运行时刻环境
Part2：空间的栈式分配

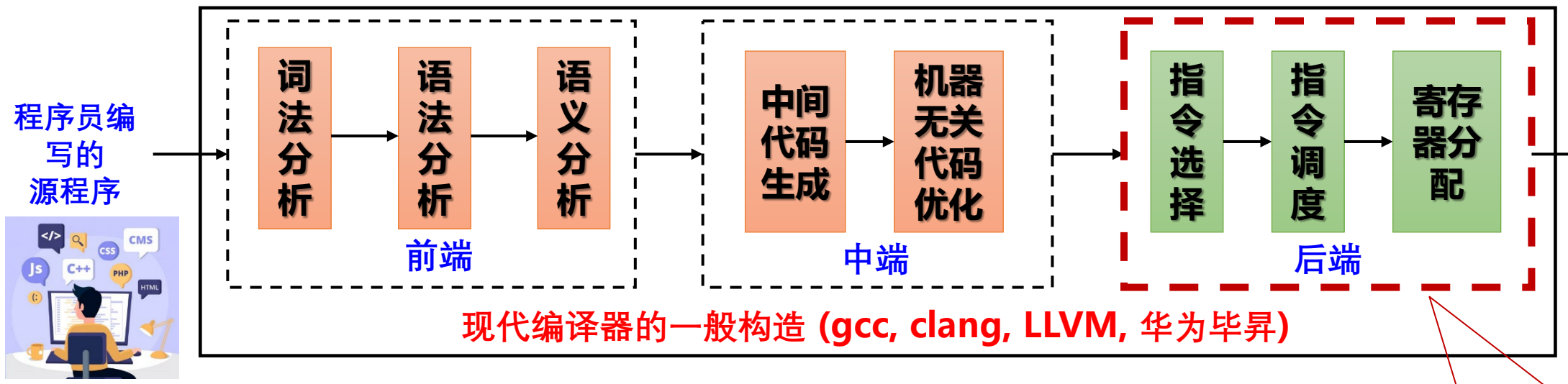
李 诚

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2024年11月04日

本节提纲



- 活动树与运行栈
- 调用序列与返回序列

目标程序需要一个运行环境！

栈式存储分配

- 对于支持过程、函数和方法的语言，其编译器通常会用栈的形式来管理其运行时刻存储
- 当一个过程被调用时，该过程的活动记录被压入栈中；当过程结束时，记录被弹出
- 这种安排不仅允许活跃时段不交叠的多个过程调用共享空间；而且可以使得过程的非局部变量的相对地址总是固定的，和调用序列无关

活动树

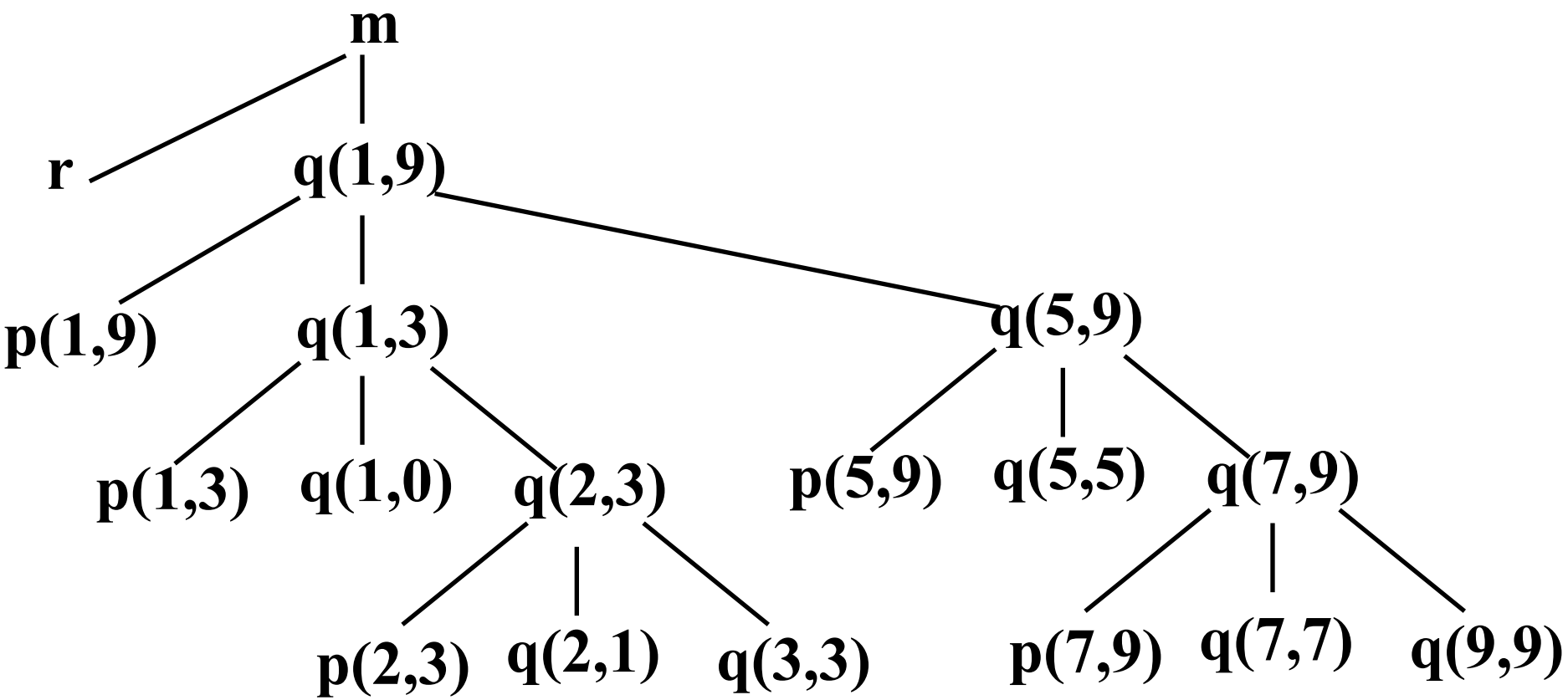
- 用来描述程序运行期间控制进入和离开各个活动的情况的树
- 树中的每个结点对应于一个活动。根结点是启动程序执行的main过程的活动
- 对于过程p，其子结点对应于被p的这次活动调用的各个过程的活动。按照调用次序，自左向右地显示它们。一个子结点必须在其右兄弟结点的活动开始之前结束。

例：一个快排程序的介绍

```
int a[11];
void readArray() /*将9个数读入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, a[p+1…n]大于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);}
```

