Lecture 8: 运行时刻环境

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

从编译到执行

- **编译器创建并管理运行时刻环境**
  - 为数据分配安排存储位置---关键任务
  - 确定访问变量时使用的机制
  - 过程之间的连接
  - 参数传递
  - 和操作系统、输入输出设备相关的其它接口
- **本章主题**
  - 存储管理：栈分配、堆管理、垃圾回收
  - 对变量、数据的访问

过程的声明/定义

- **过程：函数、过程、方法、子例程的统称**
- **过程定义是一个声明，它的最简单形式是将一个名字和一个语句联系起来。**
  - 该名字是过程名，而这个语句是过程体。在大多数语言中，返回值的过程叫做函数，完整的程序也可以看作一个过程。

过程的执行

- **当过程名出现在调用语句中时，就说这个过程在该点被调用**
  - 过程调用就是执行被调用过程的过程体
  - 过程调用也可以出现在表达式中，这时也叫做函数调用
- **运行时过程的一次执行称为过程的一次活动**
  - 过程的活动需要可执行代码和存放所需信息的存储空间，后者通常用一块连续的存储区来管理，称为活动记录

存储组织

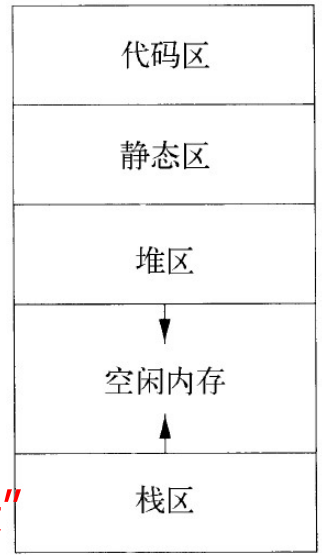
运行时刻内存分配的典型方式

- 目标程序在它自己的逻辑地址空间内运行，OS将逻辑地址映射为物理地址(内存空间实际编址)
- 四个字节构成一个机器字，多字节数据对象存储在一段连续的字节中，并把第一个字节作为它的地址

目标程序的代码放置在代码区

静态区、堆区、栈区分别放置不同类型生命期的数据值

栈区存放了“活动记录”



静态和动态存储分配

- **程序中同一个名字在运行时指向不同存储位置**
  - 静态（表示编译时刻）：编译器只需要通过scan source code，就可以完成存储分配
  - 动态（表示运行时刻）：无法在编译器决定，运行时才决定

静态和动态存储分配

- **静态分配**
  - 名字在程序被编译时绑定到存储单元，不需要运行时的任何支持
  - 绑定的生存期是程序的整个运行期间

静态分配给语言带来限制

- 递归过程不被允许
- 数据对象的长度和它在内存中位置的限制，必须是在编译时可以知道的
- 数据结构不能动态建立

静态分配给语言带来限制

- **例　C程序的外部变量、静态局部变量以及程序中出现的常量都可以静态分配**
- **声明在函数外面**
  - 外部变量 —— 静态分配
  - 静态外部变量 —— 静态分配
- **声明在函数里面**
  - 静态局部变量 —— 也是静态分配
  - 自动变量 —— 不能静态分配

动态存储分配

- **动态分配**
  - 栈式存储：和过程的调用/返回同步进行分配和回收，值的生命期和过程生命期相同
  - 堆存储：数据对象比创建它的过程调用更长寿
    - 手工进行回收
    - 垃圾回收机制

空间的栈式分配

活动树

- **过程调用（过程活动）在时间上总是嵌套的**
  - 后调用的先返回
  - 因此用栈式分配来分配过程活动所需内存空间
- **程序运行的所有过程活动可以用树表示**
  - 每个结点对应于一个过程活动
  - 根结点对应于main的过程活动
  - 过程p的某次活动对应的结点的所有子结点：此次活动所调用的各个过程活动（从左向右，表示调用的先后顺序）

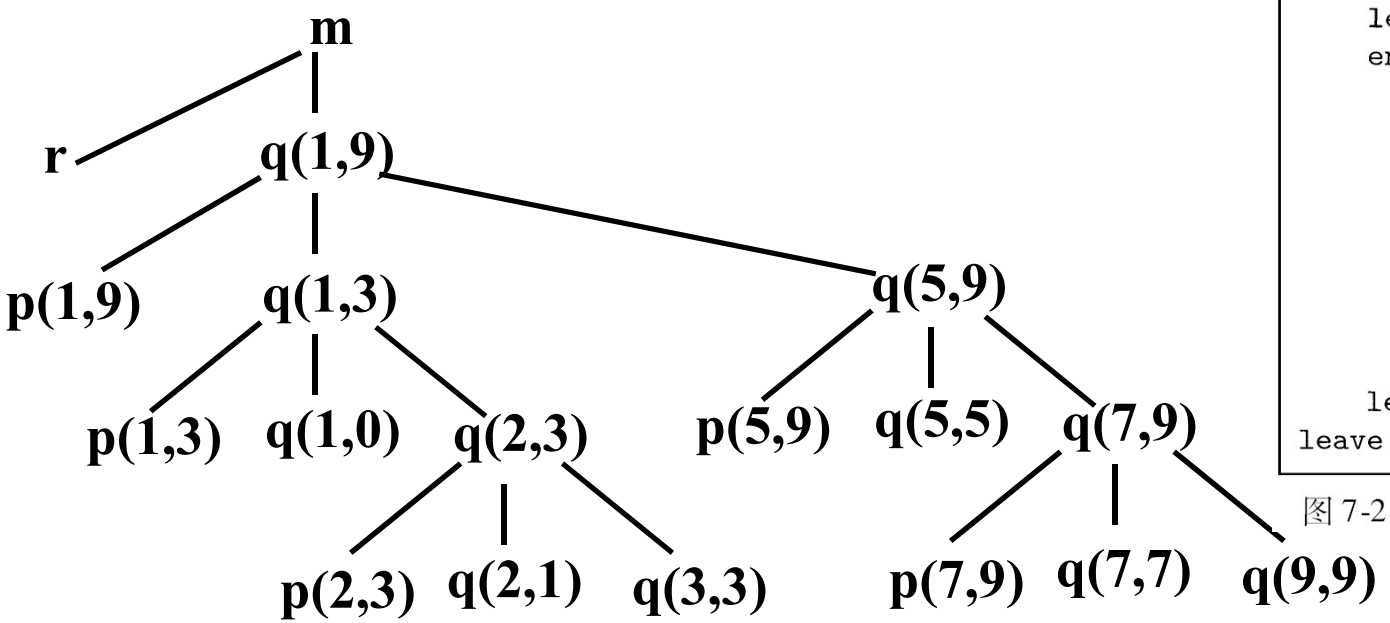
活动树

```
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，
       并且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);
}
```

图 7-2　一个快速排序程序的概要

活动树

- 用树来描绘控制进入和离开活动的方式




```
enter main()
    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)
leave main()
```

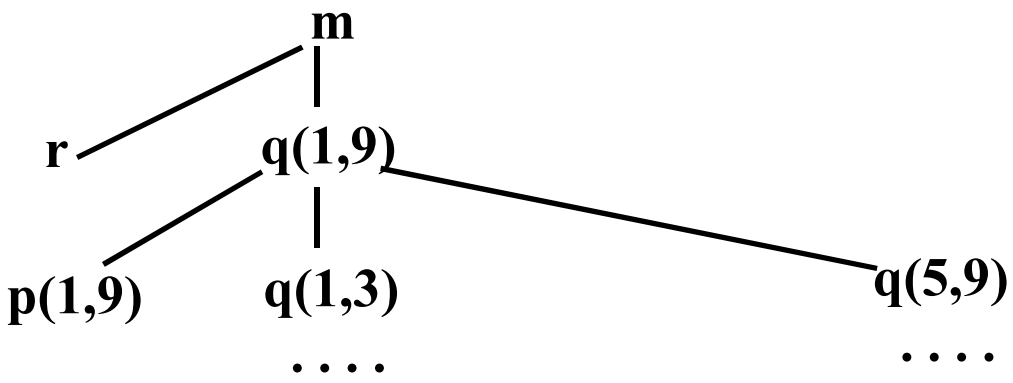
图 7-2 中程序的可能的活动序列

活动树的特点

- 每个结点代表某过程的一个活动
- 根结点代表主程序的活动
- 结点*a*是结点*b*的父结点，当且仅当控制流从*a*的活动进入*b*的活动
- 结点*a*处于结点*b*的左边，当且仅当*a*的生存期先于*b*的生存期

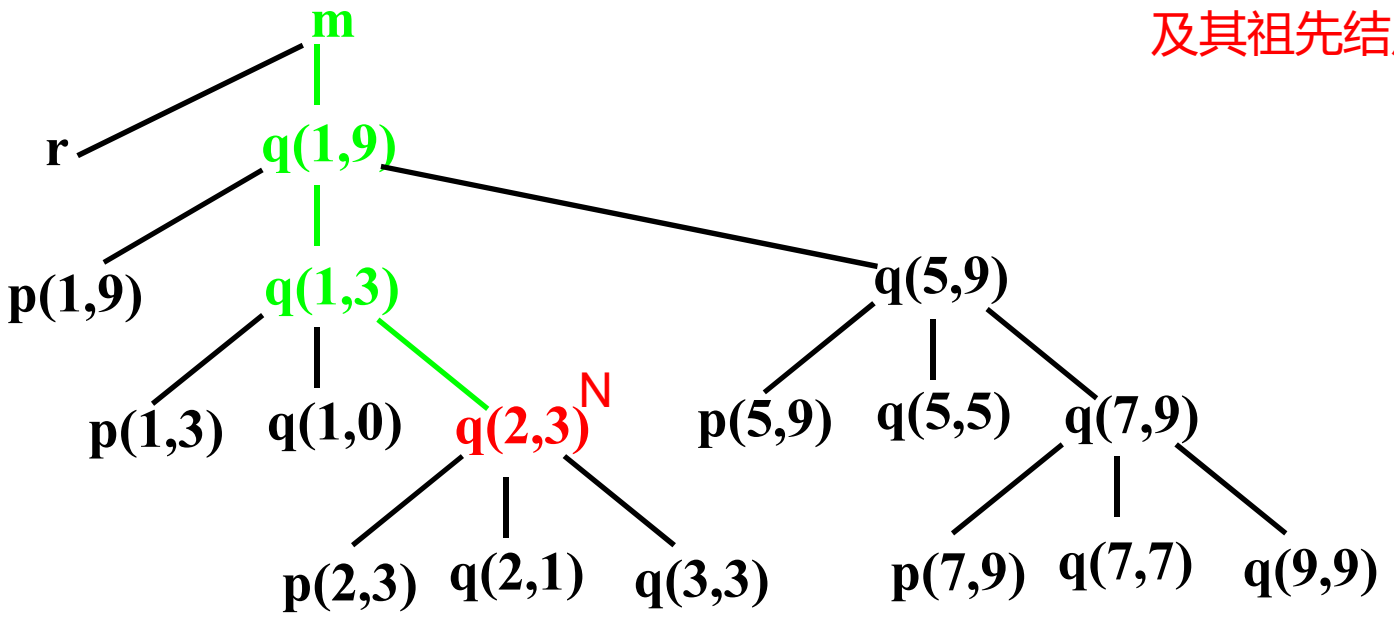
过程调用（返回）序列和活动树的前序（后序）遍历对应

假定当前活动对应结点N，那么所有尚未结束的活动对应于N及其祖先结点。



活动树和栈的关系

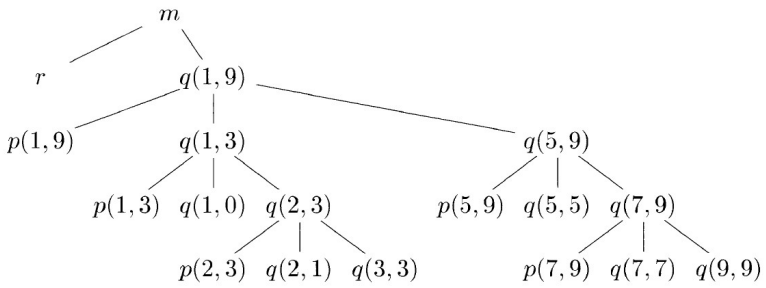
- **当前活跃着的过程活动可以保存在一个栈中**
  - **控制栈的内容：m, q (1, 9), q (1, 3), q (2, 3)**

