
(1) int main()
(2) {
(3)     int a=0;
(4)     int b=0;
(5)     {
(6)         int b=1;
(7)         {
(8)             int a=2;
(9)             printf("%d %d\n", a, b );
(10)        }
(11)        {
(12)            int b=3;
(13)            printf("%d %d\n", a, b);
(14)        }
(15)        printf("%d %d\n", a, b);
(16)    }
(17)    printf("%d %d\n", a, b);
(18) }
```

B_0 : line 1 to line 18
 B_1 : line 5 to line 16
 B_2 : line 7 to line 10
 B_3 : line 11 to line 14

8.4.1 程序块结构的符号表

- 为每个程序块建立一个符号表，程序块内的符号记录在该程序块所对应的符号表
- 建立符号表之间的联系，刻画出符号的嵌套作用域

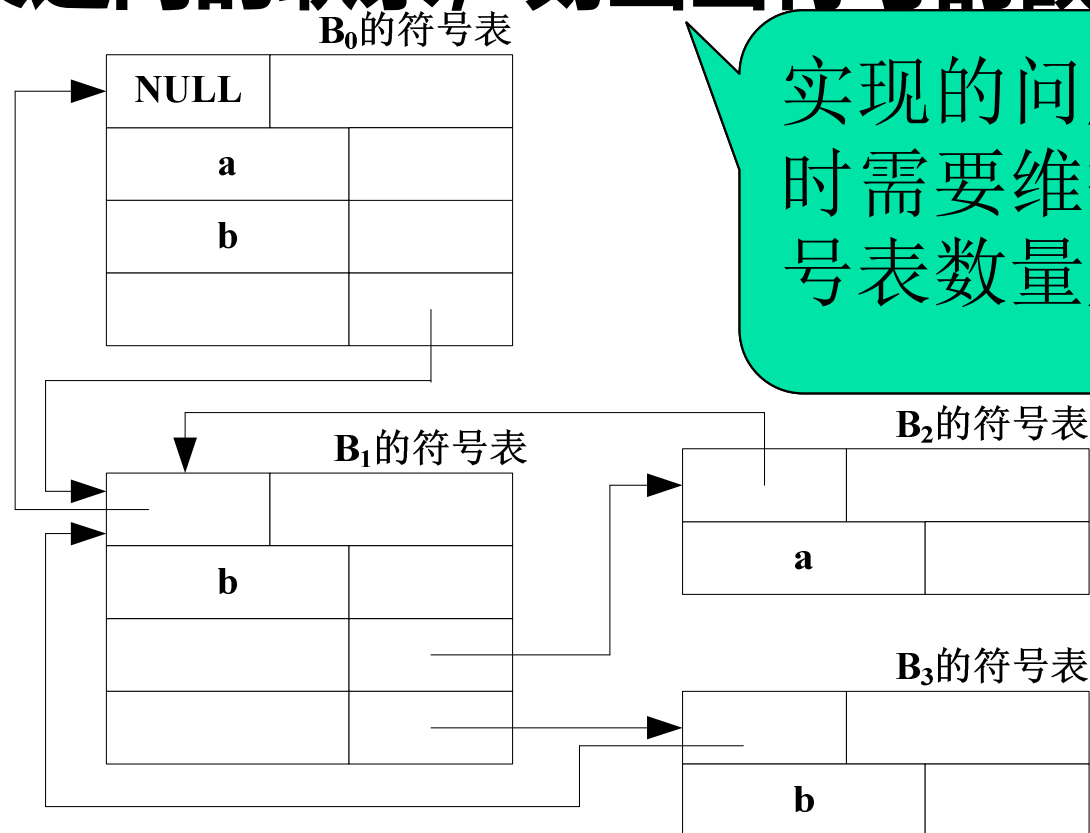


图8.11 图8.10中的程序所对应的符号表

8.4.2 程序块结构符号表的其他实现

- [1] 将所有块的符号表放在一个大数组中，然后引入一个**程序块表**来描述各程序块的符号表在大数组中的位置及其相互关系

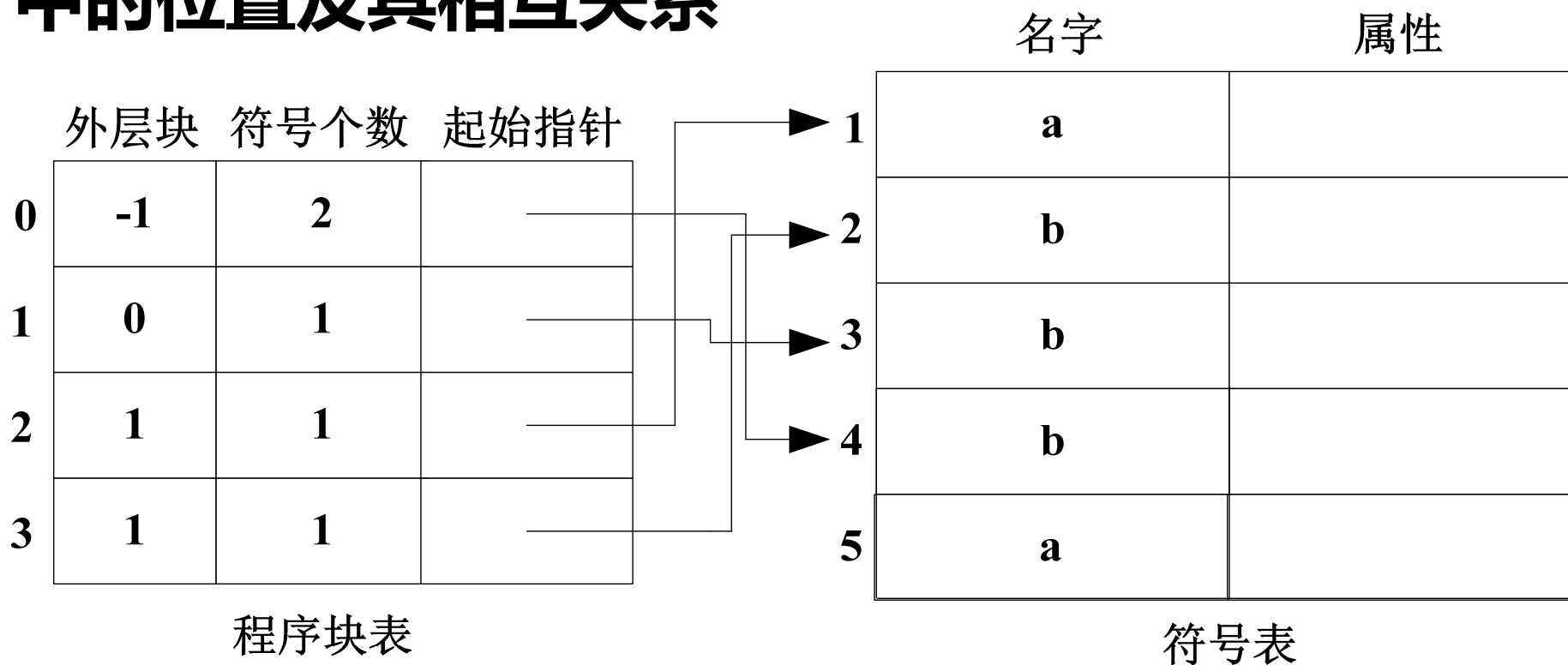


图8.12 图8.10的另一种符号表结构



8.4.2 程序块结构符号表的其他实现

- **[2] 将符号所属程序块编号放在符号表表项中。**
 - 查找某个符号的名字 $name$ 时，只有当 $name$ 和符号表中的名字字符串完全匹配，且符号表表项中的块编号和当前处理的块编号完全相同时才算查找成功
 - 程序块编号可以通过在语法制导定义中的块开始处和块结束处添加适当的语义规则计算得出。



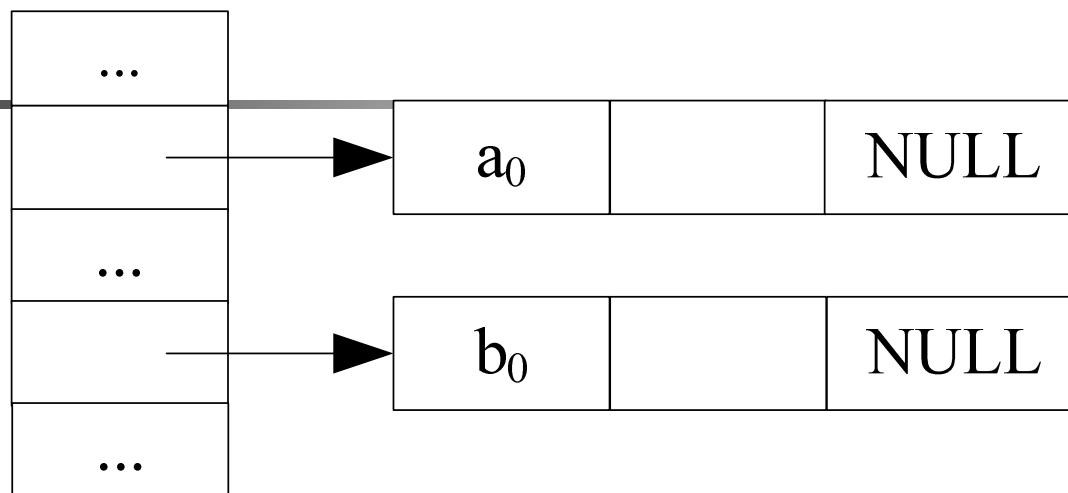
8.4.2 程序块结构符号表的其他实现

■ 程序块满足最近嵌套原则

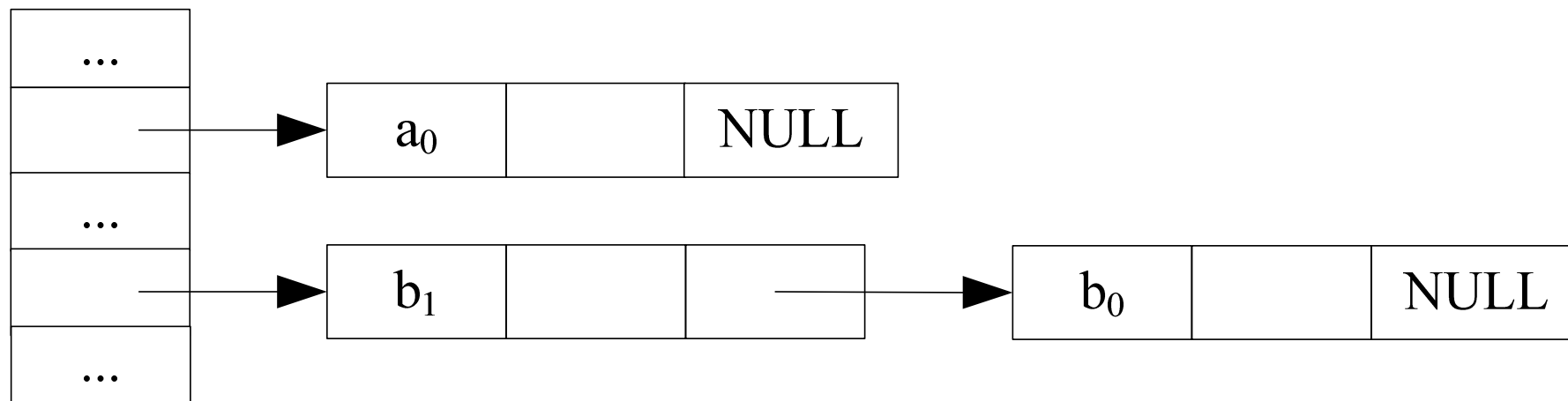