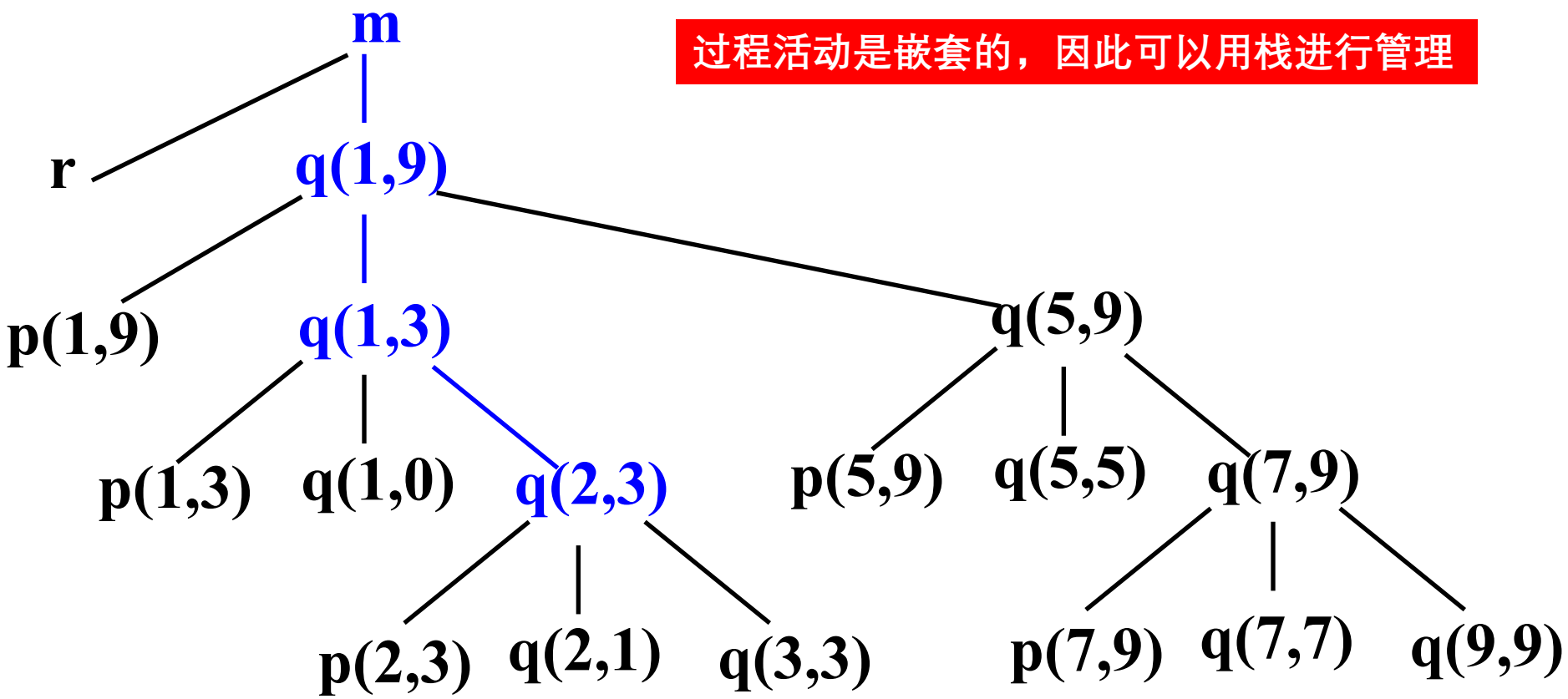
活动树和运行栈

- 活动树
  - 用树来描绘控制进入和离开活动的方式



活动树和运行栈

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



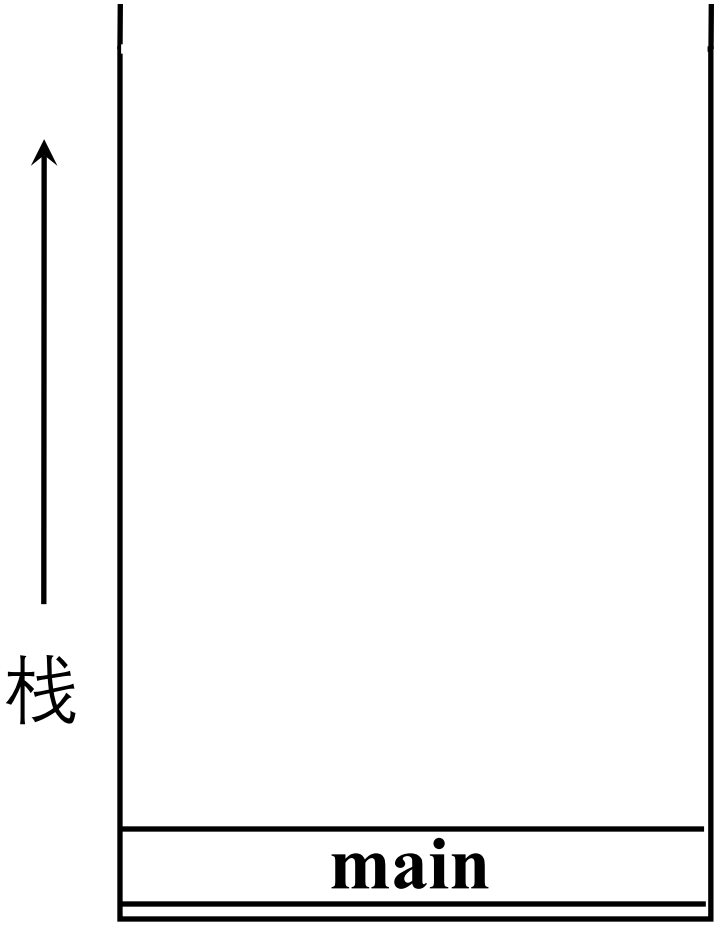
过程活动是嵌套的，因此可以用栈进行管理

活动树和运行栈

- 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）

活动树和运行栈

- 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）

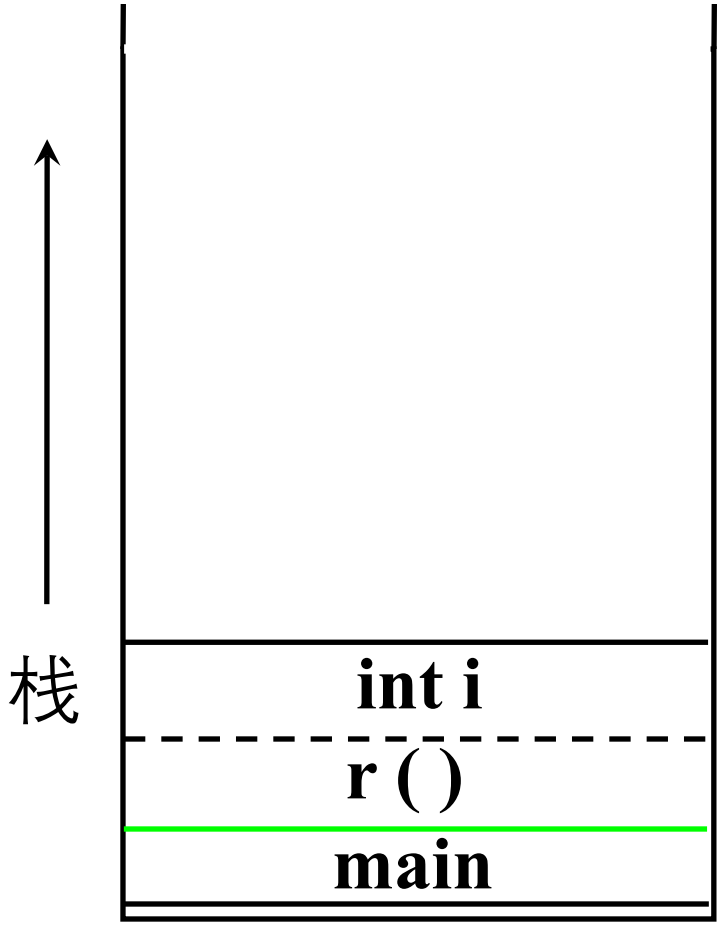


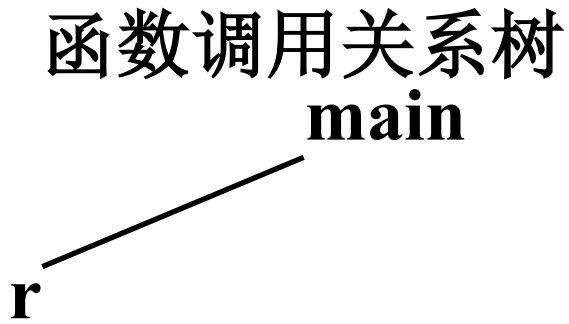
函数调用关系树

main

活动树和运行栈

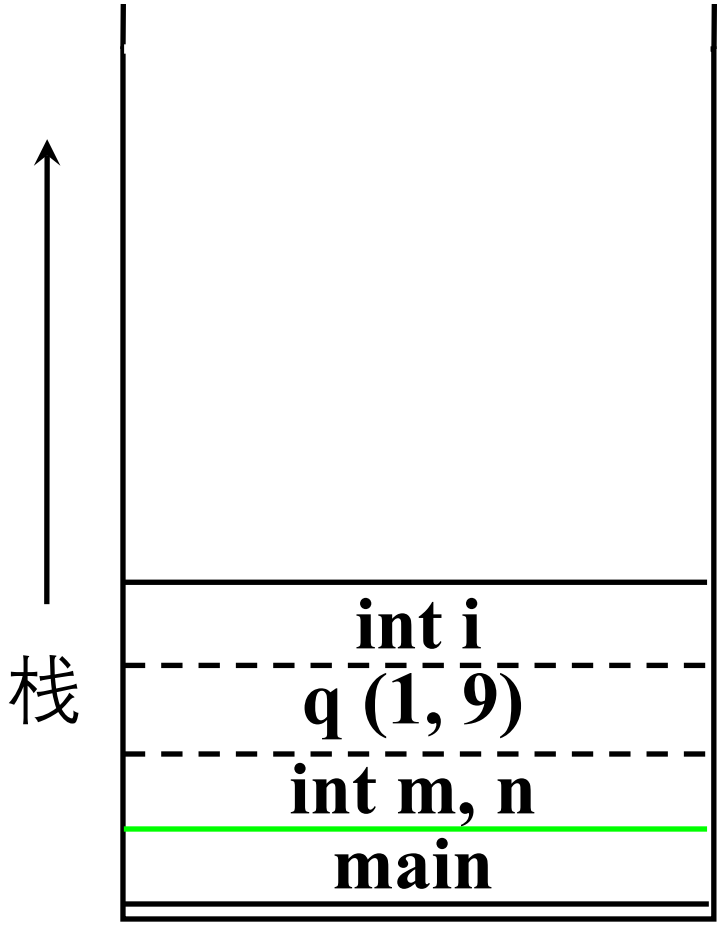
- 运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



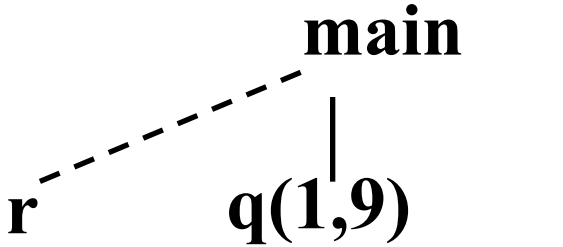


活动树和运行栈

- 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）

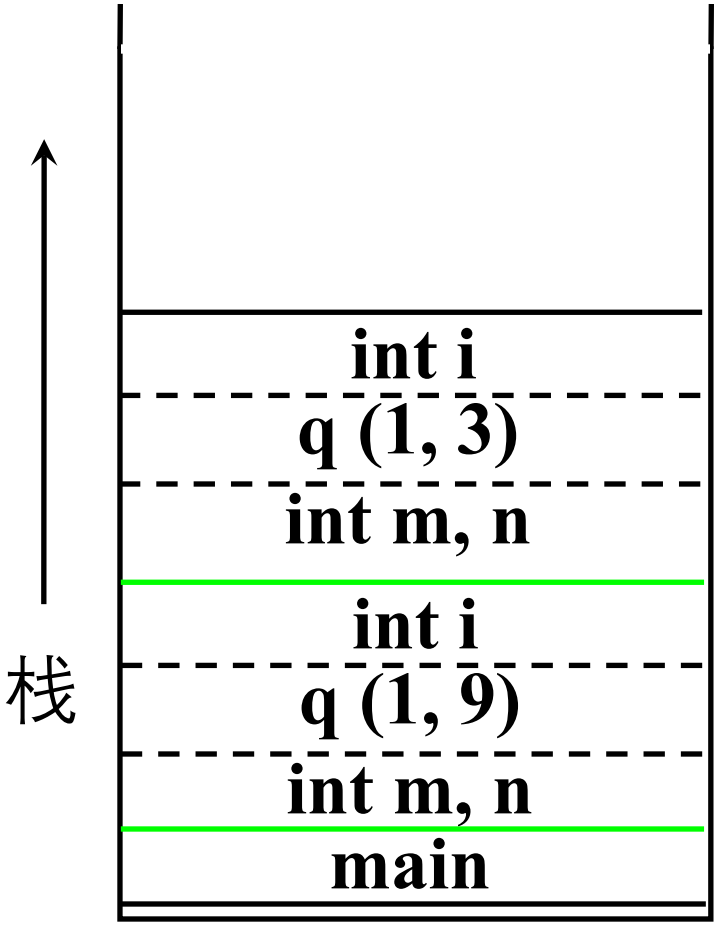


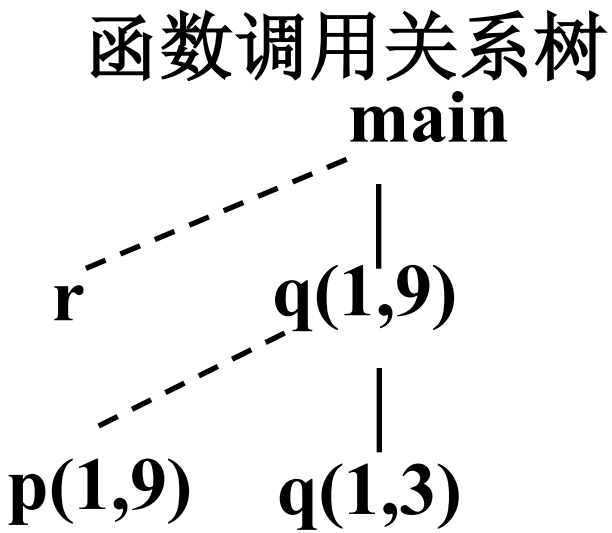
函数调用关系树



活动树和运行栈

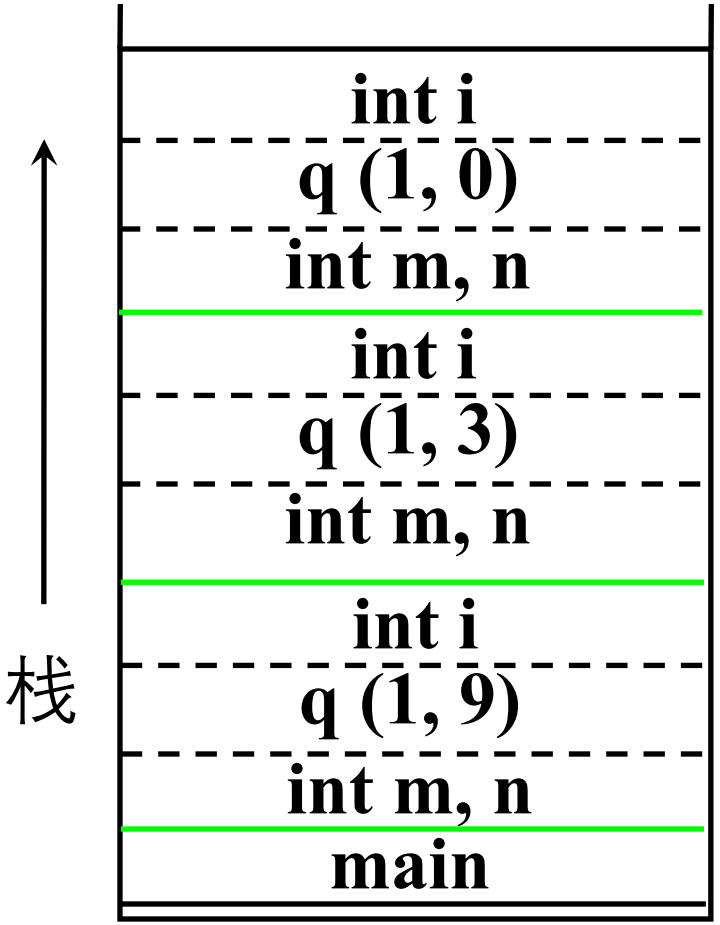
- 运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