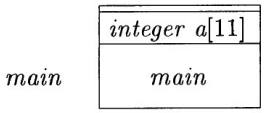


假定当前活动对应结点N，那么所有尚未结束的活动对应于N及其祖先结点。

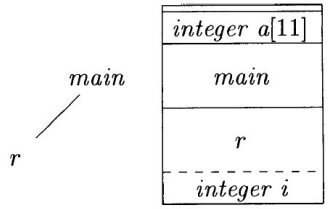
运行时刻栈的例子

- a[11]为全局变量
- main没有局部变量
- r有局部变量i
- q的局部变量i，和参数m,r

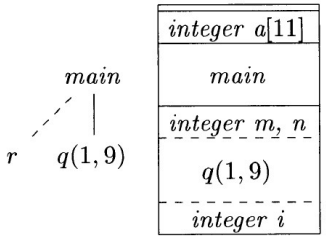




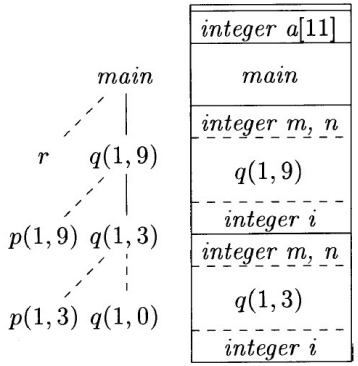
a)



b) γ被激活



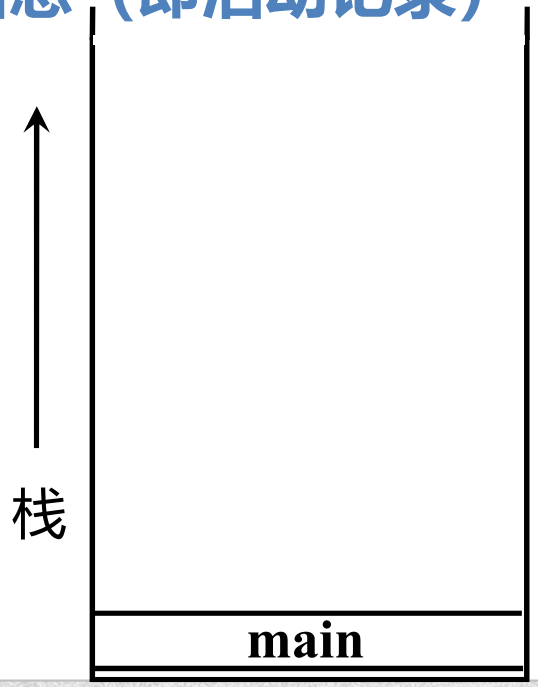
c) γ被弹出栈，$q(1,9)$ 被压栈



d) 控制返回到 $q(1, 3)$

运行时刻栈的例子

运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）

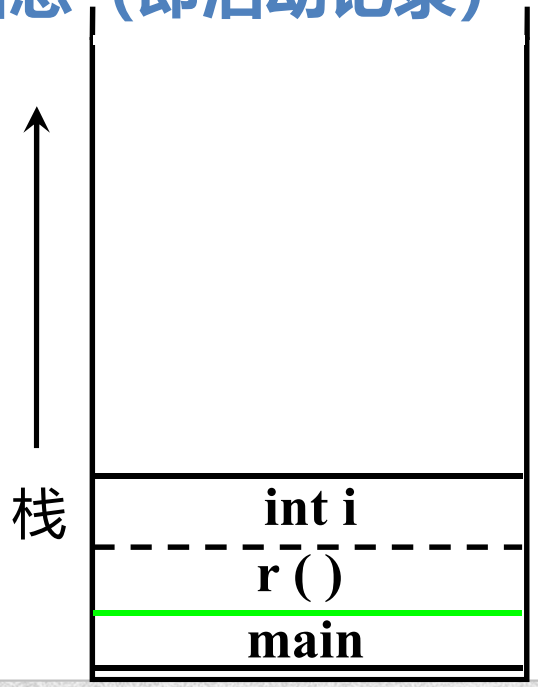


函数调用关系树

main

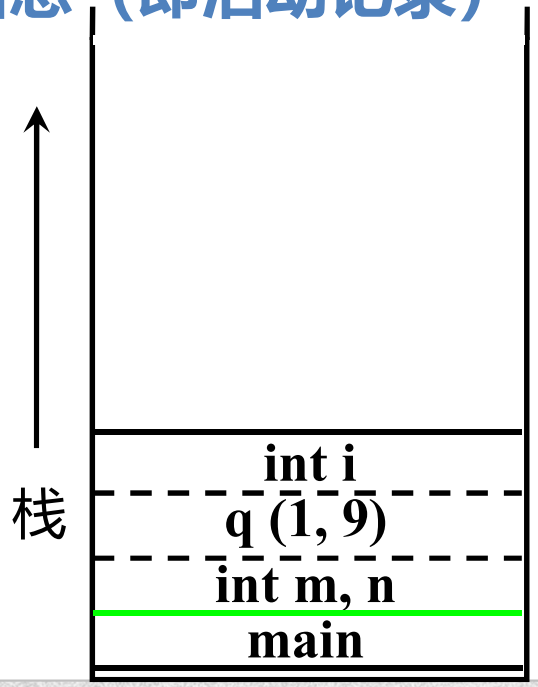
运行时刻栈的例子

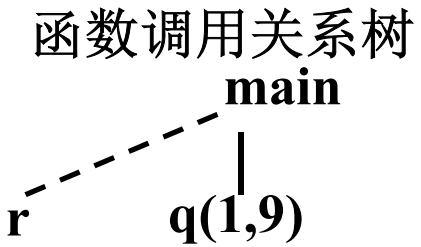
运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



运行时刻栈的例子

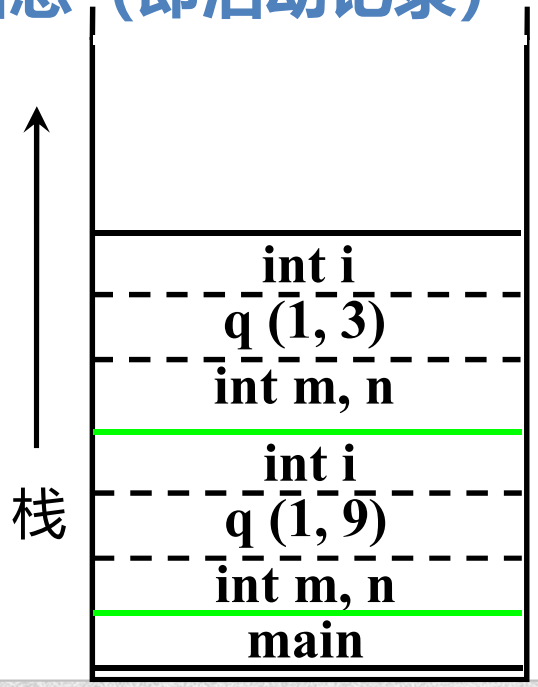
运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



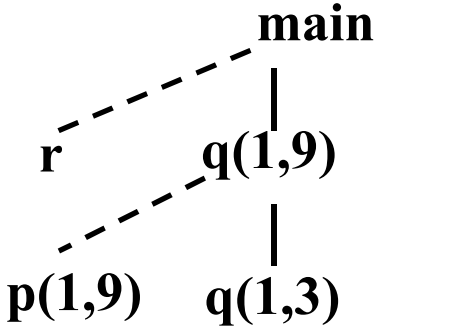


运行时刻栈的例子

运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）

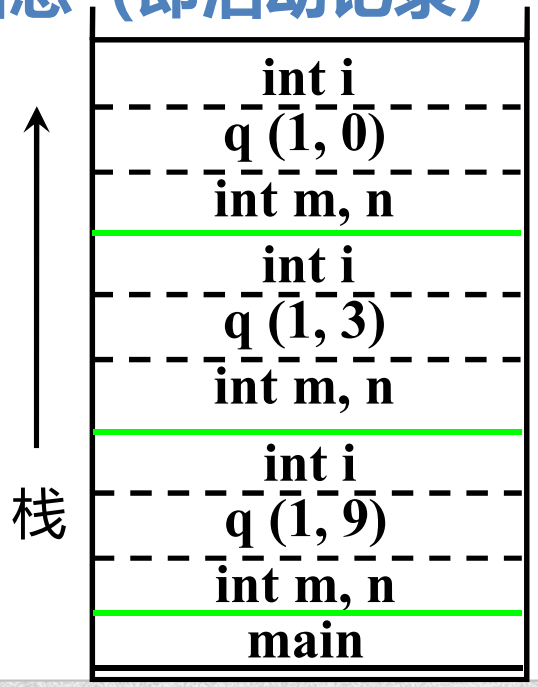


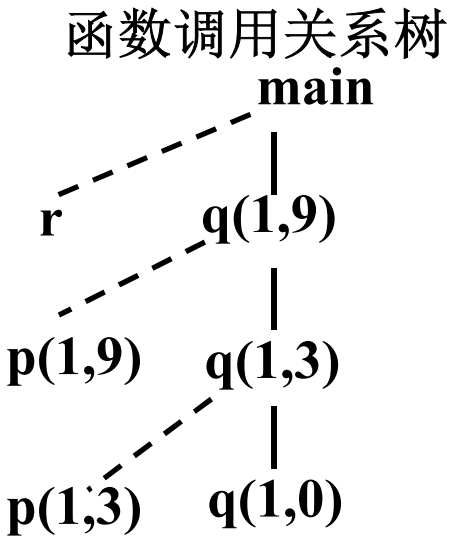
函数调用关系树



运行时刻栈的例子

运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）





活动记录的布局原则

- **过程调用和返回由控制栈进行管理，每个活跃的活动对应于栈中的一个活动记录，活动记录按照活动的开始时间，从栈底到栈顶排列**

1）临时值。比如当表达式求值过程中产生的中间结果无法存放在寄存器中时，就会生成这些临时值。

2）对应于这个活动记录的过程的局部数据。

3）保存的机器状态，其中包括对此过程的此次调用之前的机器状态信息。这些信息通常包括返回地址(程序计数器的值，被调用过程必须返回到该值所指位置)和一些寄存器中的内容(调用过程会使用这些内容，被调用过程必须在返回时恢复这些内容)。

4）一个“访问链”。当被调用过程需要其他地方(比如另一个活动记录)的某个数据时需要使用访问链进行定位。访问链将在7.3.5节中讨论。

5）一个控制链(control link)，指向调用者的活动记录。

6）当被调用函数有返回值时，要有一个用于存放这个返回值的空间。不是所有的被调用过程都有返回值，即使有，我们也可能倾向于将该值放到一个寄存器中以提高效率。

7）调用过程使用的实在参数(actual parameter)。这些值通常将尽可能地放在寄存器中，而不是放在活动记录中，因为放在寄存器中会得到更好的效率。然而，我们仍然为它们预留了相应的空间，使得我们的活动记录具有完全的通用性。

活动记录的布局原则

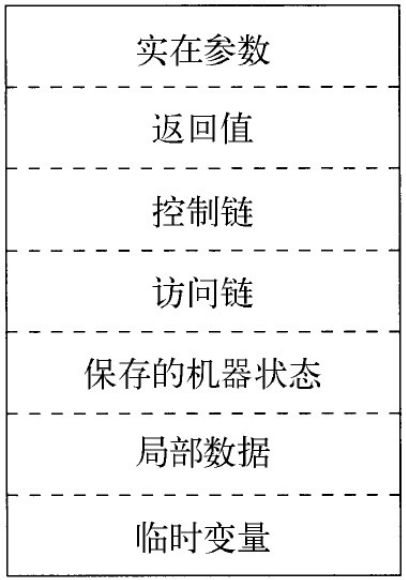
- **即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异**
- **设计这些序列和活动记录的一些原则**
  - 以活动记录中间的某个位置作为基地址
  - 长度能较早确定的域放在活动记录的中间

活动记录的布局原则

- **即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异**
- **设计这些序列和活动记录的一些原则**
  - 一般把临时数据域放在局部数据域的后面
  - 把参数域和可能有的返回值域放在紧靠调用者活动记录的地方

活动记录的布局原则

- **即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异**
- **设计这些序列和活动记录的一些原则**
  - 用同样的代码来执行各个活动的保存和恢复



调用代码序列

- **调用代码序列(calling sequence)：实现过程调用的代码段**
  - 为一个活动记录在栈中分配空间，并在此填写记录中的信息
- **返回代码序列(return sequence)：恢复机器状态，使调用者继续运行**
- **调用代码序列会分割到调用者和被调用者中**
  - 根据源语言、目标机器、操作系统的限制，可以有不同的分割方案
  - 把代码尽可能放在被调用者中

调用/返回代码序列的要求

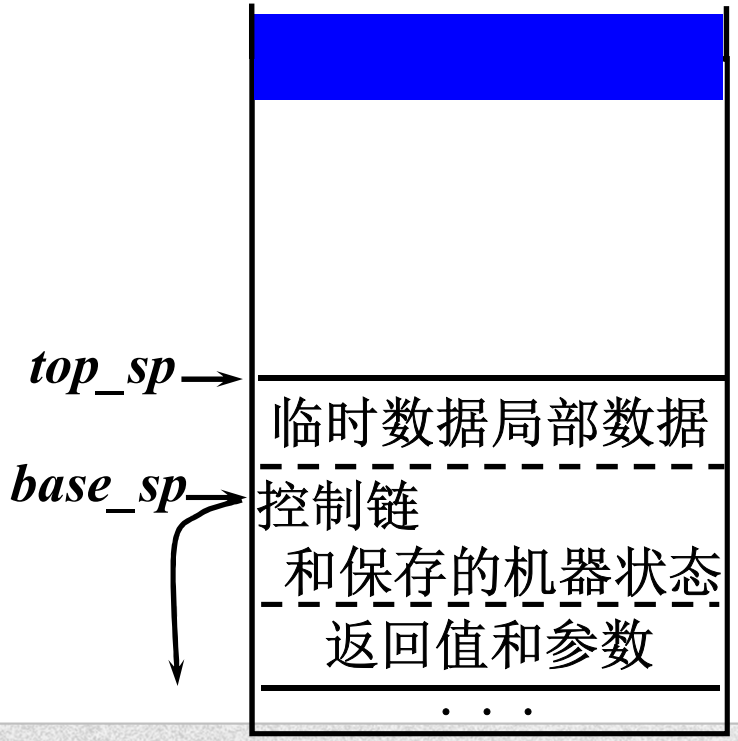
- **数据方面**
  - 能够把参数正确地传递给被调用者
  - 能够把返回值传递给调用者
- **控制方面**
  - 能够正确转到被调用过程的代码开始位置
  - 能够正确转回调用者的调用位置（的下一条指令）
- **调用代码序列和活动记录的布局相关**

调用代码序列的例子

- **Calling sequence**
  - 调用者计算实在参数的值
  - 将返回地址和原top_sp存放到被调用者的活动记录中。调用者增加top_sp的值（越过了局部数据、临时变量、被调用者的参数、机器状态字段）
  - 被调用者保存寄存器值和其他状态字段
  - 被调用者初始化局部数据、开始运行
- **Return sequence**
  - 被调用者将返回值放到和参数相邻的位置
  - 恢复top_sp和寄存器，跳转到返回地址