- 内层程序块中的局部变量只有全部处理完成之后才进入外层块
- 一旦进入外层程序块，内层块的局部变量就不会再使用了，可以从符号表中将这些符号删除
- 若用队列实现，符号表中最前面的符号一定是当前正在处理的块中的局部变量
- 符号表表项中可以不用存放块编号，而是根据符号表表项在符号表中的位置来判断。

8.4.2 程序块结构符号表的其他实现

对图8.10
中的程序



(a) 处理到语句(5)时的符号表



(b) 处理到语句(7)时的符号表

(1) **int main()**

(2) **{**

(3) **int a=0;**

(4) **int b=0;**

(5) **{**

(6) **int b=1;**

(7) **{**

(8) **int a=2;**

(9) **printf(“%d %d\n”, a, b);**

(10) **}**

(11) **{**

(12) **int b=3;**

(13) **printf(“%d %d\n”, a, b);**

(14) **}**

(15) **printf(“%d %d\n”, a, b);**

(16) **}**

(17) **printf(“%d %d\n”, a, b);**

(18) **}**

B_0 : line 1 to line 18

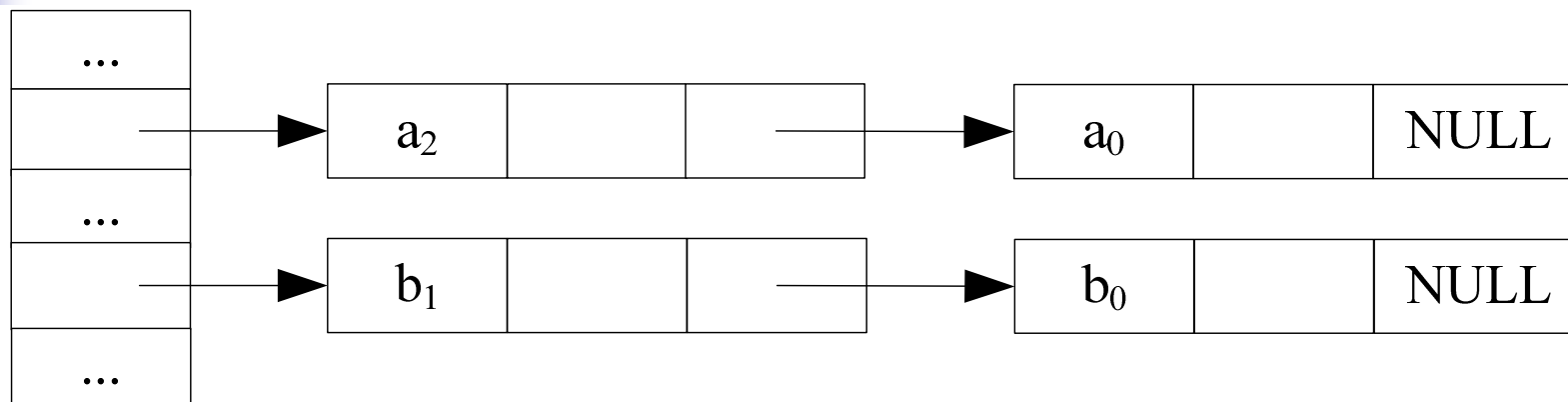
B_1 : line 5 to line 16

B_2 : line 7 to line 10

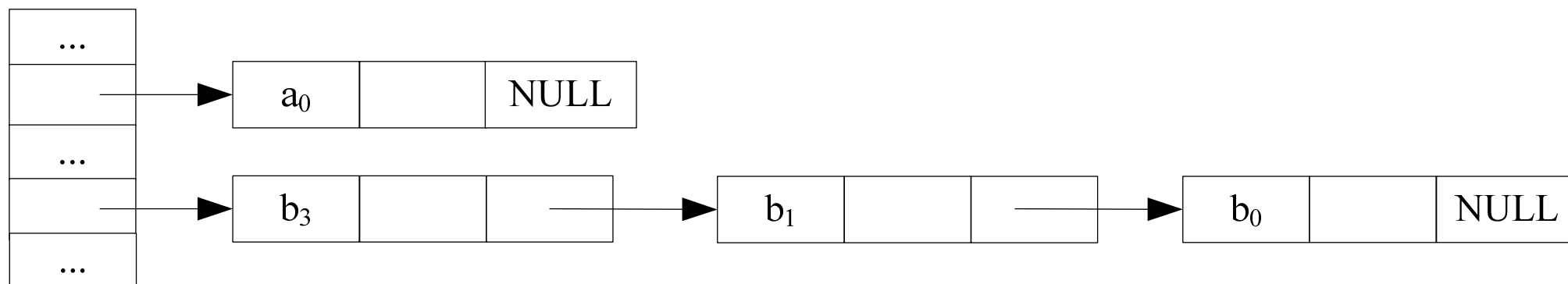
B_3 : line 11 to line 14

8.4.2 程序块结构符号表的其他实现

对图8.10中的程序



(c) 处理到语句(9)时的符号表



(d) 处理到语句(13)时的符号表



8.4.3 C语言的符号表

- 一个完整的C程序由一个或多个相对独立的函数组成，函数之间的通信依靠**参数传递**和**全局变量**
- 全局变量和函数名的作用域是整个程序，而其余变量的作用域则是定义它们的函数



8.4.3 C语言的符号表

- 如果采取将每个函数分别编译成目标代码，然后**链接装配**成一个可执行程序的处理方式，则每个函数中的符号经一遍处理即可，而且源程序中的多个函数是一个接一个处理的，不会出现交叉（即无嵌套）