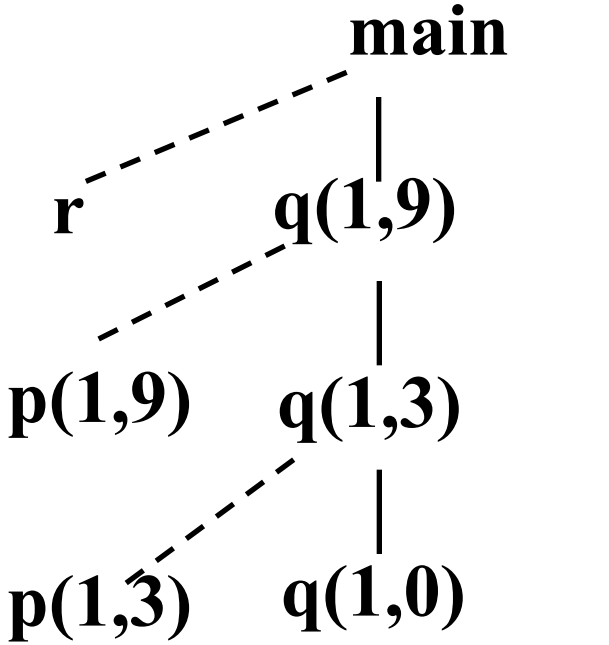


活动树和运行栈

- 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）

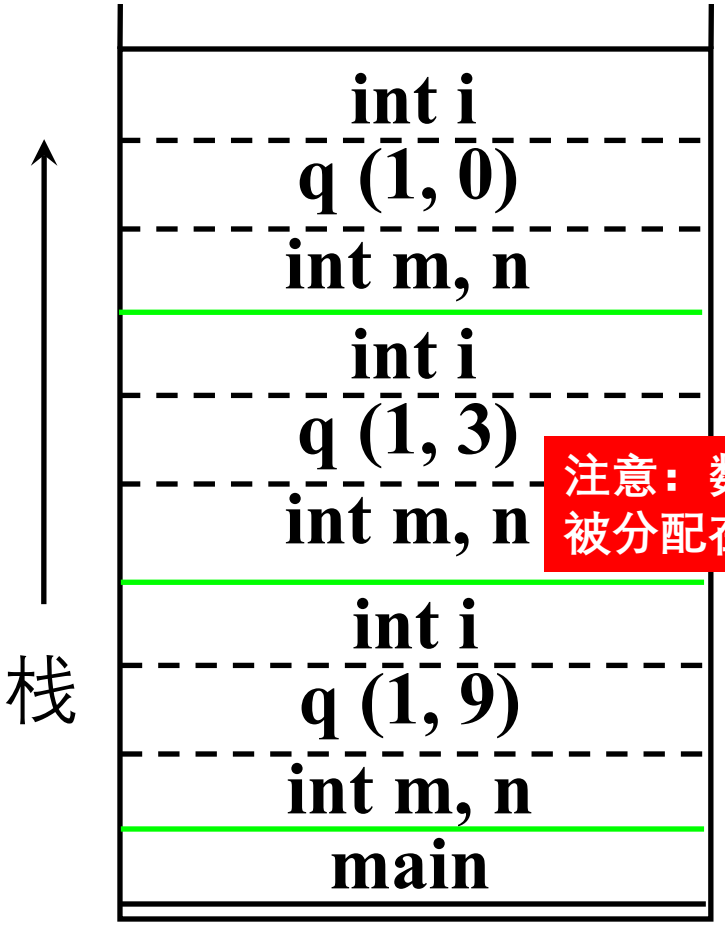


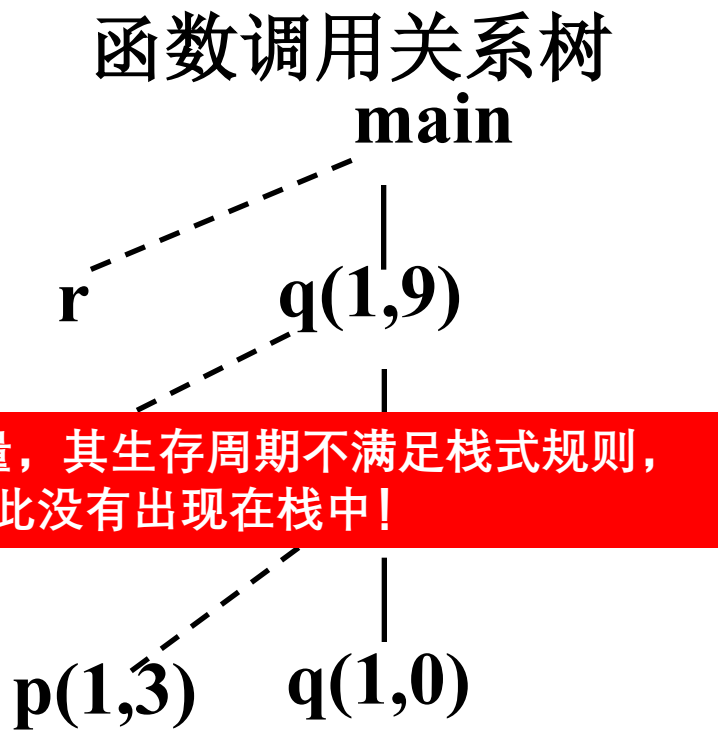
函数调用关系树



活动树和运行栈

- 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）



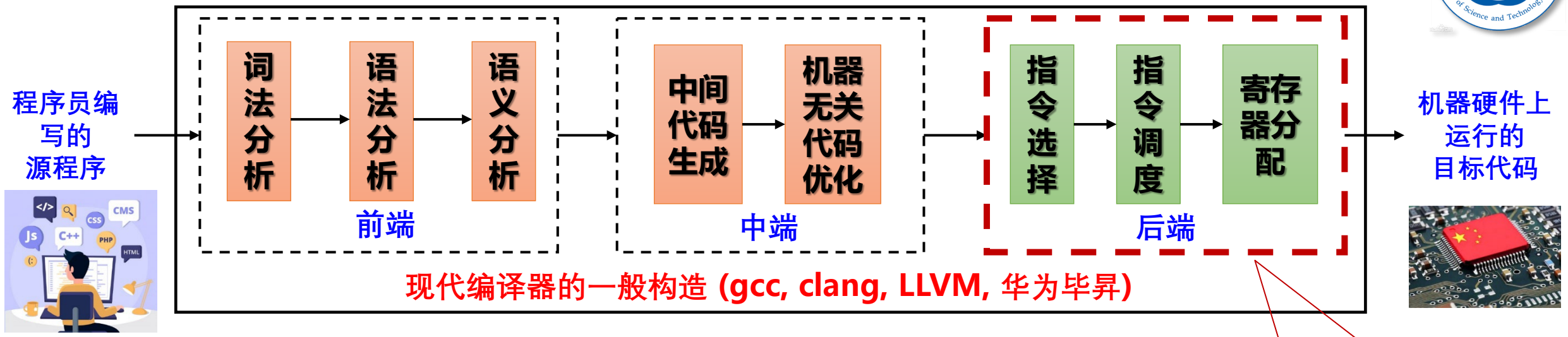


注意：数组a是全局变量，其生存周期不满足栈式规则，被分配在静态区域，因此没有出现在栈中！

活动树与控制栈

- 每个活跃的活动**都有**一个位于控制栈中的活动记录
- 活动树的根结点的记录位于**栈底**
- 程序控制所在的活动的记录位于**栈顶**
- 栈中全部活动记录的序列对应在活动树中到达当前控制所在的活动结点的**路径**

本节提纲



- 活动树与运行栈
- **调用序列与返回序列**

目标程序需要一个运行环境！

对运行栈的管理

- **代码序列**
  - 过程调用和过程返回都需要执行一些代码来管理活动记录栈，保存或恢复机器状态等

对运行栈的管理

- **代码序列**
  - 过程调用和过程返回都需要执行一些代码来管理活动记录栈，保存或恢复机器状态等
- **过程调用序列(calling sequence)**
  - 过程调用时执行的分配活动记录，把信息填入它的域中，使被调用过程可以开始执行的代码
- **过程返回序列(return sequence)**
  - 被调用过程返回时执行的恢复机器状态，释放被调用过程活动记录，使调用过程能够继续执行的代码

对运行栈的管理

- **代码序列**
  - 过程调用和过程返回都需要执行一些代码来管理活动记录栈，保存或恢复机器状态等
- **过程调用序列(calling sequence)**
  - 过程调用时执行的分配活动记录，把信息填入它的域中，使被调用过程可以开始执行的代码
- **过程返回序列(return sequence)**
  - 被调用过程返回时执行的恢复机器状态，释放被调用过程活动记录，使调用过程能够继续执行的代码
- **调用序列和返回序列常常都分成两部分，分处于调用过程和被调用过程的活动记录中**

对运行栈的管理

- 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

  - 调用者与被调用者之间交流的数据放在被调用者活动记录的开始处，尽量靠近调用者的活动记录
    - 参数域紧邻调用者活动记录
    - 返回值在参数域之上

对运行栈的管理

- 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
  - 固定长度的域放在活动记录的中间
    - 控制链
    - 访问链
    - 机器状态

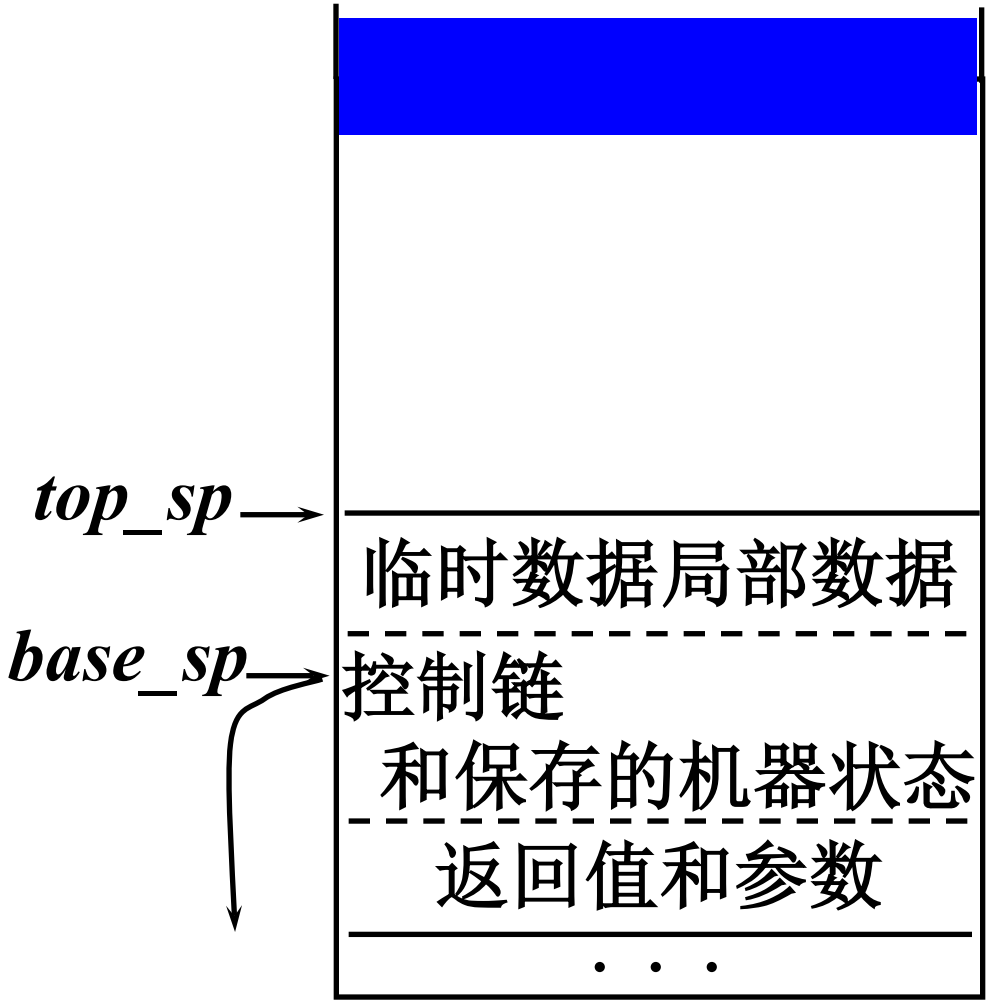
对运行栈的管理

- 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
  - 不能编译时刻确定大小的数据一般放在活动记录的末端
    - 局部动态数组
    - 临时数据

对运行栈的管理

- 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(控制链)来活动记录中的内容

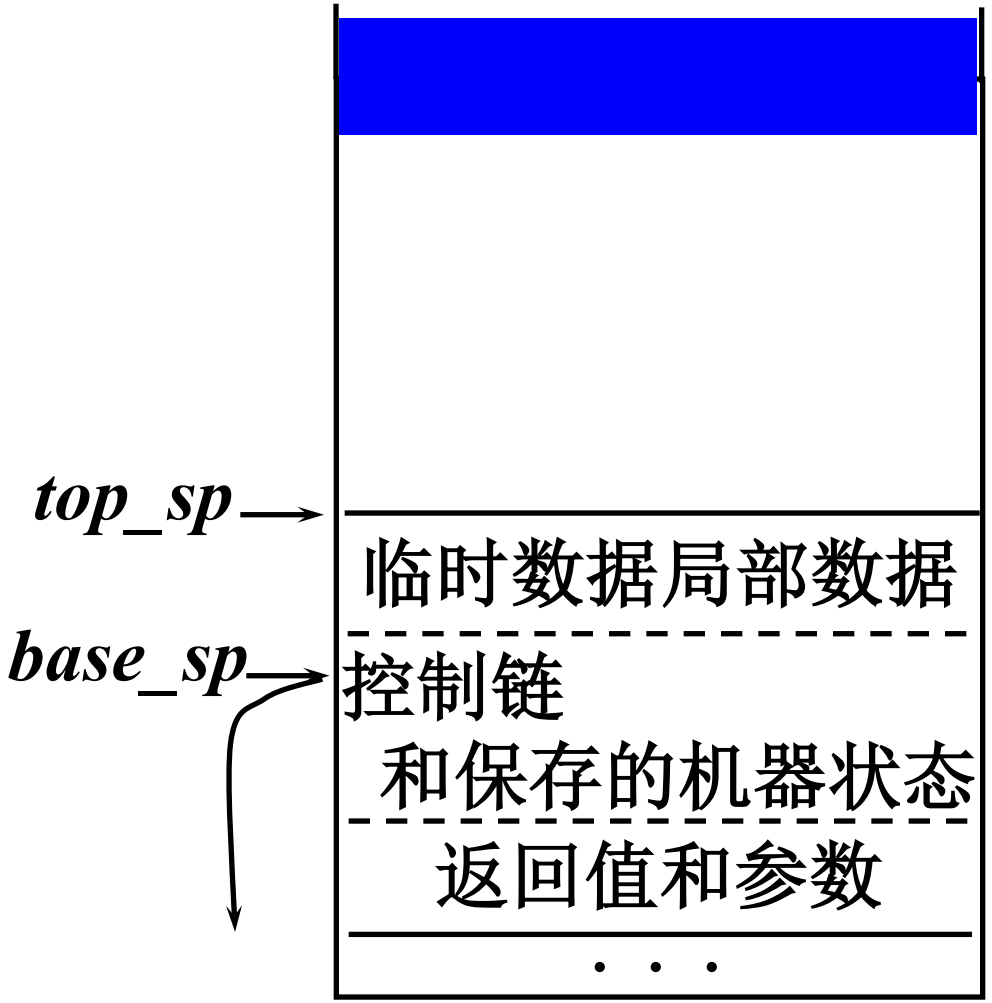
全局栈式存储分配



- **top_sp**: 栈顶寄存器，如esp、rsp。指向栈顶活动记录的末端
- **base_sp**: 基址寄存器，如ebp、rbp。指向栈顶活动记录中控制链所在位置。