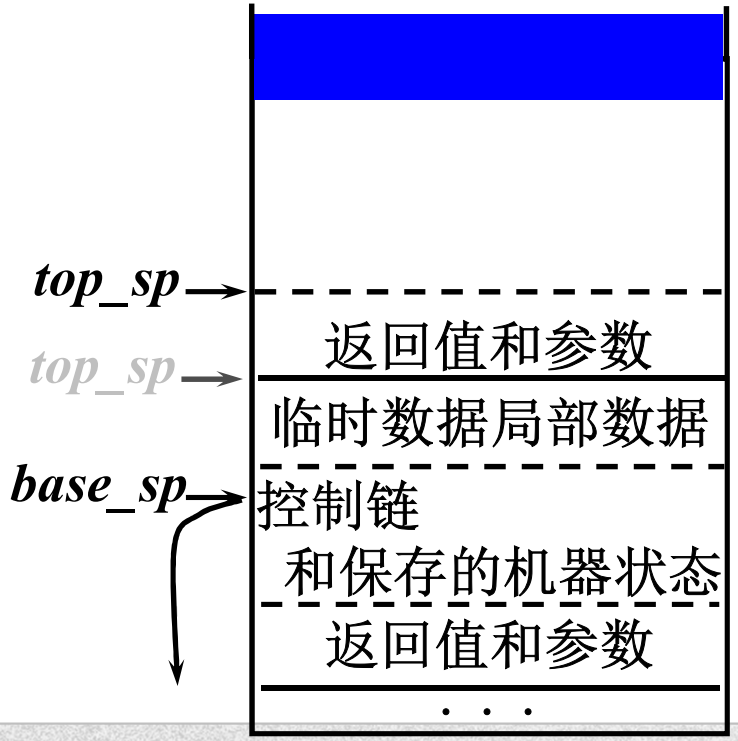
调用代码序列的例子

1、过程p调用过程q的调用序列



调用代码序列的例子

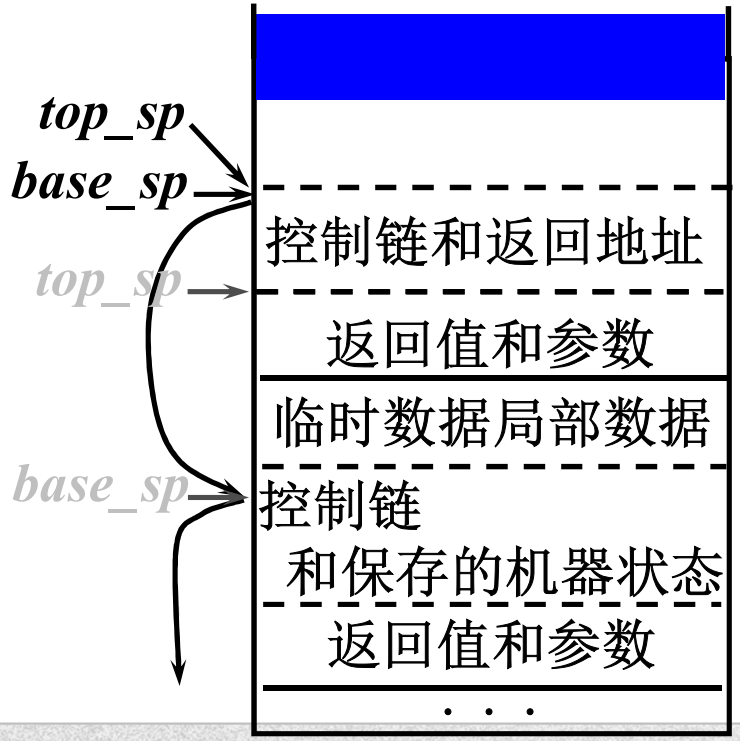
1、过程p调用过程q的调用序列



(1) p计算实参，依次放入栈顶，并在栈顶留出放返回值的空间。*top_sp*的值在此过程中被改变

调用代码序列的例子

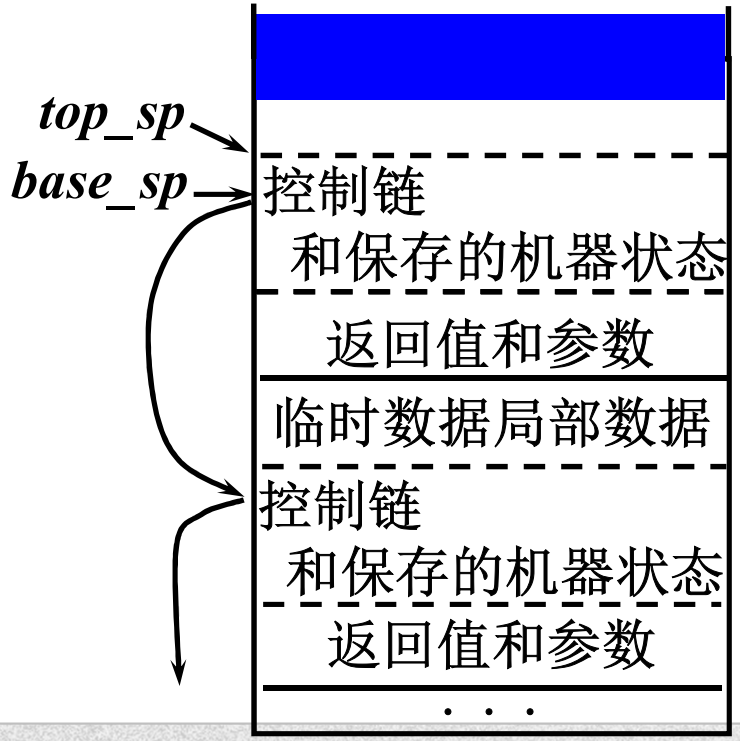
1、过程p调用过程q的调用序列



(2) p把返回地址和当前*base_sp*的值存入q的活动记录中，建立q的访问链，增加*base_sp*的值

调用代码序列的例子

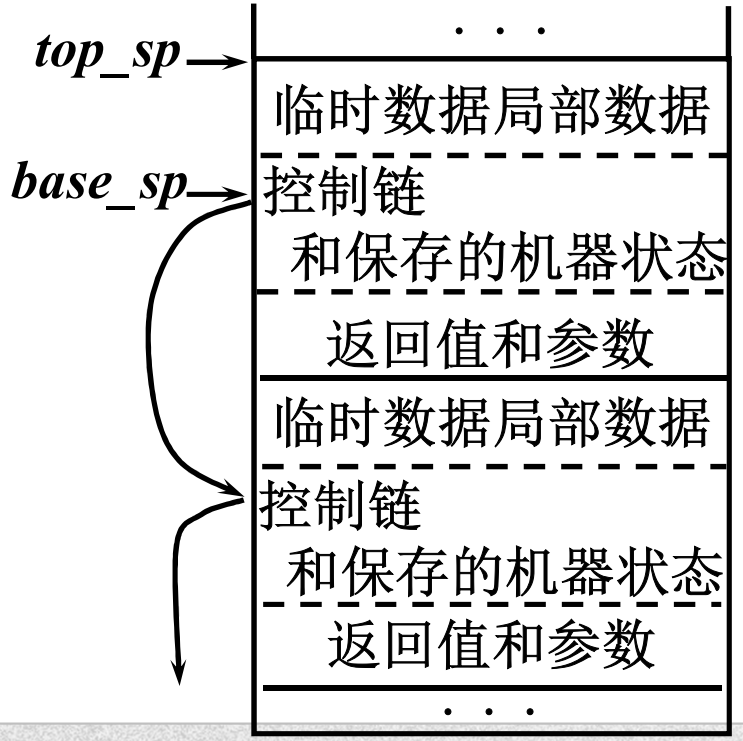
1、过程p调用过程q的调用序列



(3) q保存寄存器的值和其它机器状态信息

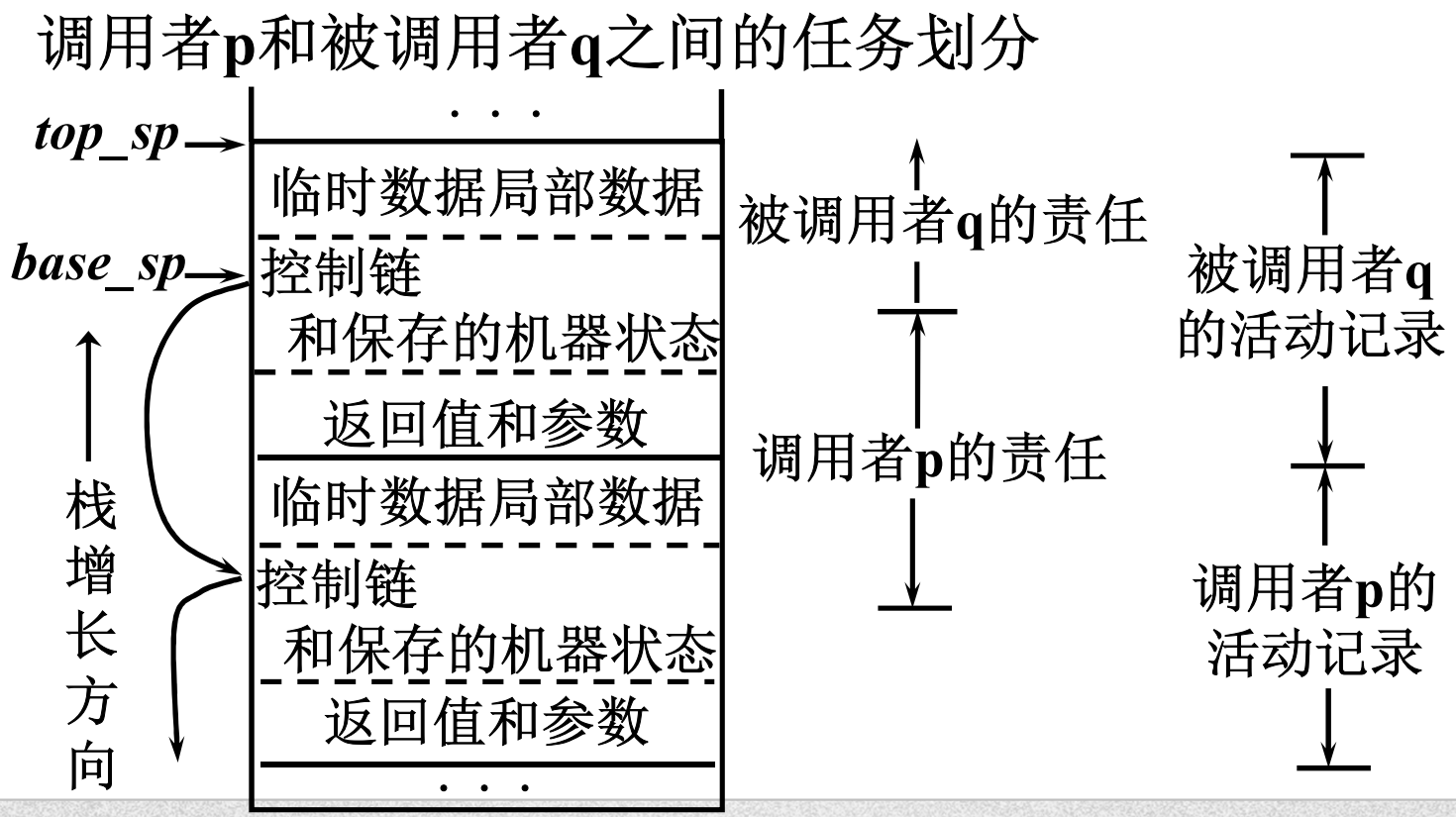
调用代码序列的例子

1、过程p调用过程q的调用序列



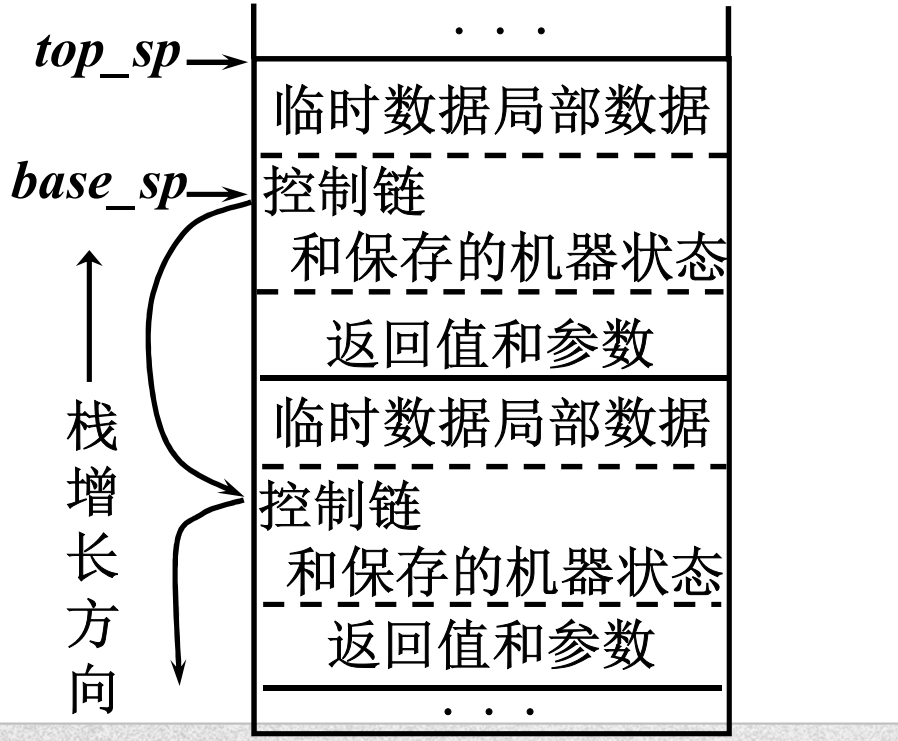
(4) q根据局部数据域和临时数据域的大小增加*top_sp*的值，初始化它的局部数据，并开始执行过程体

调用代码序列的例子


调用者p和被调用者q之间的任务划分
top_sp
base_sp
临时数据局部数据
控制链
和保存的机器状态
返回值和参数
临时数据局部数据
控制链
和保存的机器状态
返回值和参数
栈增长方向
被调用者q的责任
调用者p的责任
被调用者q的活动记录
调用者p的活动记录