8.4.3 C语言的符号表

```
int g_array[10]
int main()
{
}
int quicksort()
{
}
}
```

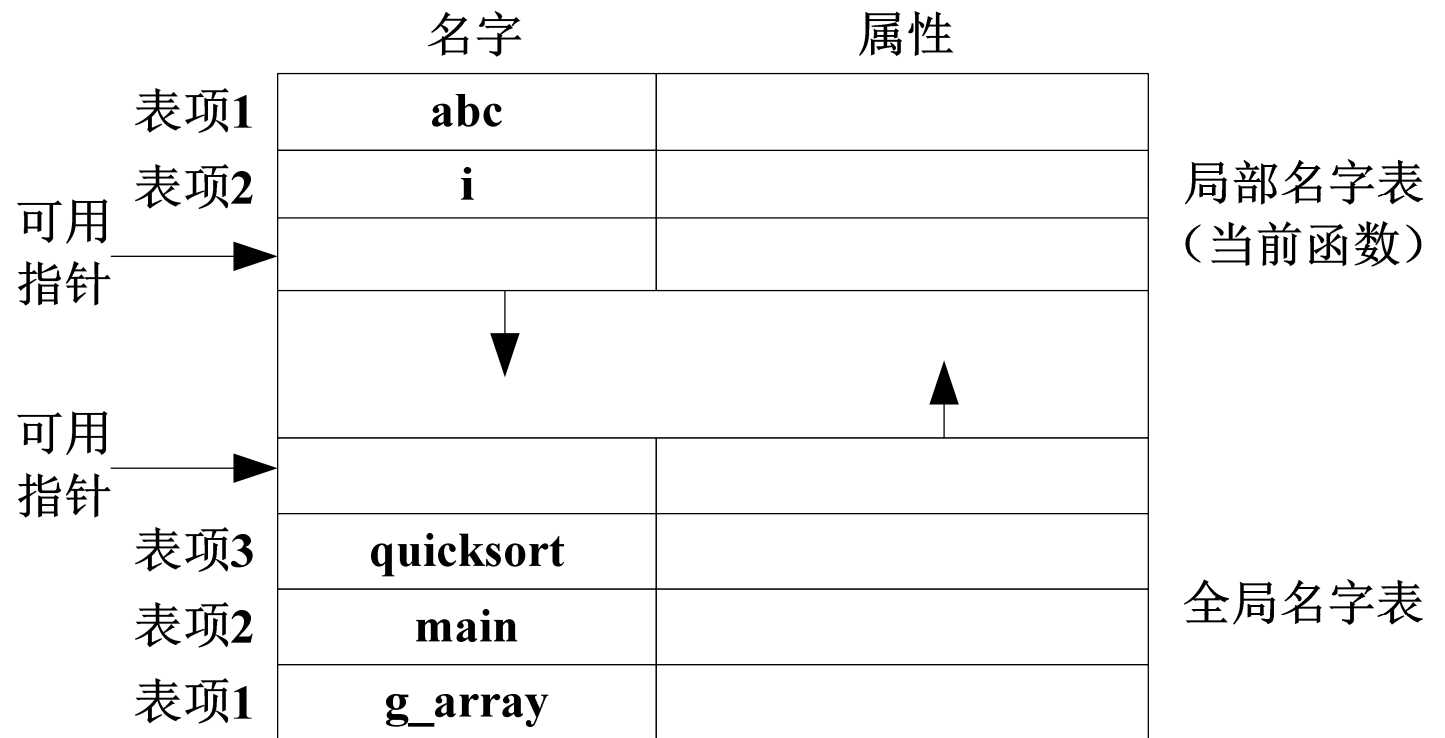


图8.16 一个完整C程序的符号表

符号表中只需要一个局部名字表来管理当前编译的函数中的符号，处理完毕即从该表中删除。

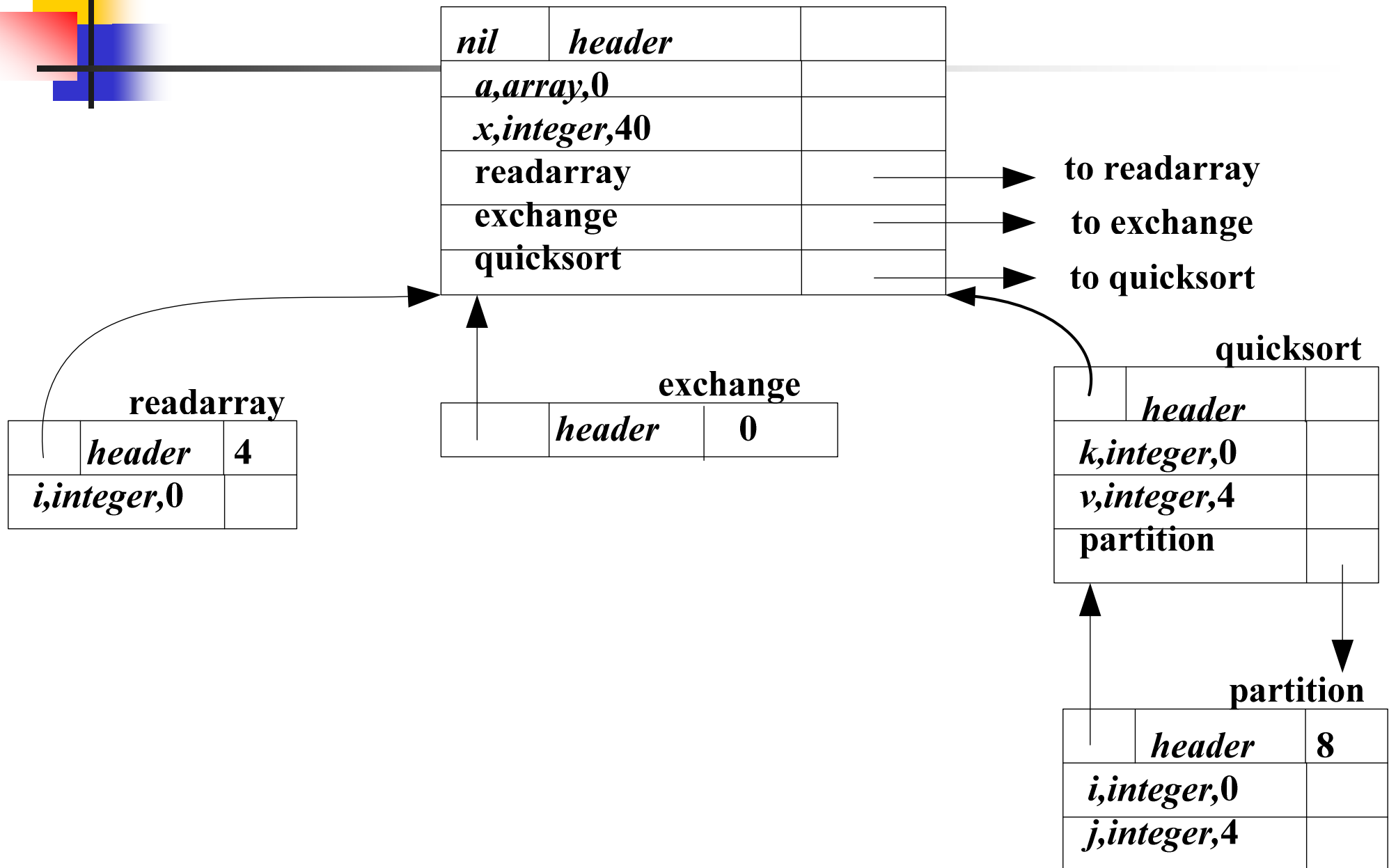


8.4.4 嵌套过程的符号表

Pascal等允许在过程中嵌套定义其它过程

```
program sort(input, output);  
  procedure readarray;  
    begin ... end{ readarray };  
  procedure exchange( i, j : integer );  
    begin ... end{ exchange };  
  procedure quicksort( m, n : integer );  
    function partition( x, y : integer );  
      begin ... end{ partition };  
    begin ... end{ quicksort };  
  begin ... end{ sort };
```

8.4.4 嵌套过程的符号表





本章小结

- **符号表用来存放编译器各阶段收集来的各种名字的类型和特征等有关信息，并供编译程序用于语法检查、语义检查、生成中间代码及生成目标代码等；**
- **源程序中会出现各种各样的名字，如函数名、函数参数名、函数中的局部变量名、全局变量名、数组名、结构名、文件名等，相应的属性可以是种属、类型、地址等。**



本章小结

- **根据符号所需的属性个数和类型的不同，可以组成不同的符号表，也可以组成统一的符号表，在组成统一符号表时，需要采用恰当的组织结构，以便可以对其进行高效处理。**
- **随着程序规模的扩大，符号名的数量会很大，因而必须关注符号表的组织和高效管理。无序线性符号表、有序线性符号表、散列表示符号表具有不同性能的组织形式。**



本章小结

- **符号表管理中必须关注到语言所规定的符号的作用域，特别是在嵌套结构的程序中符号的作用域是分层的。**
- **C语言符号表的管理。**