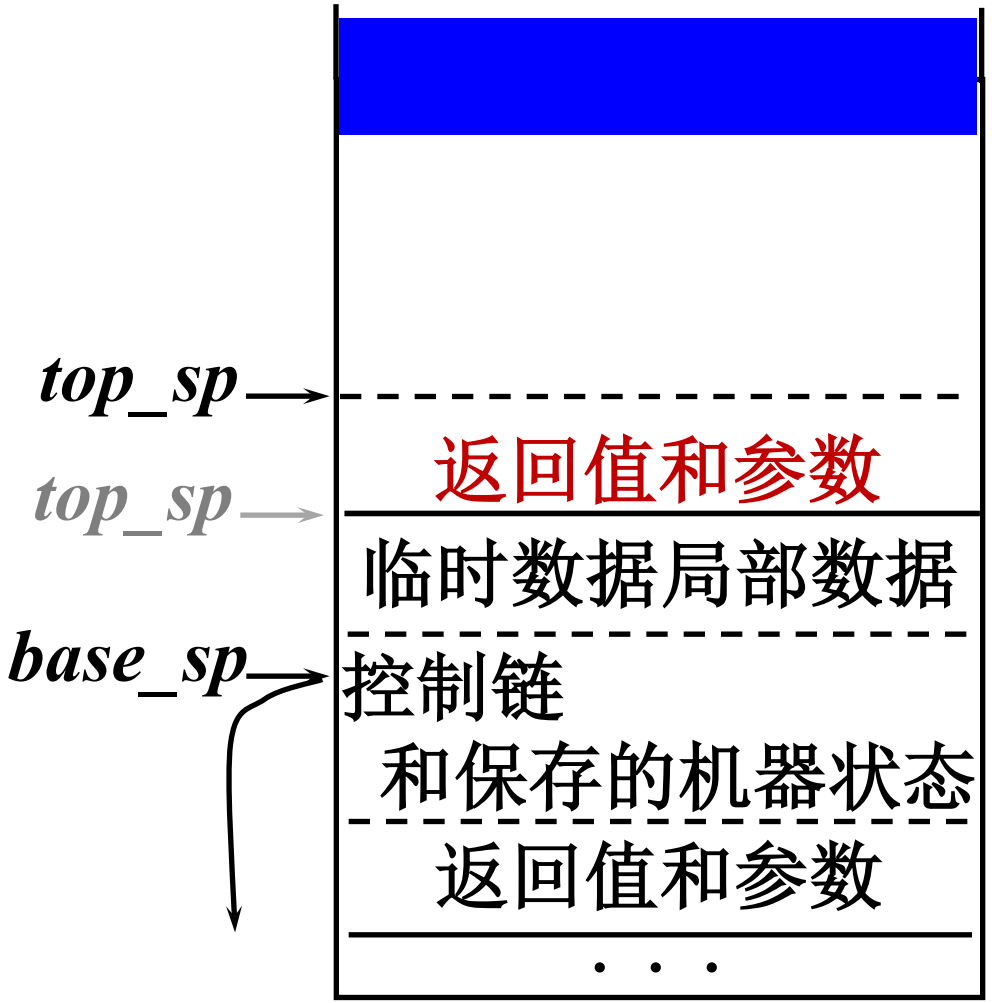
全局栈式存储分配

- 过程p调用过程q的调用序列(栈往上增长)



全局栈式存储分配

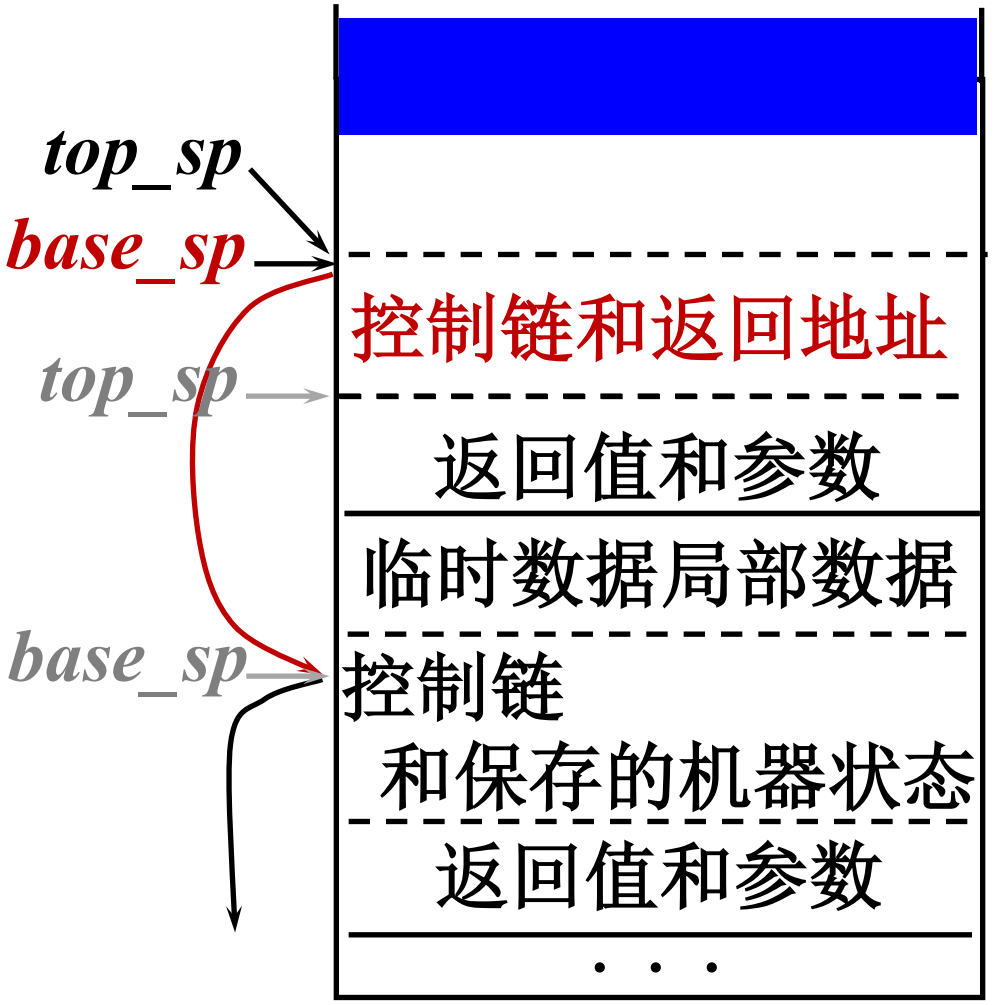
- 过程p调用过程q的调用序列(栈往上增长)