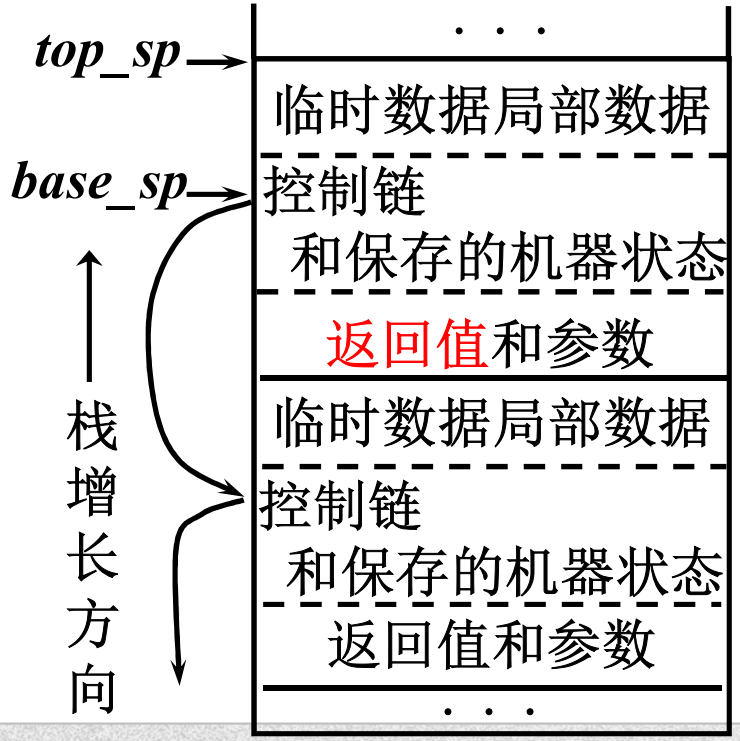
调用代码序列的例子

2、过程p调用过程q的返回序列



调用代码序列的例子

2、过程p调用过程q的返回序列

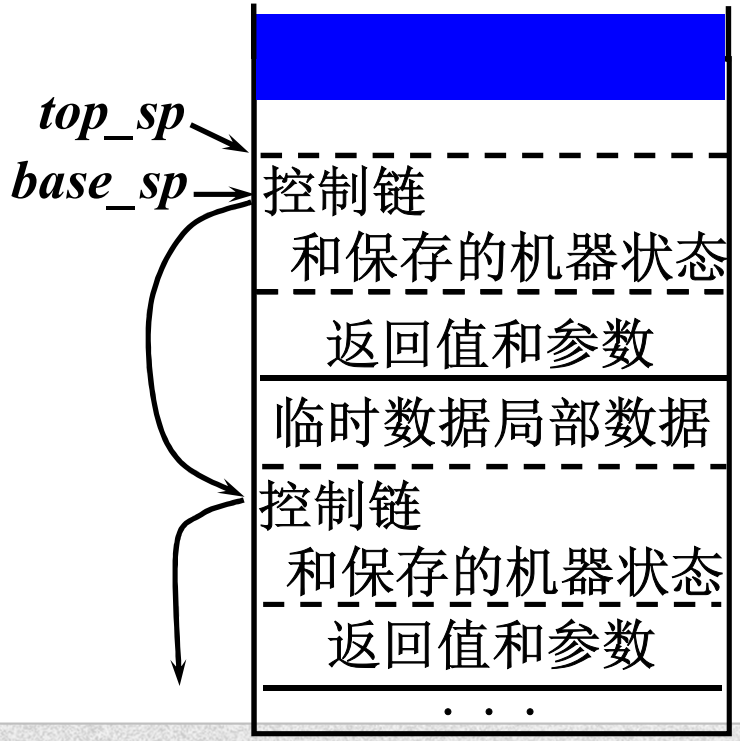


(1) q把返回值置入邻近p的活动记录的地方

参数个数可变场合难以确定存放返回值的位置，因此通常用寄存器传递返回值

调用代码序列的例子

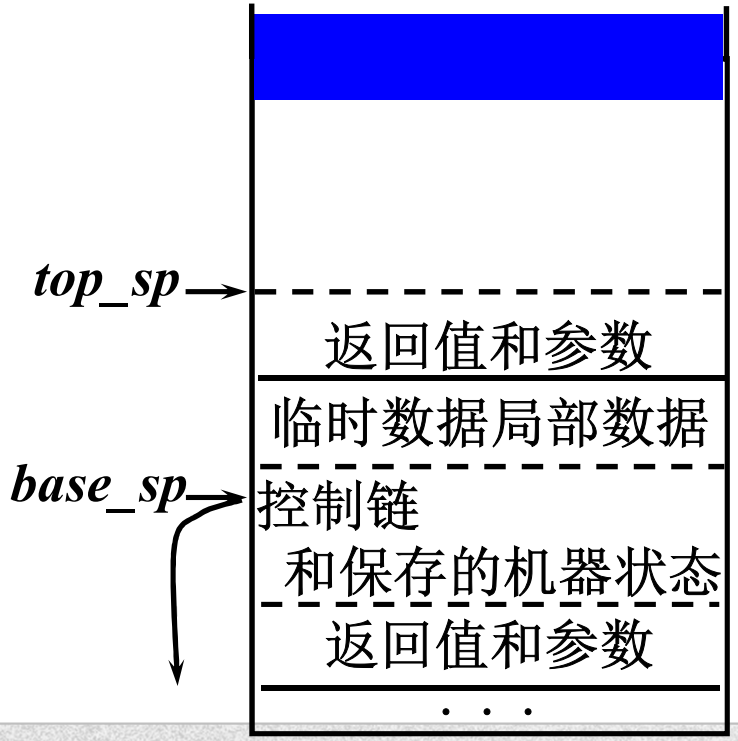
2、过程p调用过程q的返回序列



(2) q对应调用序列的步骤(4)，减小*top_sp*的值

调用代码序列的例子

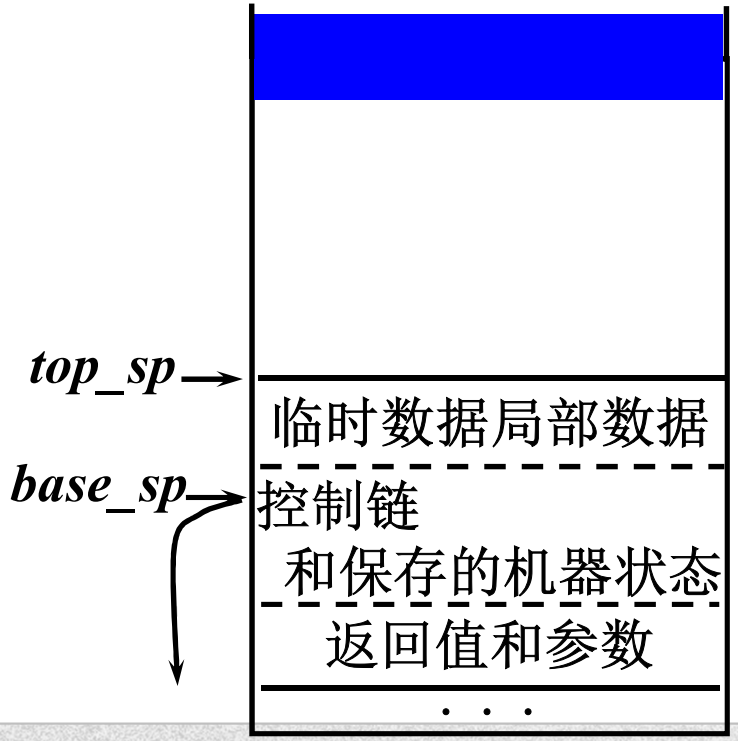
2、过程p调用过程q的返回序列



(3) q恢复寄存器(包括*base_sp*)和机器状态，返回p

调用代码序列的例子

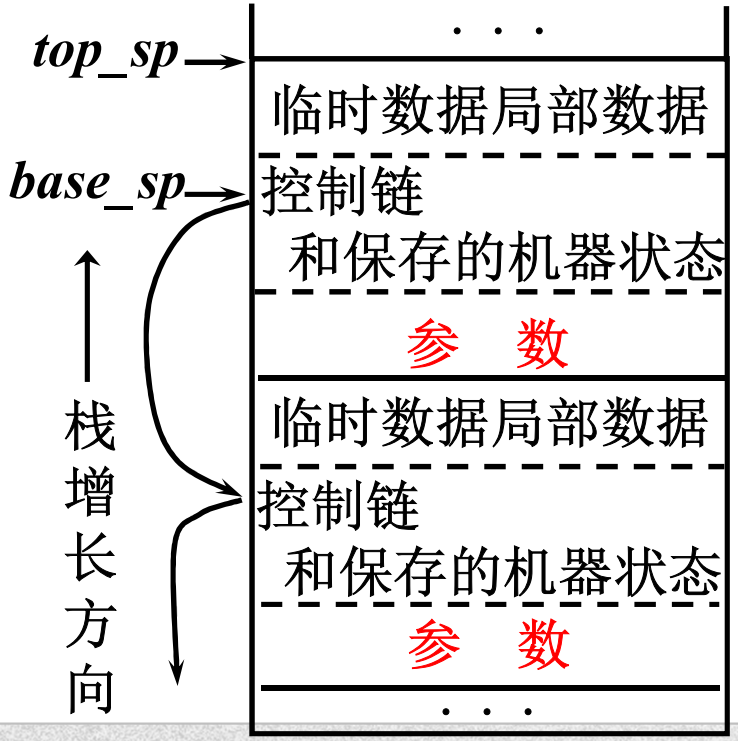
2、过程p调用过程q的返回序列



(4) p根据参数个数与类型和返回值类型调整*top_sp*，然后取出返回值

调用代码序列的例子

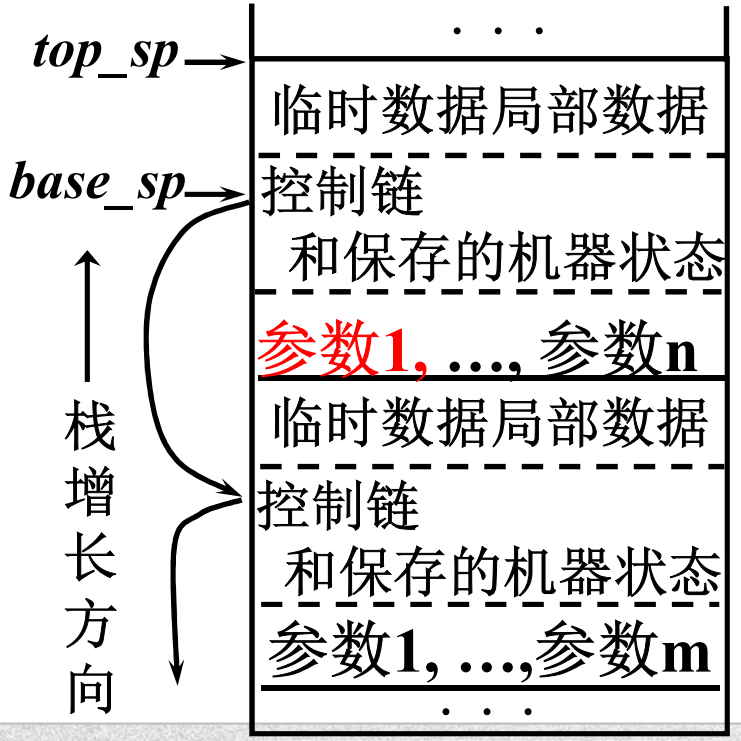
3、过程的参数个数可变的情况



(1) 函数返回值改成用寄存器传递

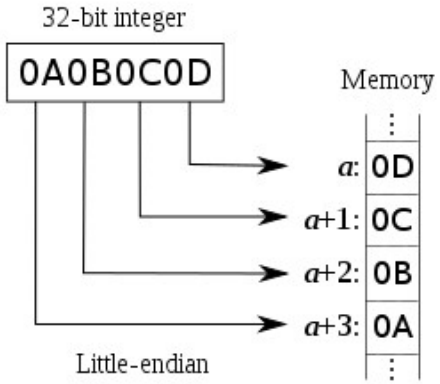
调用代码序列的例子

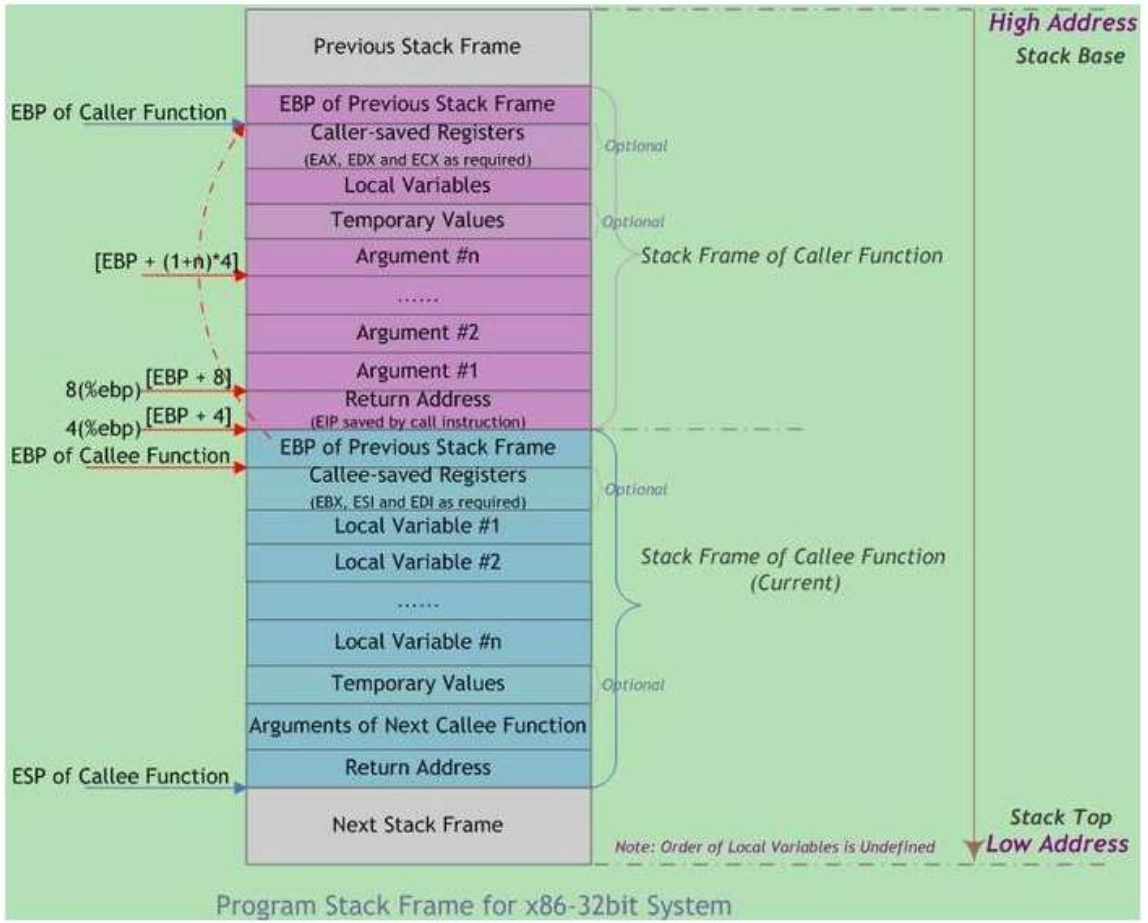
3、过程的参数个数可变的情况



(3) 从右向左压入参数，使得被调用函数能准确地知道第一个参数的位置

调用代码序列的例子





函数调用栈的可能内存布局

调用代码序列的例子

- **Stacksample.c**

stacksample.c

```c
//StackFrame.c
#include <stdio.h>
#include <string.h>

struct Strt{
    int member1;
    int member2;
    int member3;
};

#define PRINT_ADDR(x)     printf("&"#x" = %p\n", &x)
int StackFrameContent(int para1, int para2, int para3){
    int locVar1 = 1;
    int locVar2 = 2;
    int locVar3 = 3;
    int arr[] = {0x11,0x22,0x33};
    struct Strt tStrt = {0};
    PRINT_ADDR(para1); //若para1为char或short型，则打印para1所对应的栈上整型临时变量地址！
    PRINT_ADDR(para2);
    PRINT_ADDR(para3);
    PRINT_ADDR(locVar1);
    PRINT_ADDR(locVar2);
    PRINT_ADDR(locVar3);
    PRINT_ADDR(arr);
    PRINT_ADDR(arr[0]);
    PRINT_ADDR(arr[1]);
    PRINT_ADDR(arr[2]);
    PRINT_ADDR(tStrt);
    PRINT_ADDR(tStrt.member1);
    PRINT_ADDR(tStrt.member2);
    PRINT_ADDR(tStrt.member3);
    return 0;
}

int main(void){
    int locMain1 = 1, locMain2 = 2, locMain3 = 3;
    PRINT_ADDR(locMain1);
    PRINT_ADDR(locMain2);
    PRINT_ADDR(locMain3);
    StackFrameContent(locMain1, locMain2, locMain3);
    printf("[locMain1,2,3] = [%d, %d, %d]\n", locMain1, locMain2, locMain3);
    memset(&locMain2, 0, 2*sizeof(int));
    printf("[locMain1,2,3] = [%d, %d, %d]\n", locMain1, locMain2, locMain3);
    return 0;
}