(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

全局栈式存储分配

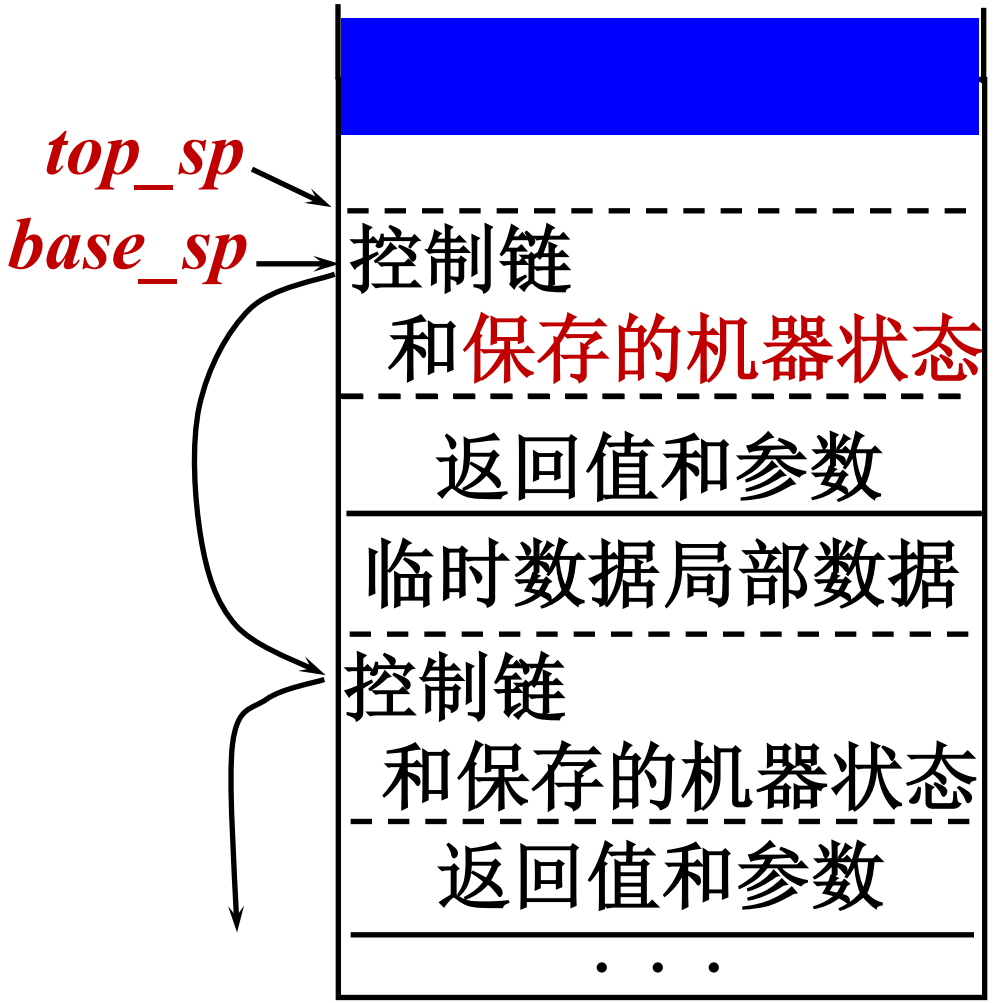
- 过程p调用过程q的调用序列



(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的控制链，改变base_sp的值

全局栈式存储分配

- 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

全局栈式存储分配

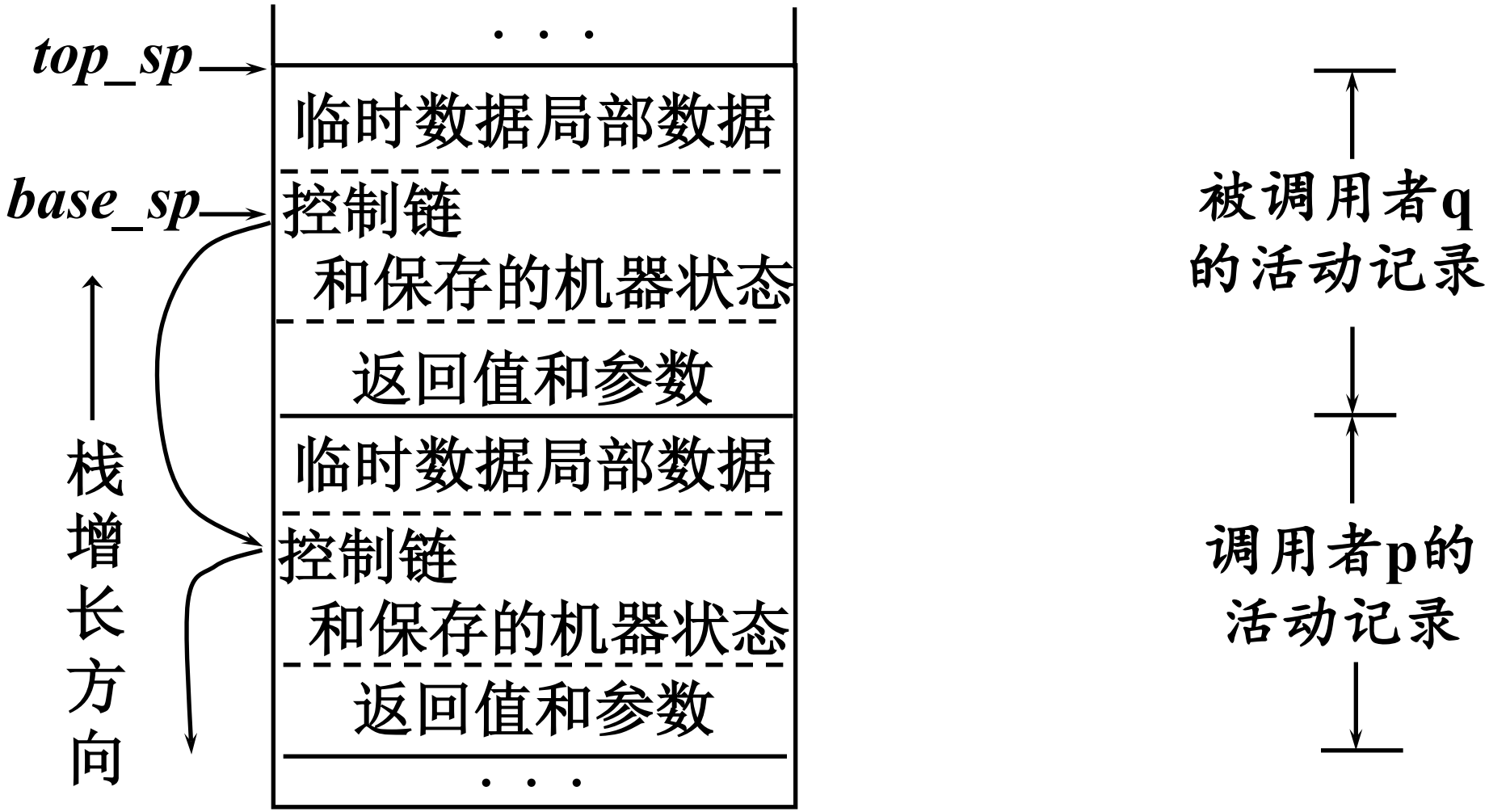
- 过程p调用过程q的调用序列

base_sp不变，指向活动记录中间

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

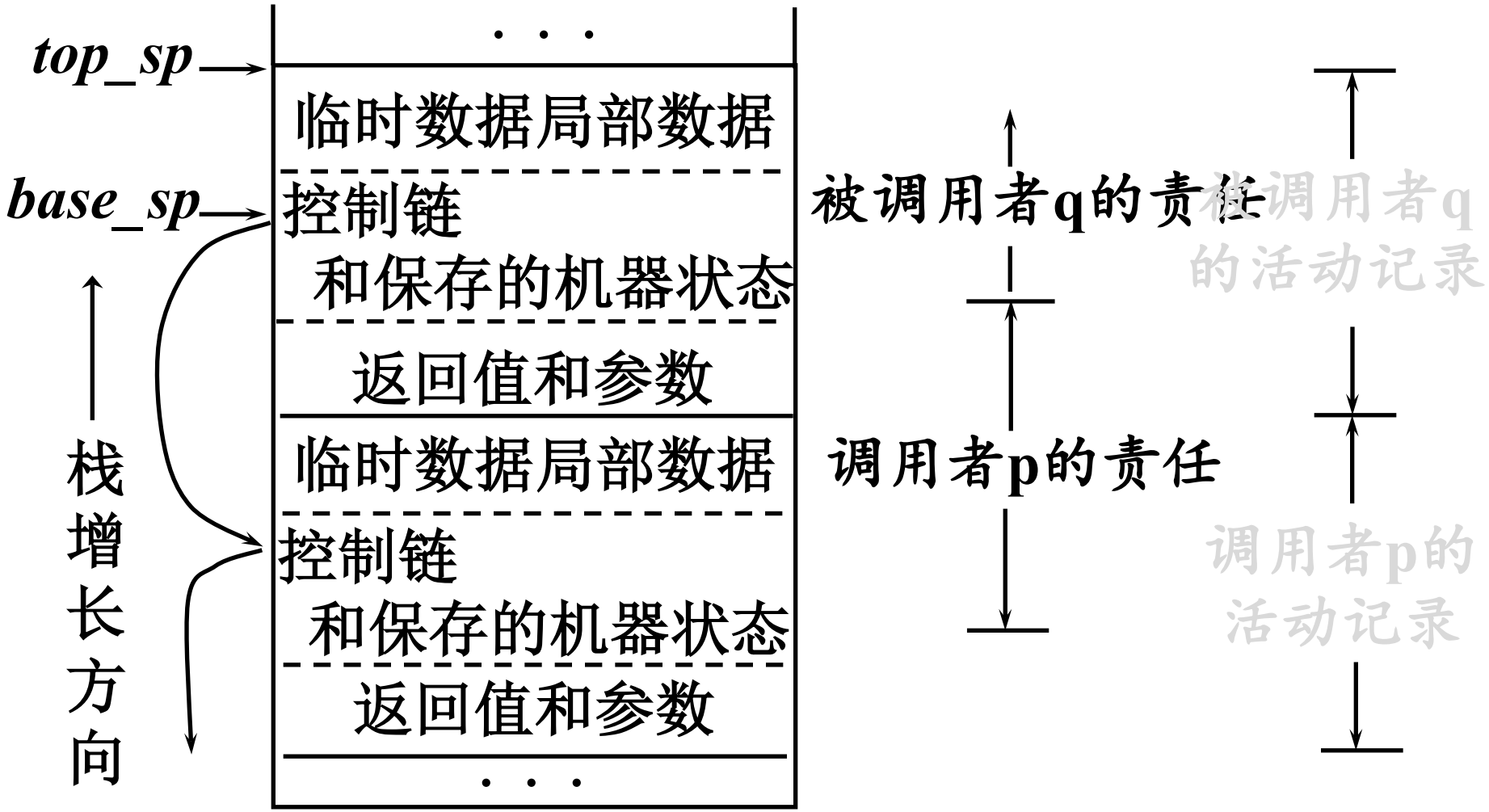
全局栈式存储分配

- 调用者p和被调用者q之间的任务划分



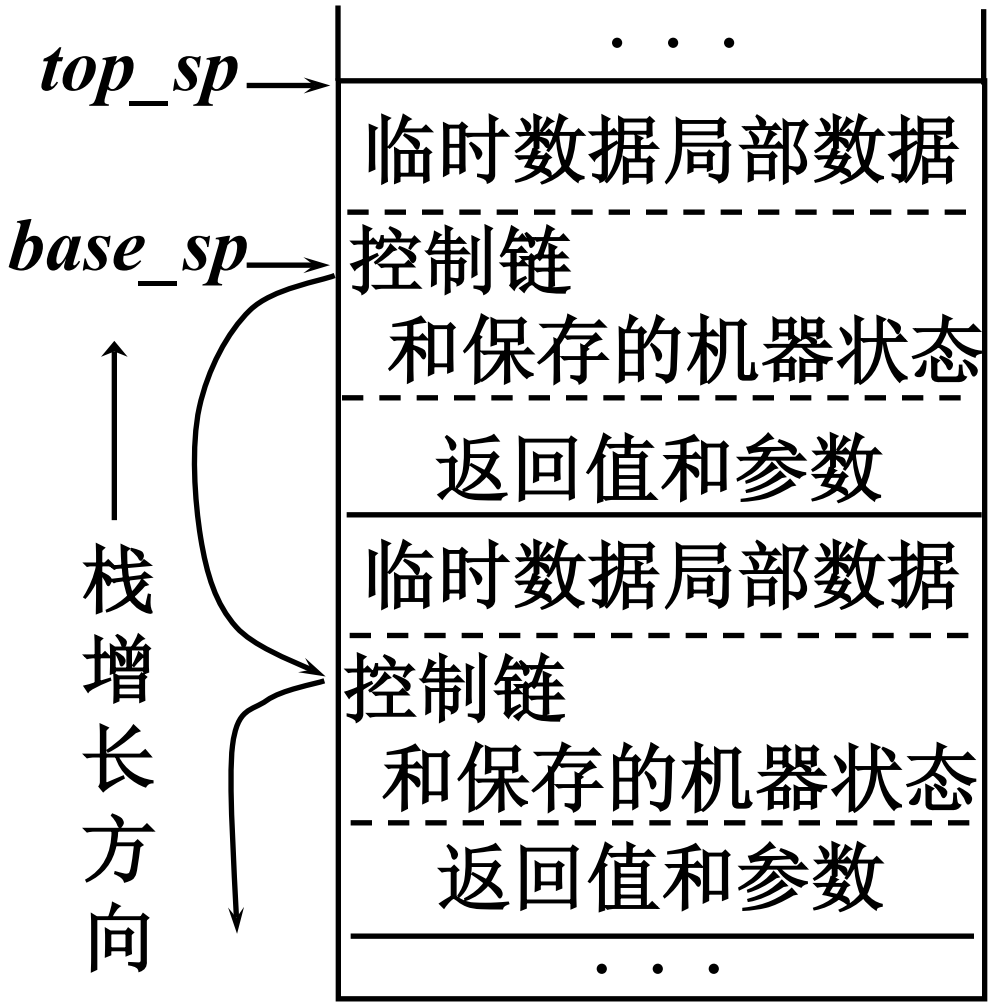
全局栈式存储分配

- 调用者p和被调用者q之间的任务划分



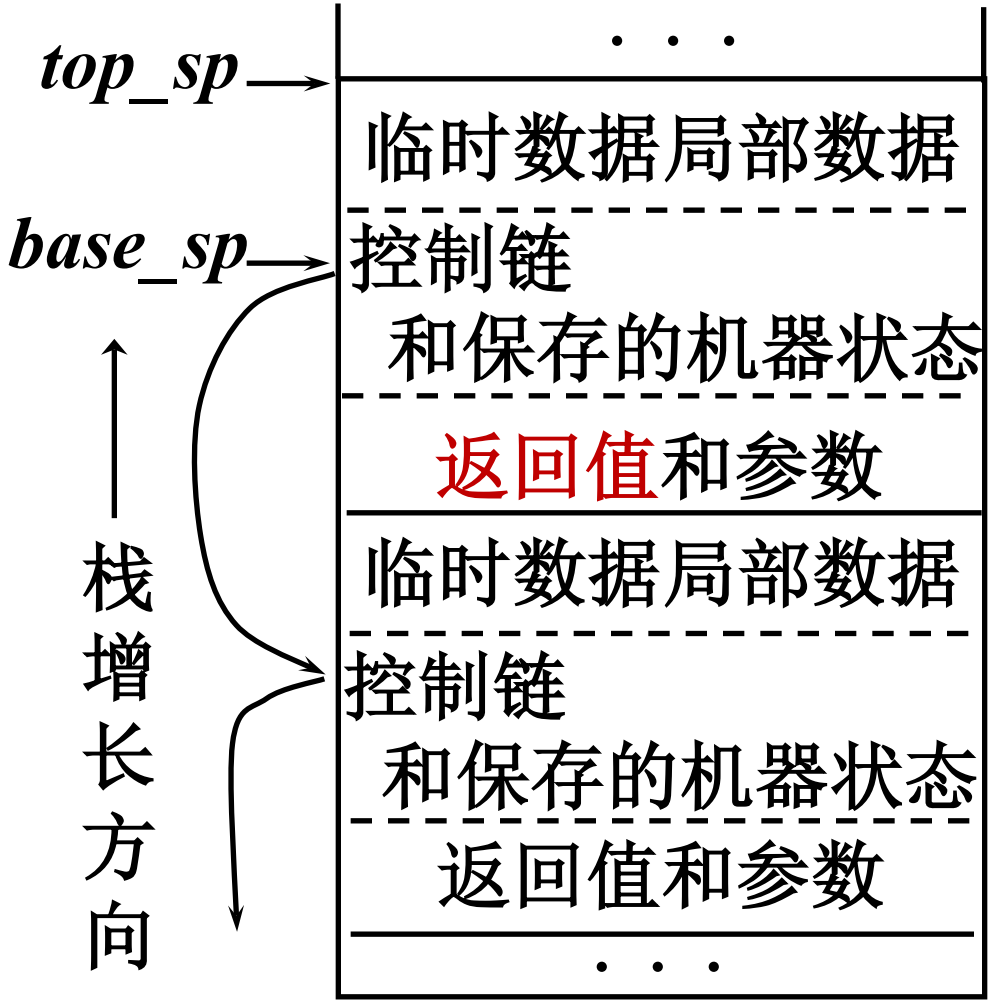
全局栈式存储分配

- 过程p调用过程q的返回序列



全局栈式存储分配

- 过程p调用过程q的返回序列

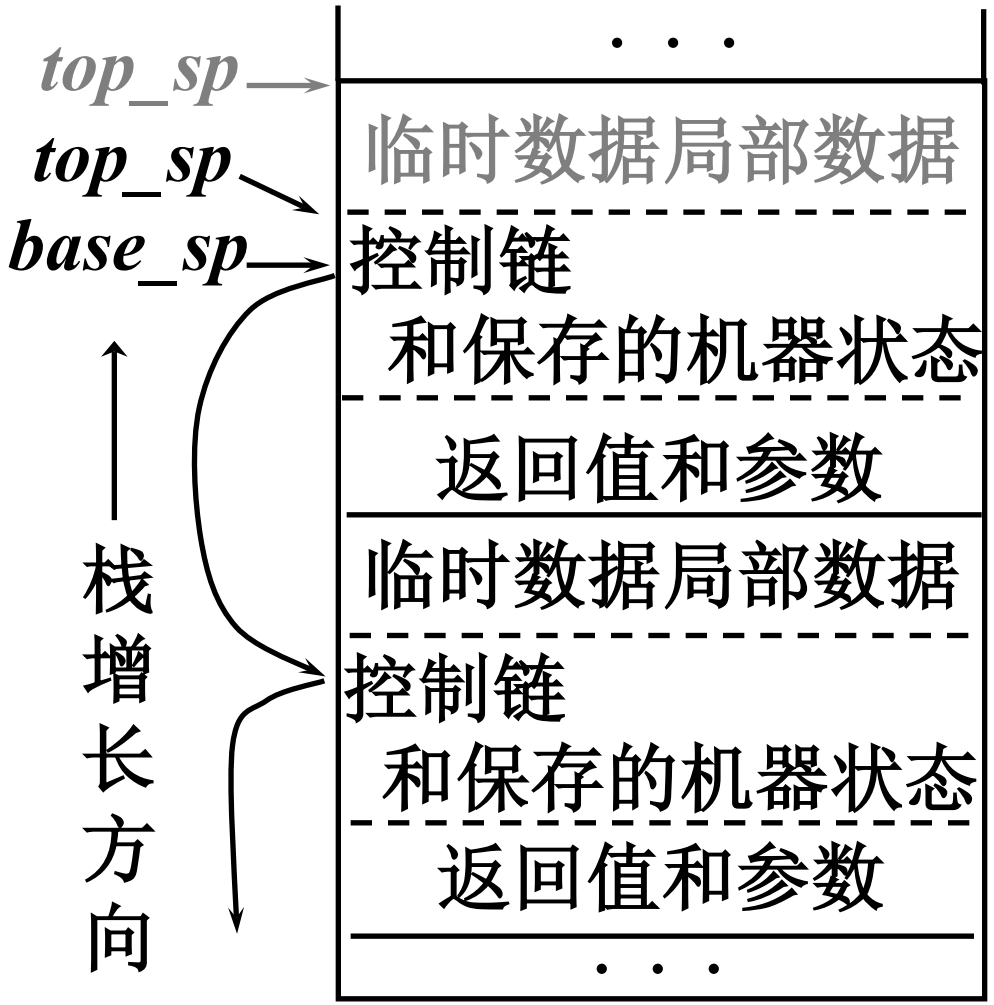


(1) q把返回值置入邻近p的活动记录的地方

引申：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

全局栈式存储分配

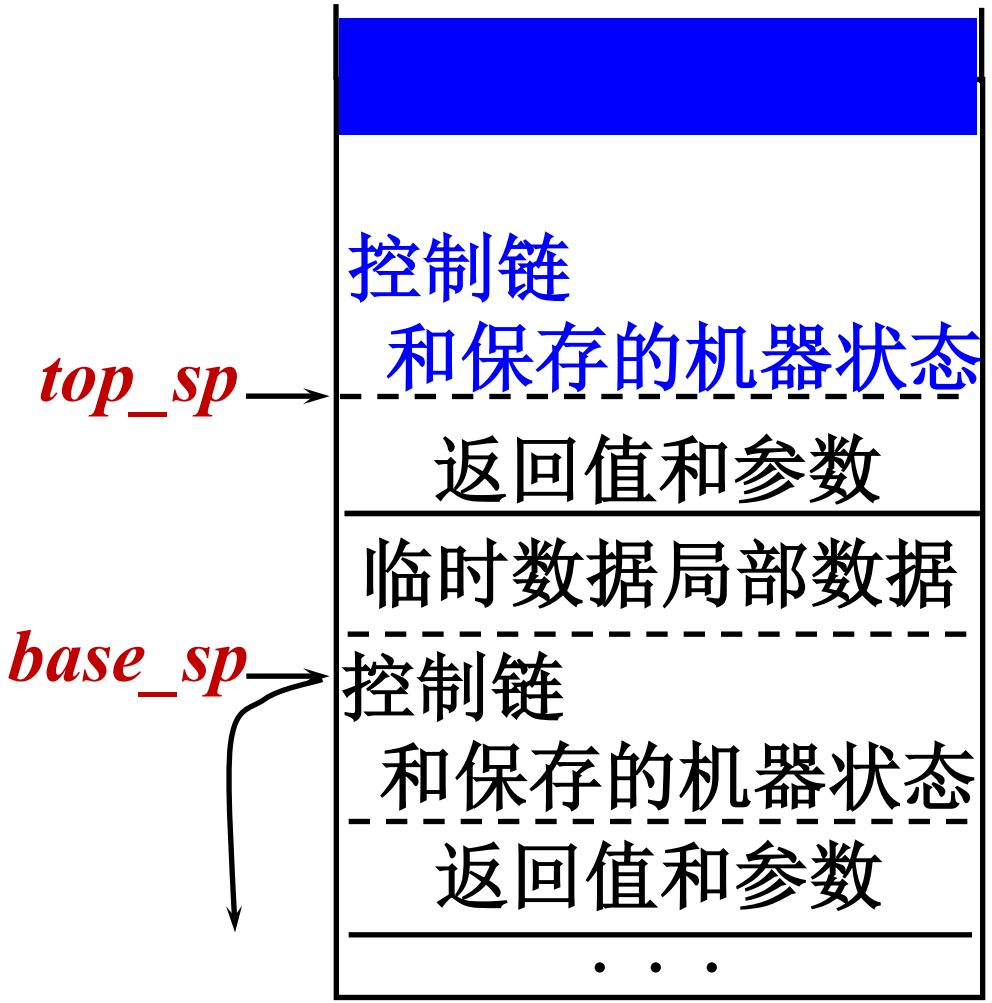
- 过程p调用过程q的返回序列



(2) q对应调用序列的步骤(4)，增加top_sp的值

全局栈式存储分配

- 过程p调用过程q的返回序列

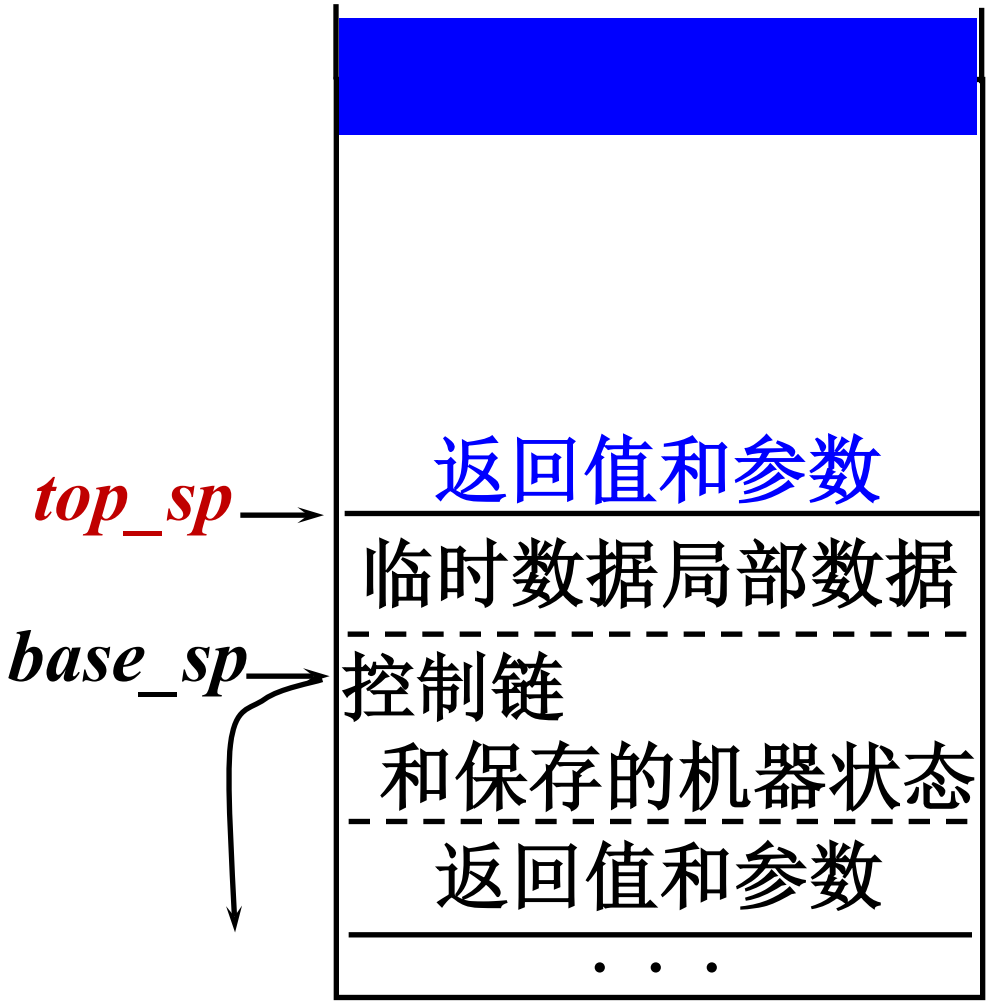


(3) q恢复寄存器(包括base_sp)和机器状态，返回p

控制权转到p

全局栈式存储分配

- 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

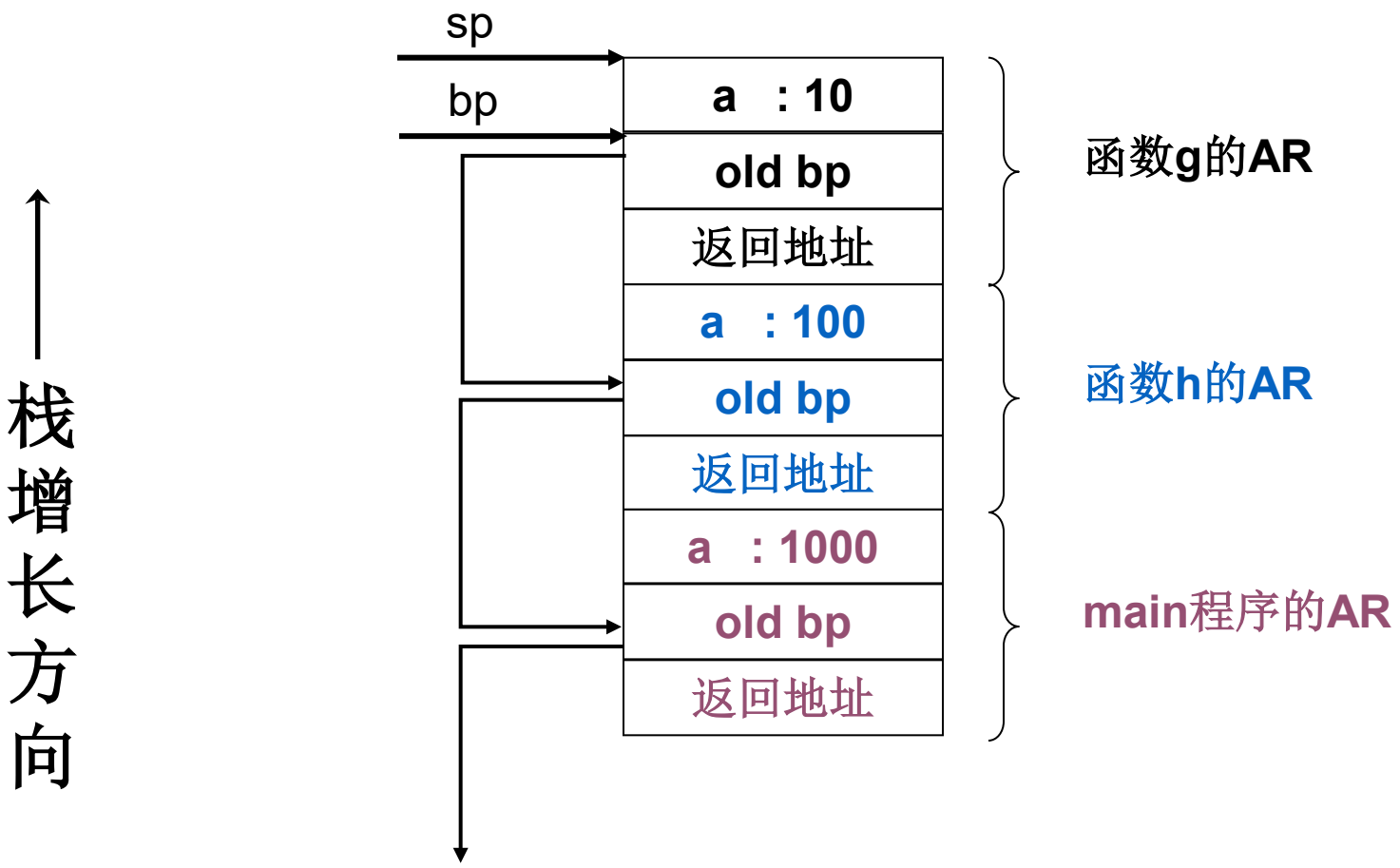
全局栈式存储分配

有C程序如下：

```
void g() { int a ; a = 10 ; }

void h() { int a ; a = 100; g();  }

main()

{  int a = 1000; h(); }
```