```

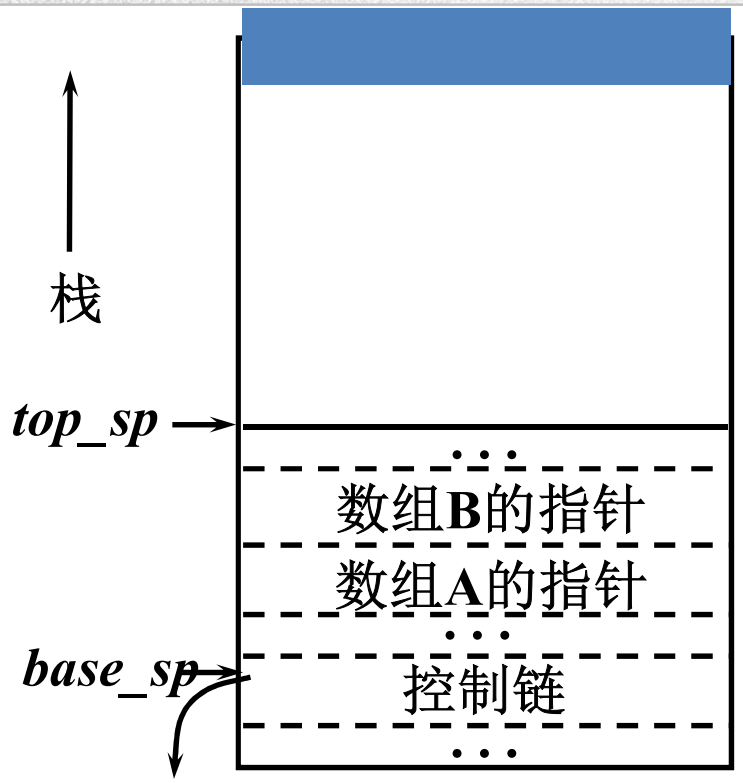
栈上可变长数据

活动记录的长度在编译时不能确定的情况

- 例：局部数组的大小要等到过程激活时才能确定

备注：　Java语言的实现是将它们分配在堆上

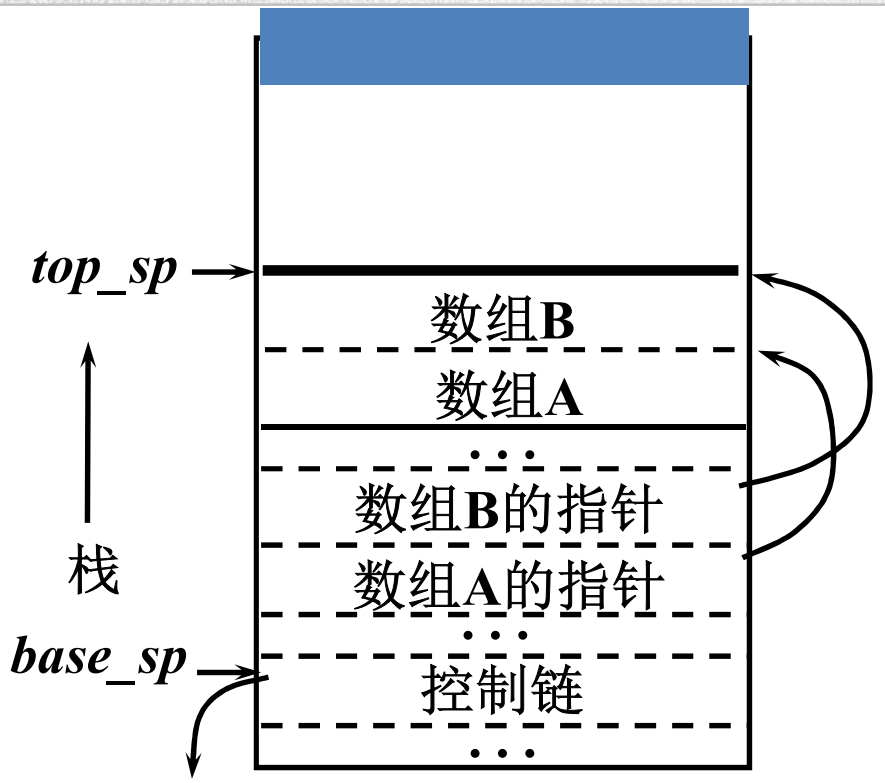
栈上可变长数据



(1) 编译时，
在活动记录中
为这样的数组
分配存放数组
指针的单元

访问动态分配的数组

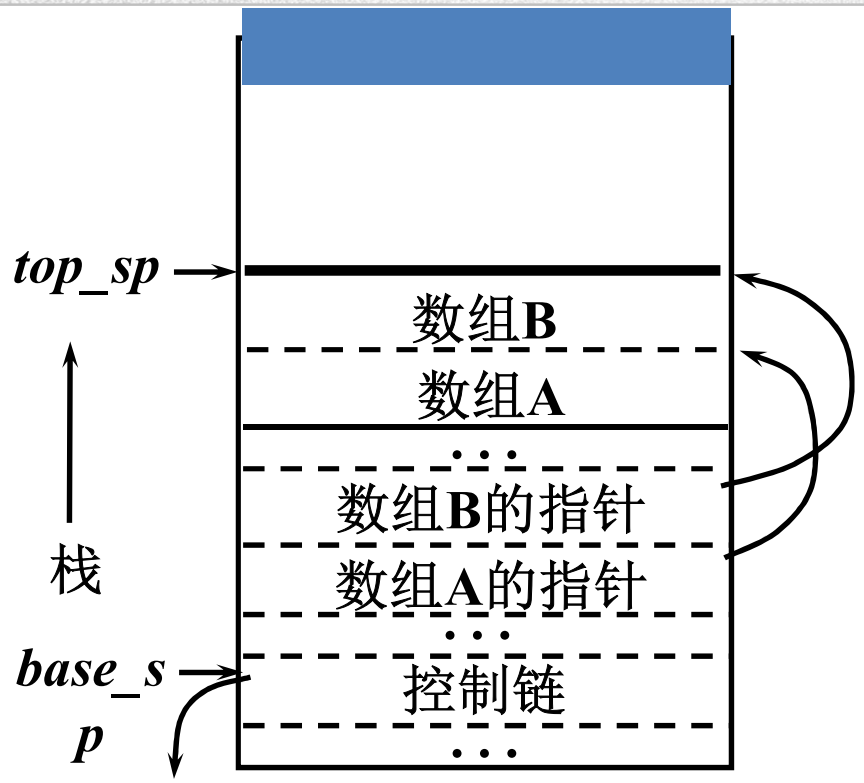
栈上可变长数据



(2) 运行时，这些指针指向分配在栈顶的数组存储空间

访问动态分配的数组

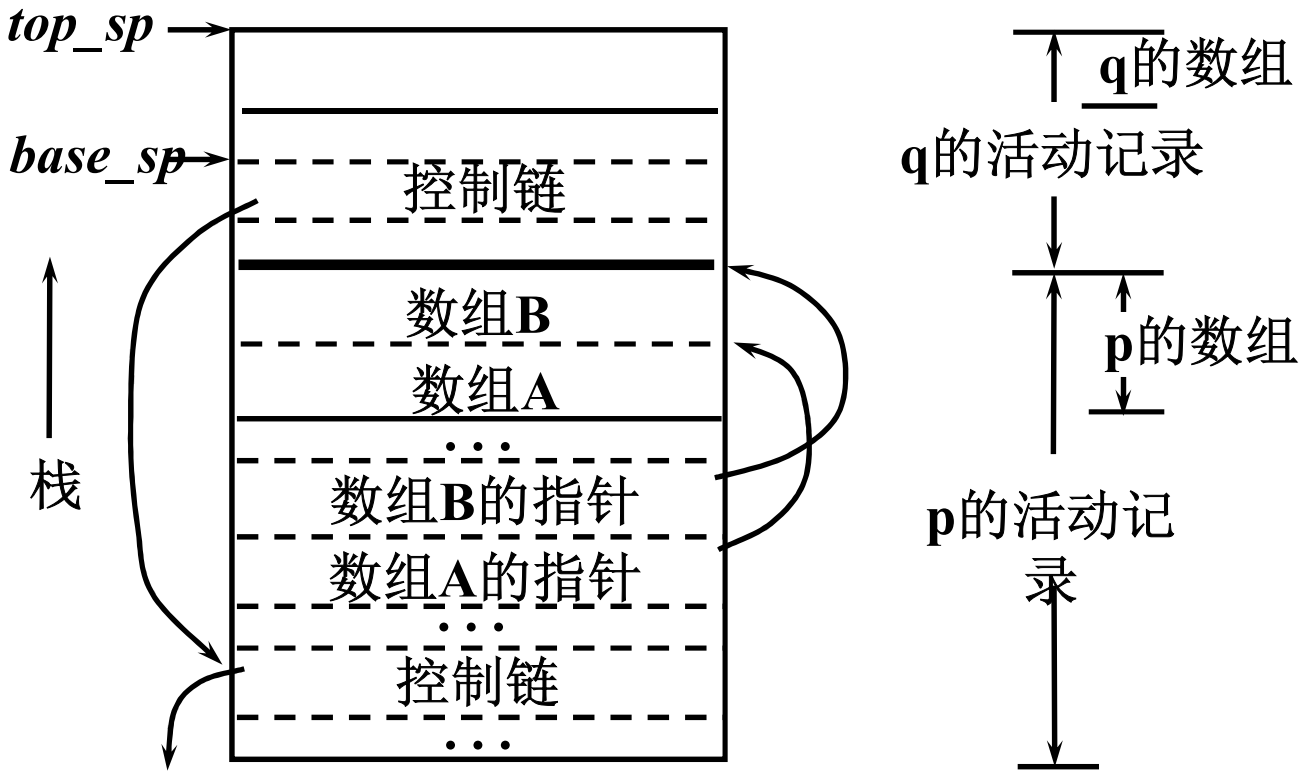
栈上可变长数据



(3) 运行时，对数组A和B的访问都要通过相应指针来间接访问

访问动态分配的数组

栈上可变长数据



访问动态分配的数组

全局栈式存储分配

悬空引用：引用某个已被释放的存储单元

全局栈式存储分配

悬空引用：引用某个已被释放的存储单元

例：main中引用p指向的对象

```
main( ) {                    |    int * dangle ( ) {
    int *q;                  |    int j = 20;
    q = dangle ( );          |    return &j;
}                            |    }
```

作业

- **教材P283:7.2.3**

栈中非局部数据的访问

作用域

- 所谓的作用域就是指某段程序文本代码
  - 一个声明起作用的那段程序文本区域，称为该声明的作用域
  - 静态作用域：声明的作用域是根据程序正文在编译时就确定的，有时也称为词法作用域 - C、PASCAL、ML
  - 动态作用域：程序中某个变量所引用的对象是在程序运行时刻根据程序的控制流信息来确定的 - APL 、Snobol和Lisp

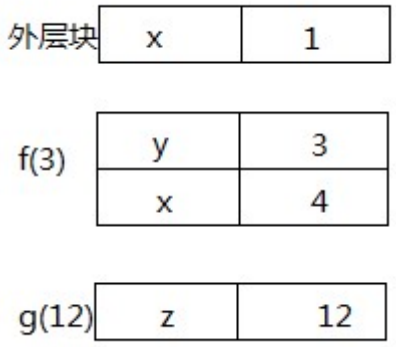
注意区分：静态存储分配，静态作用域

静态作用域

- **平坦结构，允许递归调用：C语言**

```
1 int x = 1;
2 int g(int z) { return x + z; }
3 int f(int y)
4 {
5     int x = y + 1;
6     return g(y*x);
7 }
8 f(3);
```

没有任何一个函数属于另一个函数
e.g. f 和 g是并列的



静态作用域

- **平坦结构，允许递归调用：C语言**
  - 每个函数能够访问的变量
    - 函数的局部变量：相对地址已知，且存放在当前活动记录，可以通过*base_sp*指针来访问
    - 函数非局部变量：即全局变量，在静态区，地址在编译时刻可知
  - 很容易将函数作为参数进行传递，也可以作为结果来返回
    - 参数中只需包括函数代码的开始地址。
    - 在函数中访问非局部变量的模式很简单，不需要考虑过程是如何激活的，即无须深入栈中取数据，无须访问链