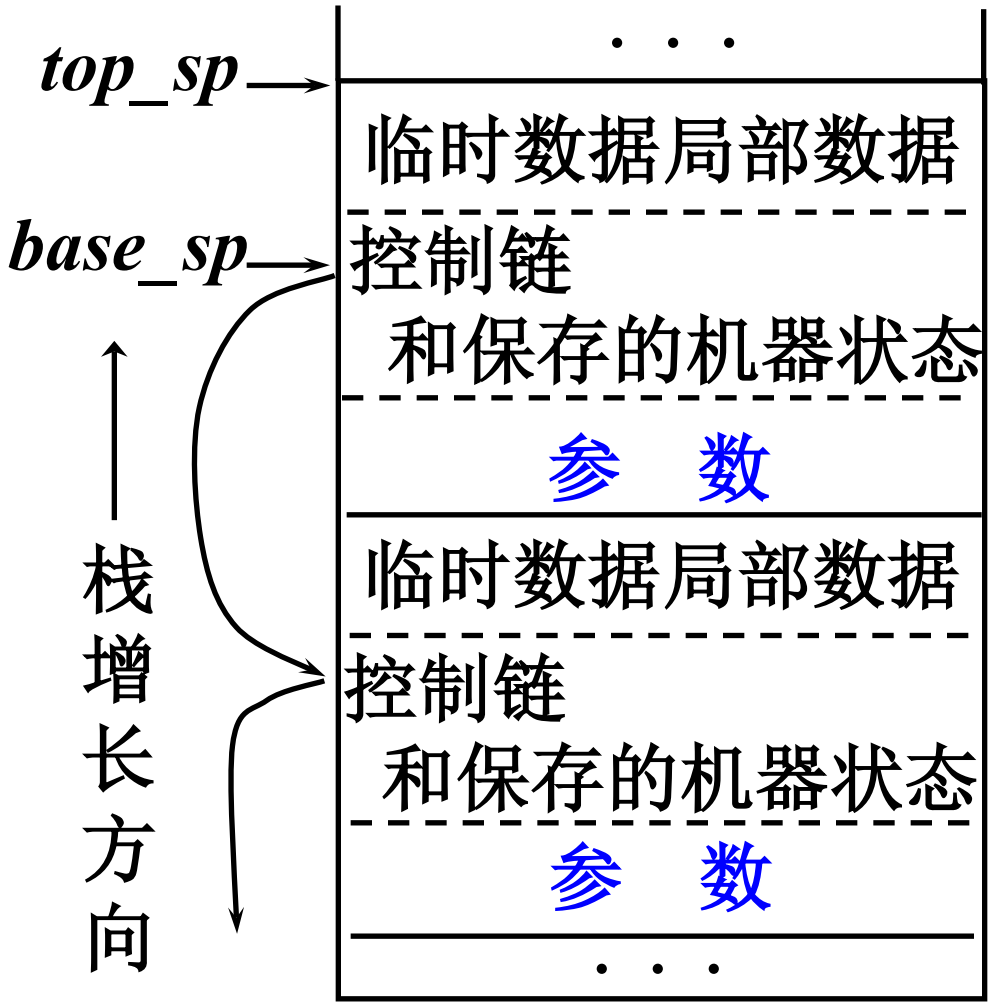
全局栈式存储分配

- 过程g被调用时，活动记录栈的（大致）内容



全局栈式存储分配

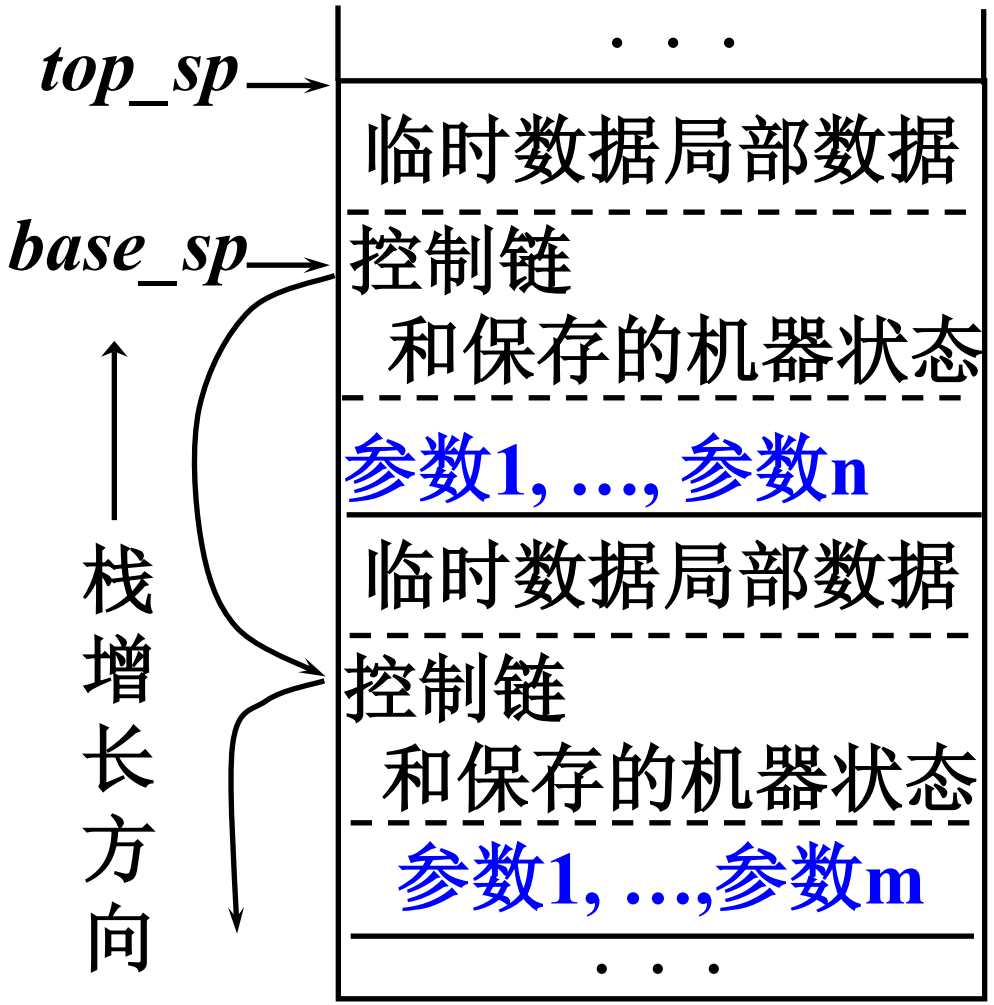
- 过程的参数个数可变的情况



(1) 函数返回值改成用寄存器传递

全局栈式存储分配

- 过程的参数个数可变的情况

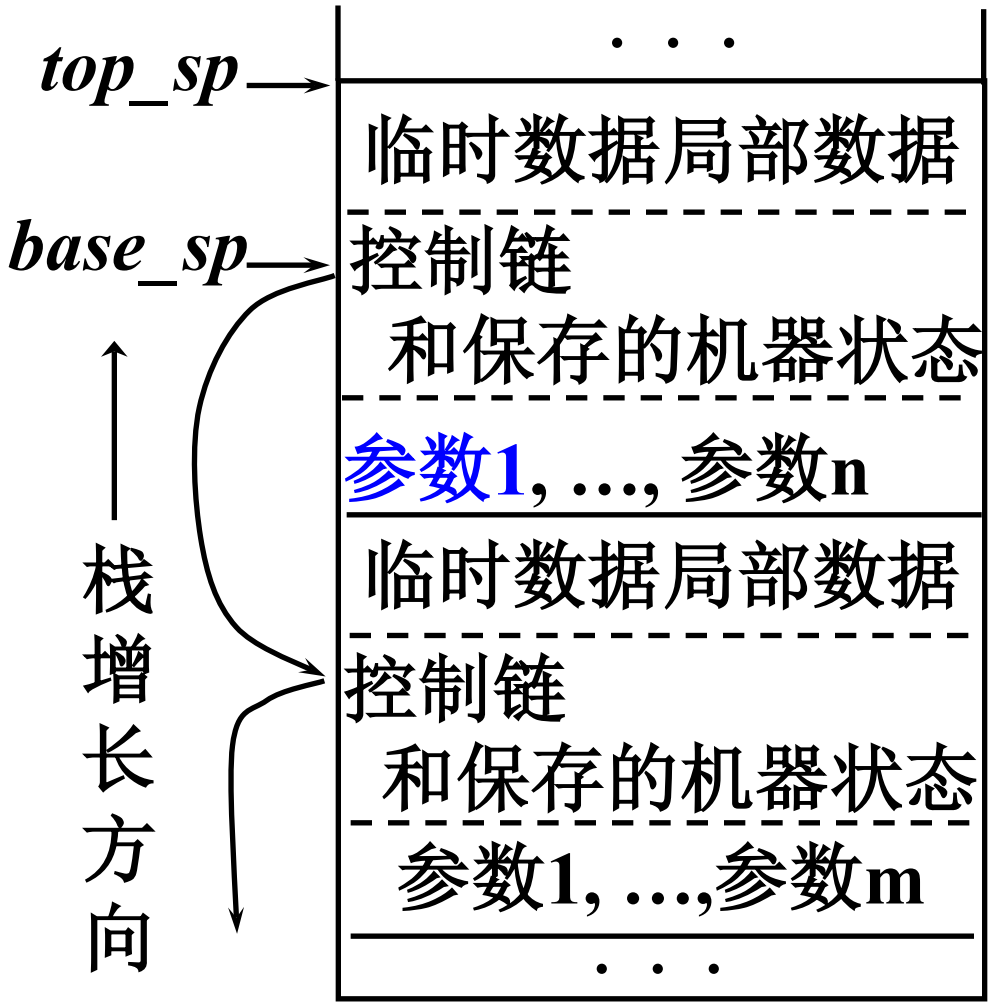


(2) 编译器产生将实参表达式逆序计算并将结果进栈的代码

自上而下依次是参数1, …, 参数n

全局栈式存储分配

- 过程的参数个数可变的情况

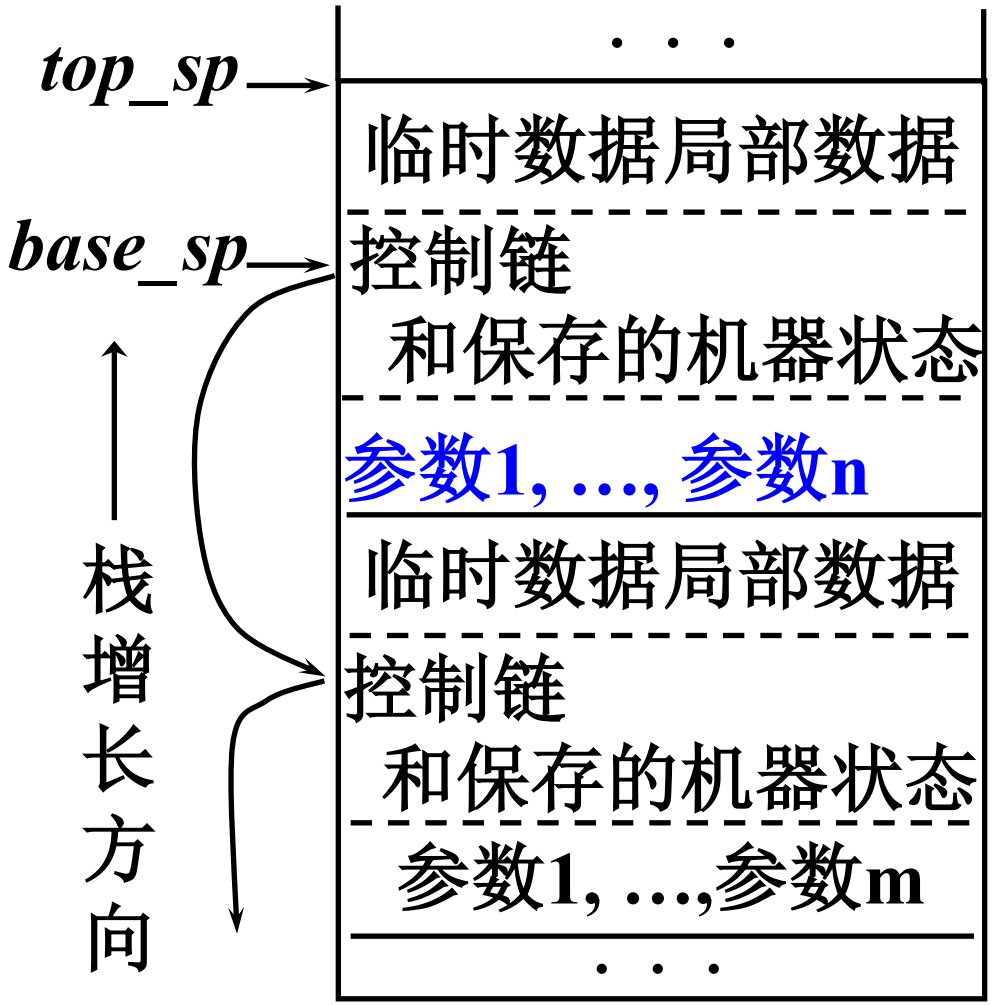


(3) 被调用函数能准确地知道第一个参数的位置

But why?

全局栈式存储分配

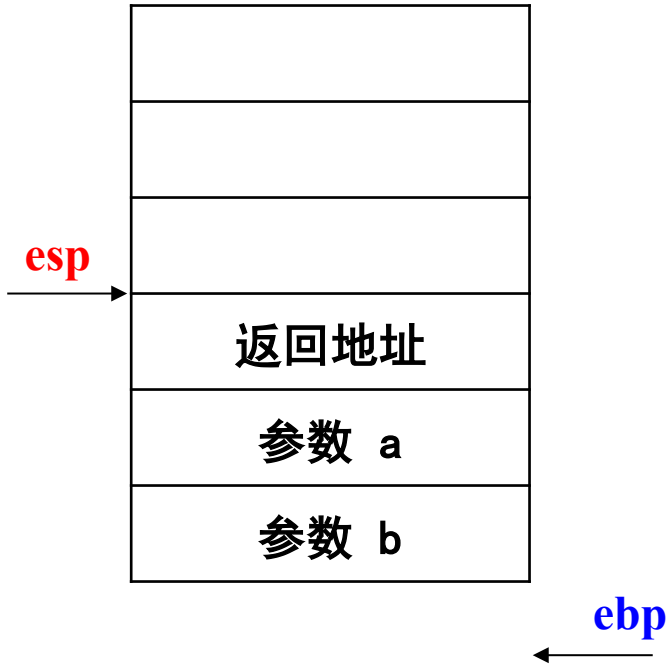
- 过程的参数个数可变的情况



(4) 被调用函数根据第一个参数到栈中取第二、第三个参数等等

有参函数的活动记录

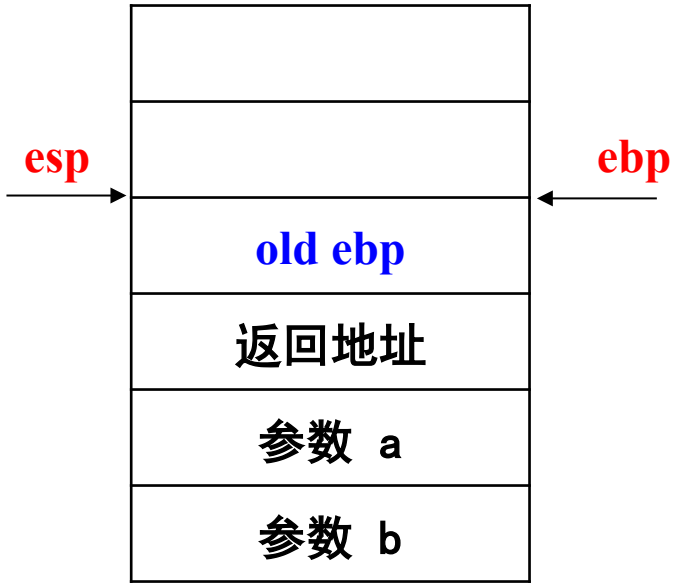
```
void func( int a , int b )
{
    int c , d;
    c = a;
    d = b;
}
```




```
.file   "ar.c"
        .text
.globl func
        .type   func,@function
func:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -8(%ebp)
        leave
        ret
```

有参函数的活动记录

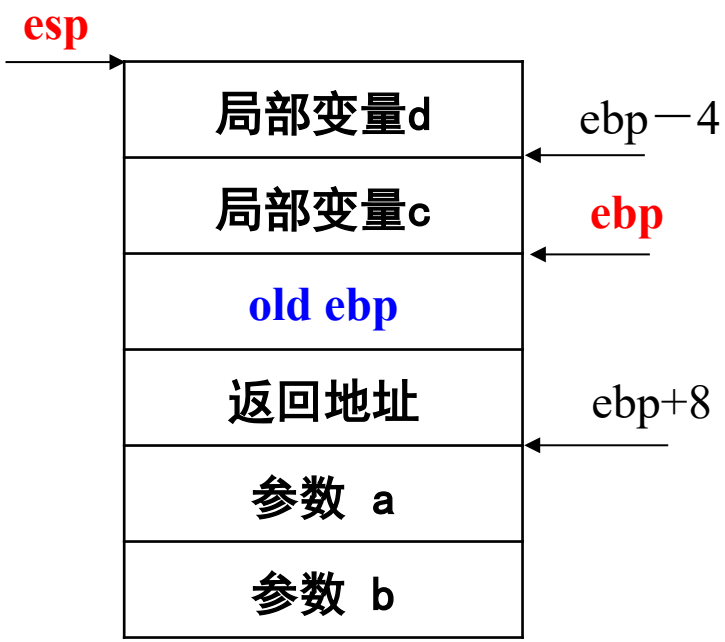
```
void func( int a , int b )
{
    int c , d;
    c = a;
    d = b;
}
```




```
.file   "ar.c"
        .text
.globl func
        .type   func,@function
func:
        pushl   %ebp //老基地址压栈
        movl    %esp, %ebp //基地址指针=栈顶指针
        subl    $8, %esp
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -8(%ebp)
        leave
        ret
```

有参函数的活动记录