静态作用域

- **嵌套定义，允许递归调用：PASCAL, ML**

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

静态作用域

- **嵌套层次(深度)**

嵌套深度是正文概念，可以根据源程序静态地确定：不内嵌于任何其他过程中的过程，嵌套深度为1；嵌套在深度为i的过程中的过程，深度为i+1

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

深度为1

sort

深度为2

readArray，exchange，quicksort

深度为3

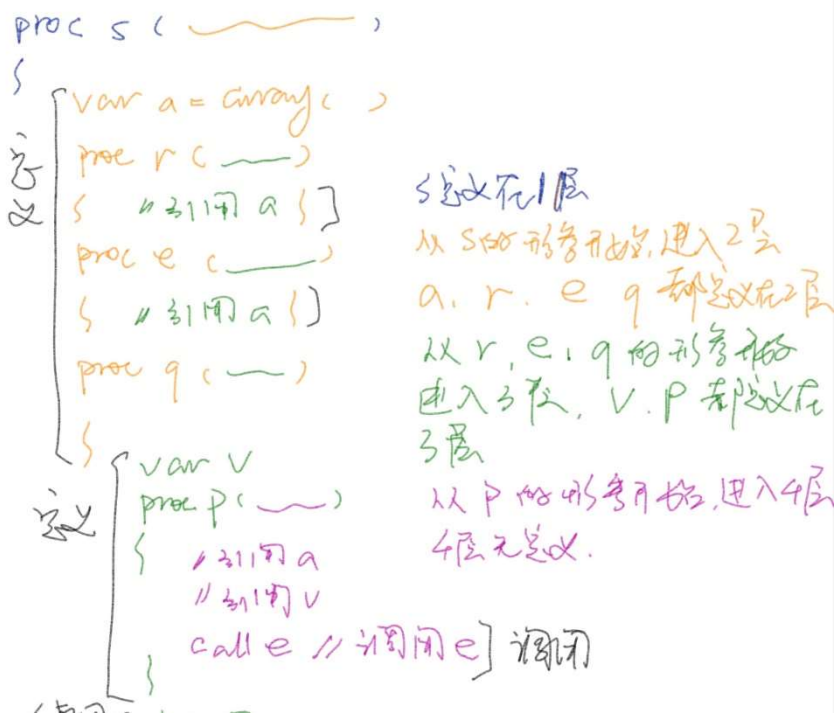
partition

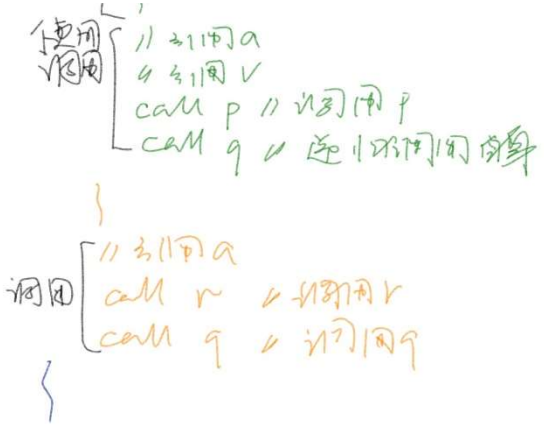
允许嵌套定义的名字作用域规则（静态作用域）

- **最内嵌套作用域规则：**
  - 由一个声明引进的标识符在这个声明所在的作用域里可见，而且在其内部嵌套的每个作用域里也可见，除非它被嵌套于内部的对同名标识符的另一个声明所掩盖。
    - fun f 可以访问其内部定义的所有名字：use a variable, call a nested function
    - 可以访问包围fun f 的任意外层过程所定义的名字（使用变量，调用函数）

允许嵌套定义的名字作用域规则（静态作用域）

- **从当前最内层作用域开始查找**
  - 如找到，则可以找到该标识符所引用的对象；
  - 否则，到直接的外层作用域里去查找，并继续向外顺序地检查外层作用域，直到到达程序的最外嵌套层次，也就是全局对象声明所在的作用域
  - 如果在所有层次上都没有找到有关声明，那么这个程序就有错误。


proc s ( )
{
定义
var a = array( )
proc r ( )
{ //引用 a }
proc e ( )
{ //引用 a }
proc q ( )
{
定义
var v
proc p ( )
{
//引用 a
//引用 v
call e //调用 e
调用
s定义在1层
从s的{开始,进入2层
a, r, e q都定义在2层
从r, e, q的{开始
进入3层, v, p都定义在
3层
从p的{开始,进入4层
4层无定义.


使用
调用
//引用 a
//引用 v
call p //调用 p
call q // 这里的调用语句
{
调用
//引用 a
call r //调用 r
call q //调用 q
{

允许嵌套定义的名字作用域规则（静态作用域）

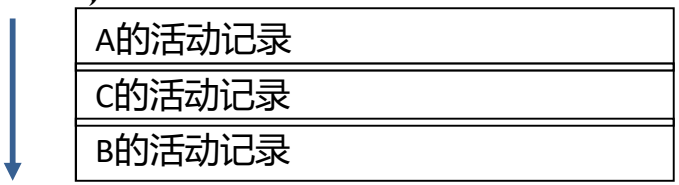
- **困难在哪里?**

注意区别：嵌套定义 v.s. 嵌套调用
即嵌套定义与活动记录栈位置没有对应关系

例：PASCAL中，如果过程A的声明中包含了过程B的声明，那么B可以使用在A中声明的变量。当B的代码运行时，如果它使用的是A中的变量。那么这个变量指向运行栈中最上层的同名变量。但是，我们不能通过嵌套层次直接得到A的活动记录的相对位置，必须通过访问链访问

```
void A()
{
   int x,y;
   void  B() { int b; x = b+y; }
   void  C(){B();}
   C();
   B();
}
```

当A调用C，C又调用B时：



当A直接调用B时：

关键问题：B和C都需要(可以)访问直系外层函数A的名字(使用变量或调用函数)，如何找到A在栈里的活动记录?

解决方法-访问链

- **使用访问链进行定位**
  - 当被调用过程需要其他地方的某个数据时需要使用访问链进行定位
  - 如果过程p在声明时嵌套在过程q的声明中，那么p的活动记录中的访问链指向最上层的q的活动记录
  - 从栈顶活动记录开始，访问链形成了一个链路，嵌套深度沿着链路逐一递减

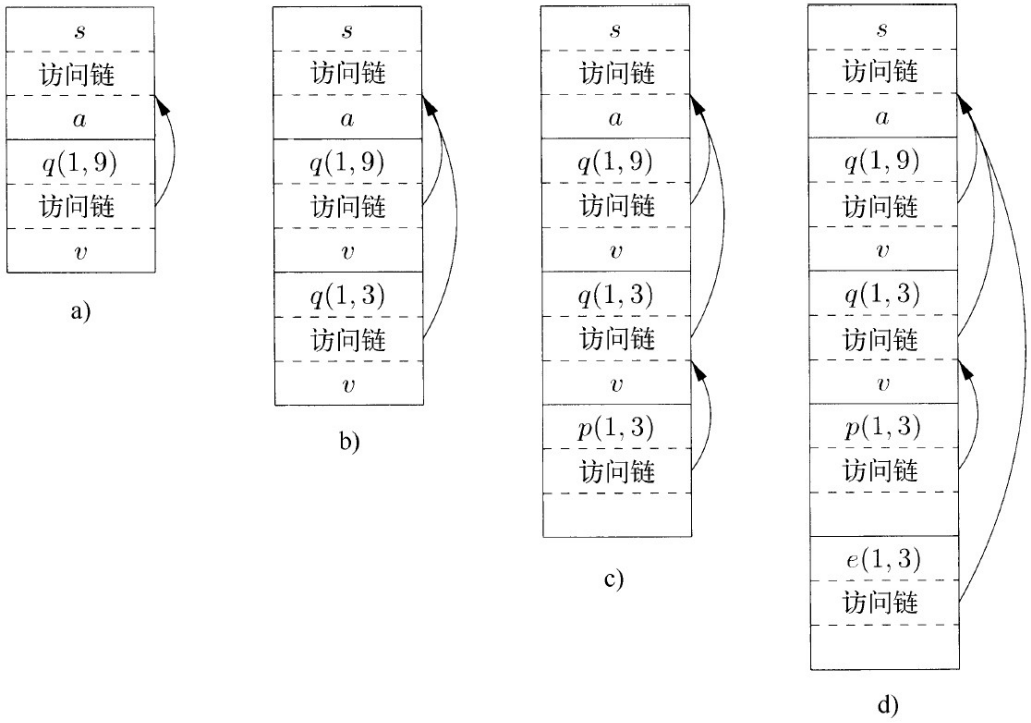
解决方法-访问链

- **设深度为n_p的过程p访问变量x，而变量x在深度为n_q的过程中声明，那么**
  - n_p-n_q在编译时刻已知
  - 从当前活动记录出发，沿访问链前进n_p-n_q次找到的活动记录中的x就是要找的变量位置
  - x相对于这个活动记录的偏移量在编译时刻已知

解决方法-访问链

- **图解如何实现的访问**

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ··· ;
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

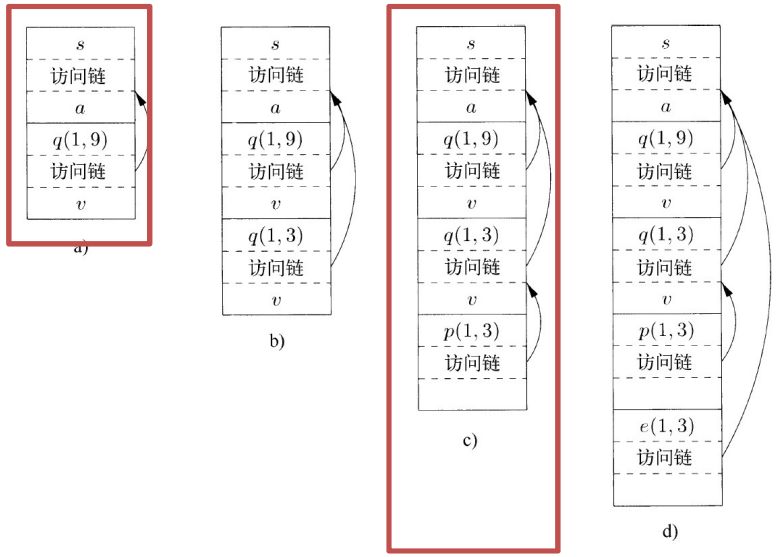


解决方法-访问链

- **如何构造并维护访问链，以过程q调用过程p为例说明**
  - Case 1：p的深度大于q：根据作用域规则，p必然在q中直接定义；那么p的访问链指向当前活动记录，

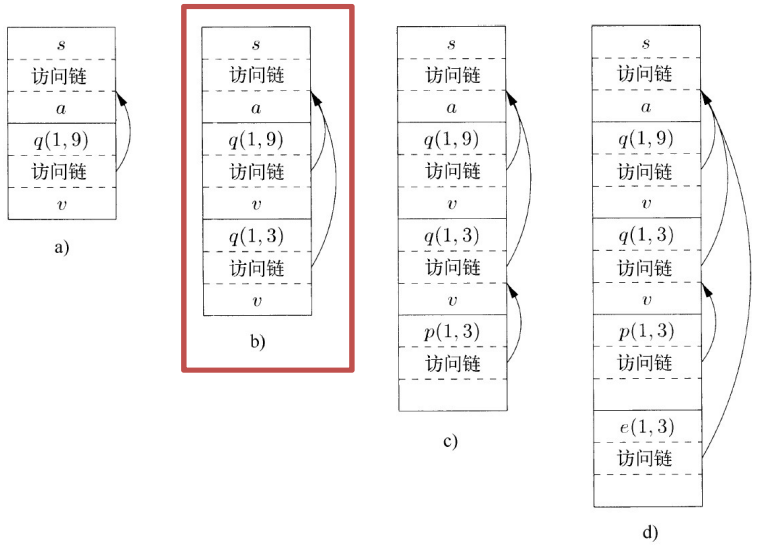
对应于规则中“fun f 可以访问其内部定义的所有名字use a variable, call a nested function”

此时，定义的嵌套关系与调用的嵌套关系一致



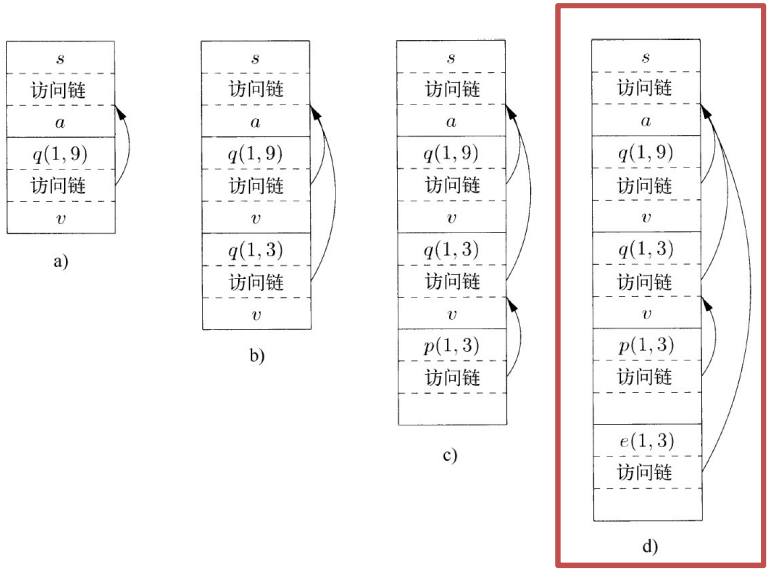
解决方法-访问链

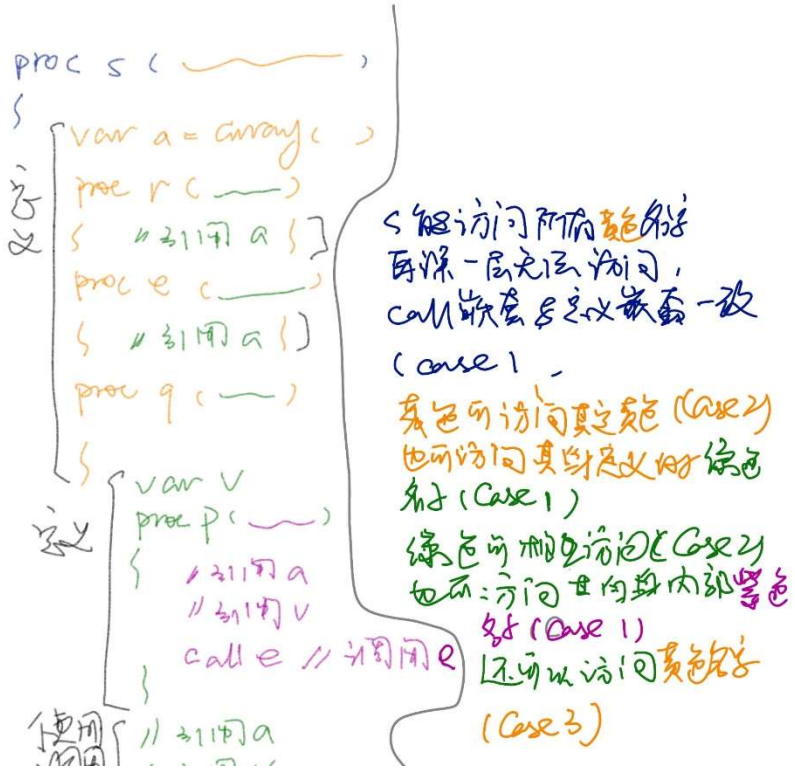
- **如何构造并维护访问链，以过程q调用过程p为例说明**
  - Case 2：递归调用（p=q）：新活动记录的访问链等于当前记录的访问链
    - q(1,9)调用q(1,3)



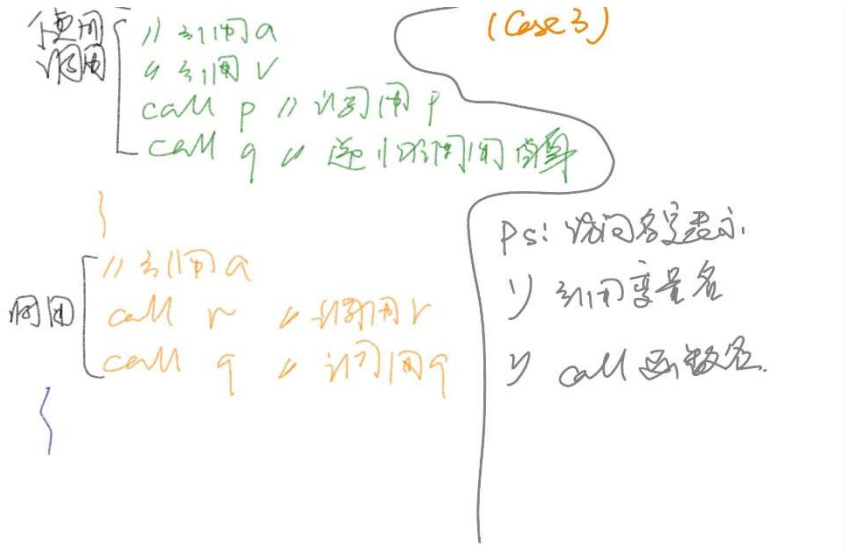
解决方法-访问链

- **如何构造并维护访问链，以过程q调用过程p为例说明**
  - Case 3：p的深度小于等于q的深度：此时必然有过程r，p直接在r中定义，而q嵌套在r中；p的访问链指向栈最高的r的活动记录
    - p调用exchange





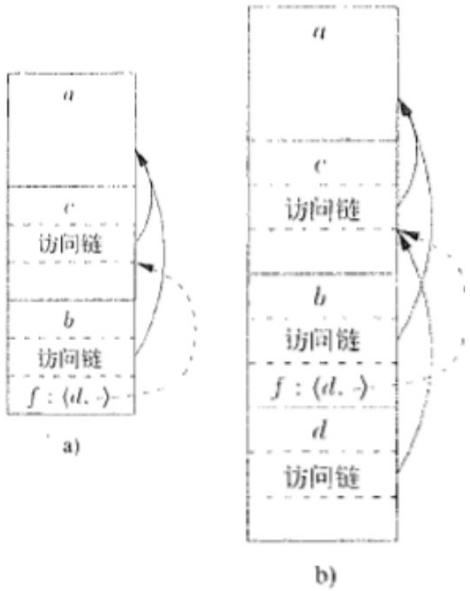
S能访问所有的标识符
再深一层无法访问，
call嵌套多定义嵌套一次
(case 1)
黄色的访问直接能(case 2)
也可访问其对应的绿色
名字(case 1)
绿色的相互访问(case 2)
也可：访问其内部内部紫色
名字(case 1)
还可以访问黄色的
(case 3)



解决方法-访问链

- **过程指针型参数的访问链**
  - 在传递过程指针参数时，过程型参数中不仅包含过程的代码指针，还包括正确的访问链

```
fun a(x) =
    let
        fun b(f) =
            ··· f ··· ;
        fun c(y) =
            let
                fun d(z) = ···
            in
                ··· b(d) ···
            end
    in
        ··· c(1) ···
    end;
```



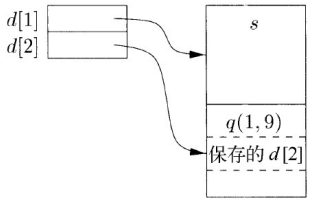
实现方法-display表

- **用访问链访问数据时，访问开销和嵌套深度差有关**
- **使用显示表可以提高效率，访问开销为常量**
  - 数组d为每个嵌套深度保留一个指针
    - 指针d[i]指向栈中最高的、嵌套深度为i的活动记录
    - 如果程序p中访问嵌套深度为i的过程q中声明的变量x，那么d[i]直接指向相应的（必然是q的）活动记录
    - 注意：i在编译时刻已知

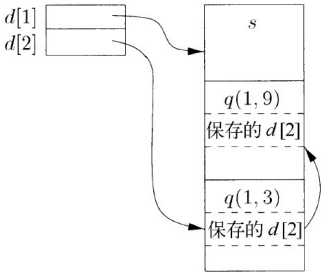
实现方法-display表

- **显示表维护例子**

调用过程p时，在p的活动记录中保存d[np]的值，并将d[np]设置为当前活动记录。从p返回时，恢复d[np]的值。

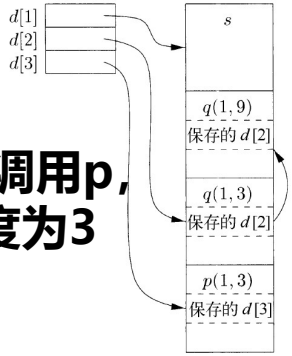


a)



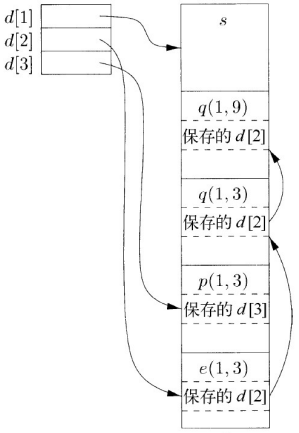
b)

q(1,9)调用q(1,3)时，q的深度为2



c)

q(1,3)调用p，p的深度为3



d)

q调用e
e的深度为2

参数调用

基本概念

- 参数传递：调用函数和被调用函数的传递参数的约定 (Convention)，现代编译器利用寄存器进行参数传递，以减少和内存的通讯，提高调用过程的执行效率.
- R-value：一个表达式的值，可以出现在赋值式右边的值，如：``x + 3''，表达式的计算结果就是 ``x + 3'' 的 R-value，R-value 不一定有存储空间与之对应.
- L-value：存储空间，其保存的内容是可以修改并可访问的，可以在赋值式左边出现；如：C 语言的赋值表达式要求等号的左边一定是 L-value，

```
int a [10], i;
*a = 1;
a = &i; /* Error: L-value required */
```

- 任何一个表达式一定有 R-value，但是不一定有 L-value，如：上例中数组名 a 在表达式中出现将转换为指针常量. 指针常量是没有 L-value的. 但是，对指针的引用 *a 是有 L-value的，即是 *a 所指的存取空间；
- 主要的参数传递方式：call by value, call by reference, call by value result (copy restore), call by name.

- 形参和实参相关联的几种方法
  - 传值调用
  - 引用调用
  - 复制-恢复
  - 传名调用

Call by value

- 实参的右值传给被调用过程
- 值调用可以如下实现
  - 把形参当作所在过程的局部名看待，形参的存储单元在该过程的活动记录中。
  - 调用过程计算实参，并把右值放入形参的存储单元中。

Call by value

Call by value

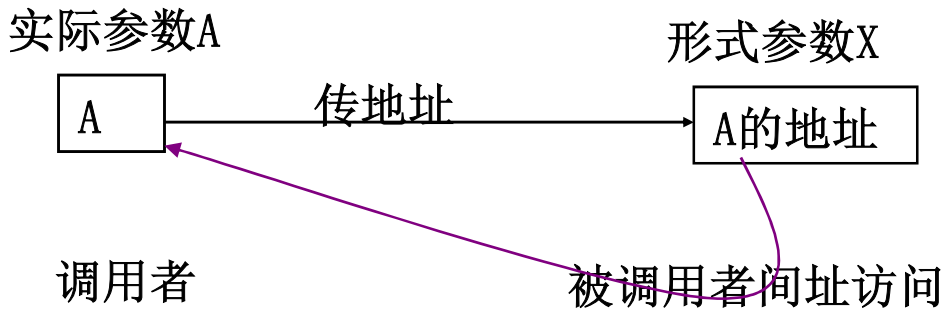
- 形参如同过程中的局部变量一样处理，即函数在被调用时，在 AR 中有存储空间.
- 调用函数首先对实参进行计值，并将其 R-value，保存到被调用函数的形参对应的 AR 单元上.
- 在被调用函数中对形参的任何修改不会影响到调用函数 AR 中的实参.
- 使用情况：Pascal 语言没有用 var 定义的形参；C 和 Java 语言唯一的参数传递方式；C++ 语言没有用 & 定义的形参.

Example

```
void f(int *p)
{
  *p = 5;
  p = NULL;
}
main()
{
  int *q = (int *)
    malloc(sizeof(int));
  *q=0;
  f(q);
}
/* f返回后，q所指的内存单元没有改变，
   但是该内存单元的R-value修改为5 */
```

Call by reference

- 实参的左值传给被调用过程
- 引用调用可以如下实现：
  - 把实参的左值放入形参的存储单元。
  - 在被调用过程的目标代码中，任何对形参的引用都是通过传给该过程的指针来间接引用实参的。



Call by reference

Call by reference

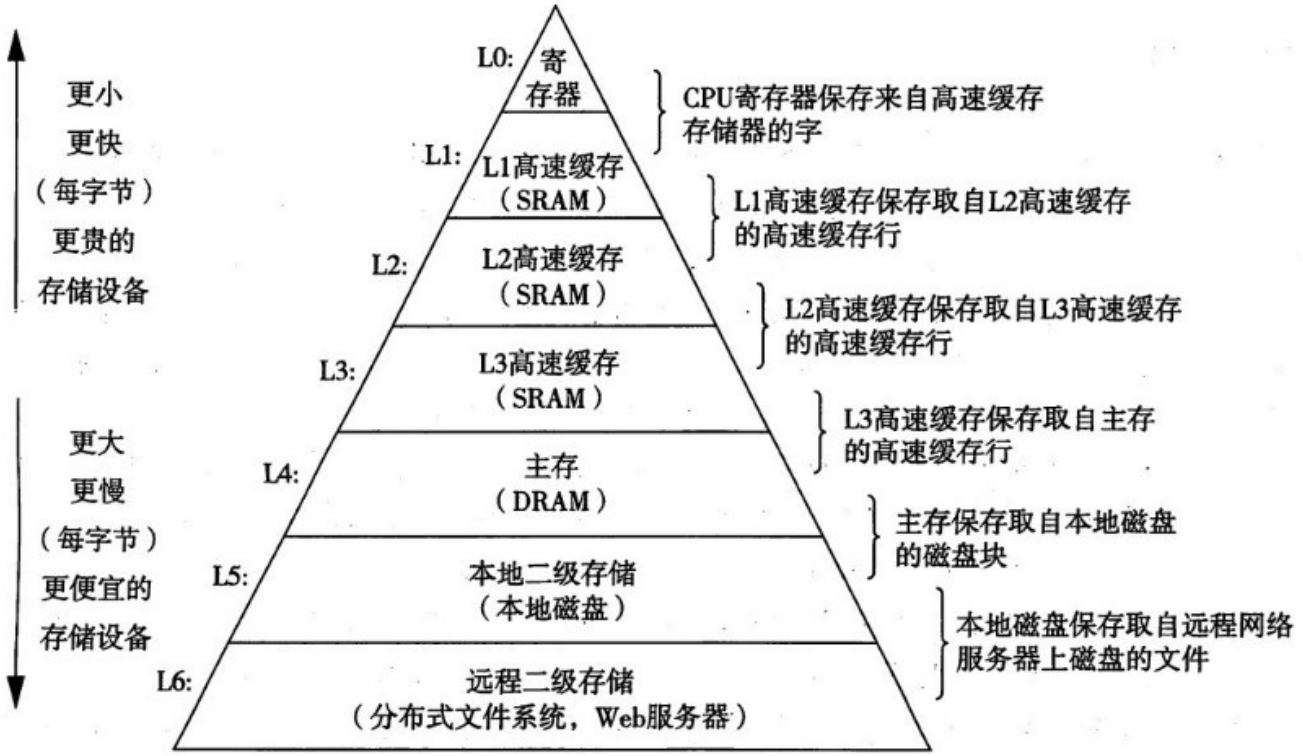
- 如果实参是变量名，直接将将该变量的 L-value 拷贝到被调用函数的 AR 中.
- 如果实参是表达式，没有 L-value，如：``a + 2'' 或 ``2''，则先对表达式计值，再将计算结果保存到新创建的存储空间中，将该存储空间的地址传给被调用函数.
- 在被调用函数中对形参的任何修改将会影响到调用函数 AR 中的实参.
- 使用情况：Pascal 语言用 var 定义的形参. C++ 语言用 & 定义的形参. Fortran. Java 语言的对象等.

Example

```
void swap(int *x, int *y)
{
  int tmp;
  tmp = *x; *x = *y; *y = tmp;
}
main()
{
  int a = 1, b = 2;
  swap(&a, &b);
  printf("a now %d, b now %d\n",
    a, b);
}
/* C利用指针实现call by reference */
```

堆管理

计算机内存分层


更小
更快
（每字节）
更贵的
存储设备
更大
更慢
（每字节）
更便宜的
存储设备
L0:
寄存器
CPU寄存器保存来自高速缓存存储器的字
L1:
L1高速缓存（SRAM）
L1高速缓存保存取自L2高速缓存的高速缓存行
L2:
L2高速缓存（SRAM）
L2高速缓存保存取自L3高速缓存的高速缓存行
L3:
L3高速缓存（SRAM）
L3高速缓存保存取自主存的高速缓存行
L4:
主存（DRAM）
主存保存取自本地磁盘的磁盘块
L5:
本地二级存储（本地磁盘）
本地磁盘保存取自远程网络服务器上磁盘的文件
L6:
远程二级存储（分布式文件系统，Web服务器）

计算机内存分层

- 现代计算机都设计成程序员不用关心内存子系统的细节就可以写出正确的程序
- 程序的效率不仅取决于被执行的指令数，还取决于执行每条指令需要多长时间
- 执行一条指令的时间区别非常可观
- 差异源于硬件技术的基本局限：构造不了大容量的高速存储器
- 数据以块（缓存行、页）为单位在相邻层次之间进行传送
- 数据密集型程序可从恰当利用内存子系统中获益

程序局部性

- **大多数程序的大部分时间在执行一小部分代码，并且仅涉及一小部分数据**
- **时间局部性**
  - 程序访问的内存单元在很短的时间内可能再次被程序访问
- **空间局部性**
  - 毗邻被访问单元的内存单元在很短的时间内可能被访问

程序局部性

- 即使知道哪些指令会被频繁执行，最快的缓存也可能没有大到足以把它们同时放在其中，因此必须动态调整最快缓存的内容
- 把最近使用的指令保存在缓存是一种较好的最优化利用内存分层的策略
- 改变数据布局或计算次序也可以改进程序数据访问的时间和空间局部性

堆 管 理

- **堆用来存放生存期不确定的数据**
  - C++和Java允许程序员用new创建对象，它们的生存期没有被约束在创建它们的过程活动的生成期之内
  - 实现内存回收是内存管理器的责任
- **堆空间的回收有两种不同方式**
  - 程序显式释放空间：new生成的对象可以生存到被delete为止，malloc申请的空间生存到被free为止 （C/C++）
  - 垃圾收集器自动收集（Java）

堆 管 理

- **内存管理器把握的基本信息是堆中空闲空间**
  - 分配/回收堆区空间的子系统
  - 根据语言而定
    - C、C++需要手动回收空间
    - Java可以自动回收空间（垃圾收集）

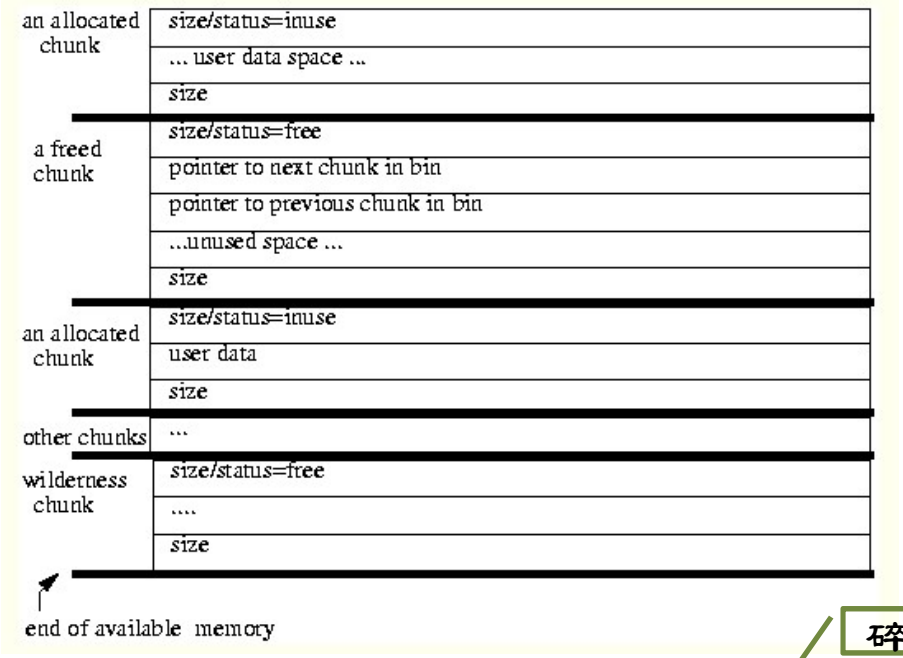
存储管理器

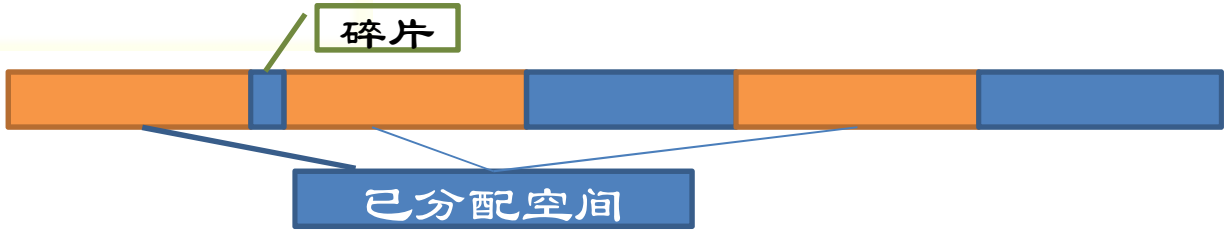
- **基本功能**
  - 分配：为每个内存请求分配一段连续的、适当大小的堆空间
    - 首先从空闲的堆空间分配
    - 如果不行则从操作系统中获取内存、增加堆空间
  - 回收：把被回收的空间返回空闲空间缓冲池，以满足其他内存需求
- **评价存储管理器的特性**
  - 空间效率：使程序需要的堆空间最小，即减小碎片
  - 程序效率：充分运用内存系统的层次，提高效率
  - 低开销：使分配/收回内存的操作尽可能高效

堆空间的碎片问题

- 随着程序分配/回收内存，堆区逐渐被割裂成为若干空闲存储块（窗口，hole）和已用存储块的交错
- 分配一块内存时，通常是把一个窗口的一部分分配出去，其余部分成为更小的块
- 回收时，被释放的存储块被放回缓冲池。通常要把连续的窗口接合成为更大的窗口

堆空间的碎片问题

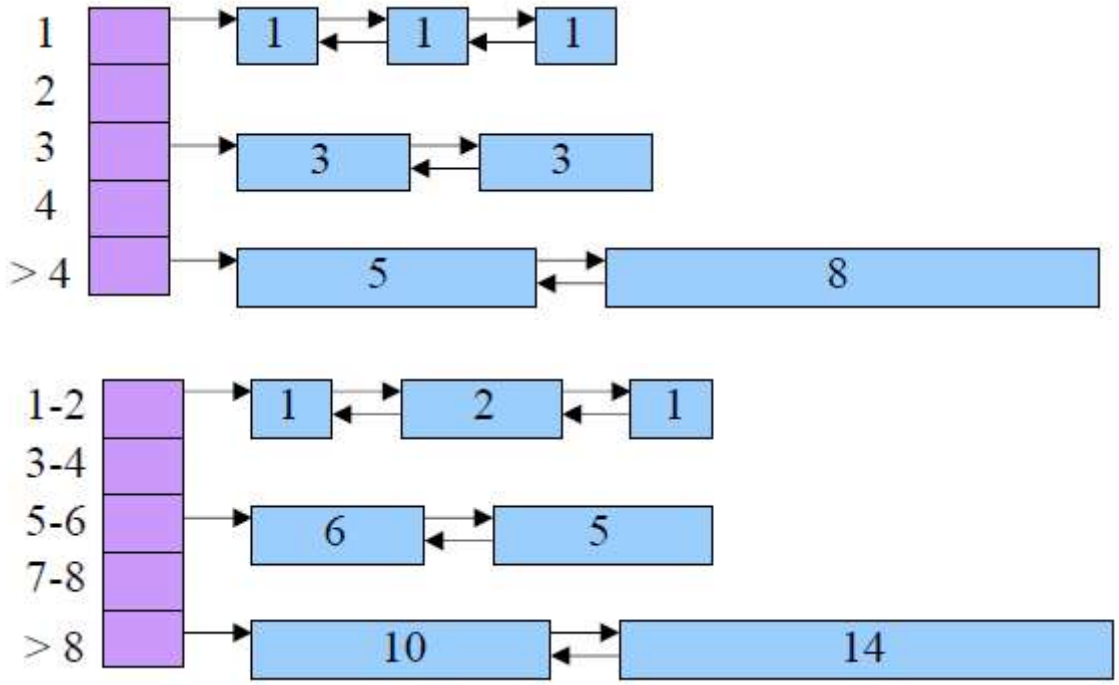

an allocated chunk
size/status=inuse
... user data space ...
size
a freed chunk
size/status=free
pointer to next chunk in bin
pointer to previous chunk in bin
...unused space ...
size
an allocated chunk
size/status=inuse
user data
size
other chunks
...
wilderness chunk
size/status=free
....
size
end of available memory


碎片
已分配空间

堆空间分配方法

- **Best-Fit**
  - 总是将请求的内存分配在满足请求的最小的窗口中
  - 好处：可以将大的窗口保留下来，应对更大的请求
- **First-Fit**
  - 总是将对象放置在第一个能够容纳请求的窗口中
  - 放置对象时花费时间较少，但是总体性能较差
  - 但是first-fit的分配方法通常具有较好的数据局部性
    - 同一时间段内的生成的对象经常被分配在连续的空间内

有效实现best-fit

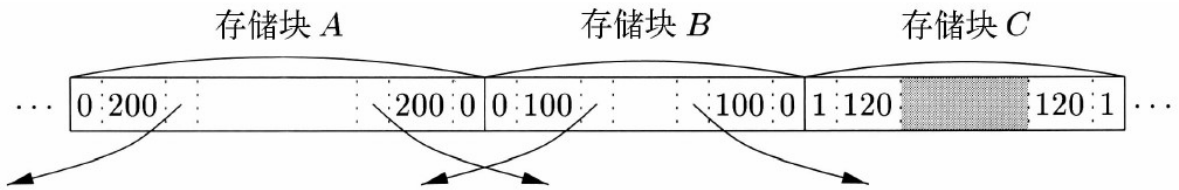


管理和接合空闲空间

- **当回收一个块时，可以把这个块和相邻的块接合起来，构成更大的块**
- **支持相邻块接合的数据结构**
  - 边界标记：在每一块存储块的两端，分别设置一个free/used位；相邻的位置上存放字节总数
  - 双重链接的、嵌入式的空闲块列表：列表的指针存放在空闲块中、用双向指针的方式记录了有哪些空闲块

例子

- **相邻的存储块A、B、C**
  - 当回收B时，通过对free/used位的查询，可以知道B左边的A是空闲的，而C不空闲。
  - 同时还可以知道A、B合并为长度为300的块
  - 修改双重链表，把A替换为A、B接合后的空闲块
- **注意：双重链表中一个结点的前驱并不一定是它邻近的块**



处理手工存储管理

- **两大问题**
  - 内存泄露：未能删除不可能再被引用的数据
  - 悬空指针引用：引用已被删除的数据
- **其他问题**
  - 空指针访问/数组越界访问
- **解决方法**
  - 自动存储管理
  - 正确的编程模式

正确的编程模式（1）

- **对象所有者（Object ownership）**
  - 每个对象总是有且只有一个所有者（指向此对象的指针）；只有通过Owner才能够删除这个对象
  - 当Owner消亡时，这个对象要么也被删除，要么已经被传递给另一个owner
    - 语句v=new ClassA；创建的对象的所有者为v
    - 即将对v进行赋值的时刻（v的值即将消亡）
      - 要么v已经不是它所指对象的所有者；比如g=v可以把v的ownership传递给g
      - 要么需要在返回/赋值之前，执行delete v操作
  - 编程时需要了解各个指针在不同时刻是否owner
  - 防止内存泄漏，避免多次删除对象。不能解决悬空指针问题

正确的编程模式（2）

- **引用计数**
  - 每个动态分配的对象附上一个计数：记录有多少个指针指向这个对象
  - 在赋值/返回/参数传递时维护引用计数的一致性
  - 在计数变成0之时删除这个对象
  - 可以解决悬空指针问题；但是在递归数据结构中仍然可能引起内存泄漏
  - 需要较大的运行时刻开销
- **基于区域的分配**
  - 将一些生命期相同的对象分配在同一个区域中
  - 整个区域同时删除

垃圾回收

垃圾回收

- **垃圾**
  - 狭义：不能被引用（不可达）的数据
  - 广义：不需要再被引用的数据
- **垃圾回收：自动回收不可达数据的机制，解除了程序员的负担**
- **使用的语言**
  - Java、Perl、ML、Modula-3、Prolog、Smalltalk

垃圾回收器的设计目标

- **基本要求：**
  - 语言必须是类型安全的：保证回收器能够知道数据元素是否为一个指向某内存块的指针
  - 类型不安全的语言：C，C++
- **性能目标**
  - 总体运行时间：不显著增加应用程序的总运行时间
  - 空间使用：最大限度地利用可用内存
  - 停顿时间：当垃圾回收机制启动时，可能引起应用程序的停顿。这个停顿应该比较短
  - 程序局部性：改善空间局部性和时间局部性

可达性

- **直观地讲，可达性就是指一个存储块可以被程序访问到**
- **根集：不需要指针解引用就可以直接访问的数据**
  - Java：静态成员、栈中变量
- **可达性**
  - 根集的成员都是可达的
  - 对于任意一个对象，如果指向它的一个指针被保存在可达对象的某字段中、或数组元素中，那么这个对象也是可达的
- **性质**
  - 一旦一个对象变得不可达，它就不会再变成可达的

垃圾检测的算法

- **引用计数法**
  - 给对象一个引用计数器，有引用+1，引用消失-1
  - 如果一个对象没有引用值，那么这个对象就是垃圾，需要回收
  - 存在问题：循环垃圾，即如果a、b两个对象相互引用，并没有其他的引用。那么他们的计数器即不为0，也属于需要回收的垃圾

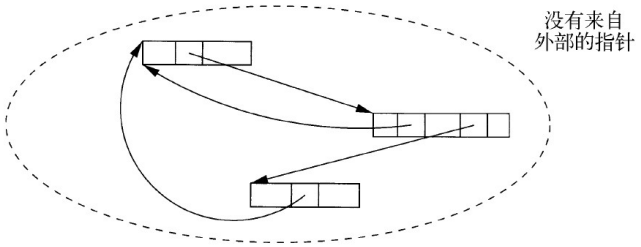


图 7-18　一个不可达的循环数据结构

垃圾检测的算法

- **可达性分析法**
  - 以根集对象为起始点进行搜索，如果有对象不可达的话，即是垃圾对象
  - 这里的根集一般包括java栈中引用的对象、方法区常良池中引用的对象本地方法中引用的对象等

垃圾回收的算法

- **标记-清除（Mark-sweep）**
  - 标记垃圾对象，统一回收
  - 优点：基础算法，简单实现
  - 缺点：效率低，产生大量碎片以根集对象为起始点进行搜索，如果有对象不可达的话，即是垃圾对象

垃圾回收的算法

- **复制（Copying）**
  - 内存分为平等2分，复制使用对象到另外一块
  - 优点:复制成本小、解决了碎片的问题 缺点：内存需求更多
- **标记-整理（Mark-Compact）**
  - 标记存活对象，清除为标记对象，并压缩存活对象
- **分代收集算法**
  - 分代的垃圾回收策略

Thank you!