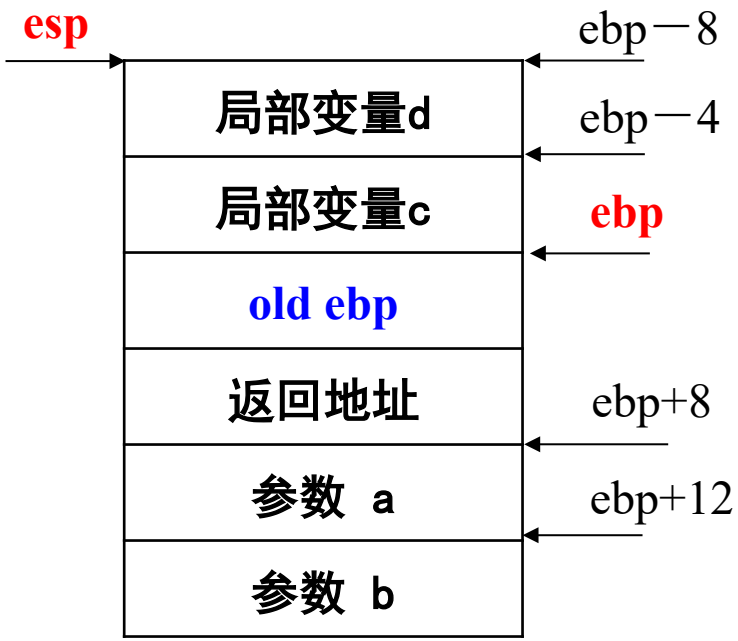
```
void func( int a , int b )
{
    int c , d;
    c = a;
    d = b;
}
```




```
.file   "ar.c"
        .text
.globl func
        .type   func,@function
func:
        pushl   %ebp //老基地址压栈
        movl    %esp, %ebp //基地址指针=栈顶指针
        subl    $8, %esp //分配c,d局部变量空间
        movl    8(%ebp), %eax //将a值放进寄存器
        movl    %eax, -4(%ebp) //将a值赋给c
        movl    12(%ebp), %eax
        movl    %eax, -8(%ebp)
        leave
        ret
```

有参函数的活动记录

```
void func( int a , int b )
{
    int c , d;
    c = a;
    d = b;
}
```



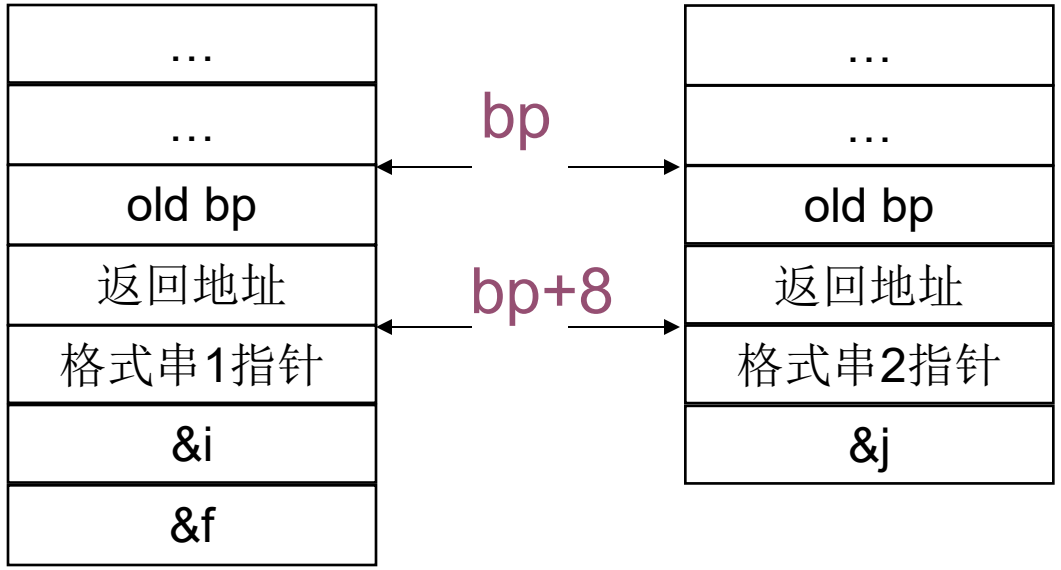

```
.file   "ar.c"
        .text
.globl func
        .type   func,@function
func:
        pushl   %ebp //老基地址压栈
        movl    %esp, %ebp //基地址指针=栈顶指针
        subl    $8, %esp //分配c,d局部变量空间
        movl    8(%ebp), %eax //将a值放进寄存器
        movl    %eax, -4(%ebp) //将a值赋给c
        movl    12(%ebp), %eax //将b值放进寄存器
        movl    %eax, -8(%ebp) //将b值赋给d
        leave
        ret
```

scanf的可变参数

有如下C程序：

```
main()
{
 int i ; float f;  int j ;
 scanf(“%d%f”, &i, &f); //第一次调用时3个参数
 scanf(“%d”, &j ); //第二次调用时 2个参数
 return 0;
}
```

scanf的可变参数



scanf的第一次调用时AR　　scanf的第二次调用时AR

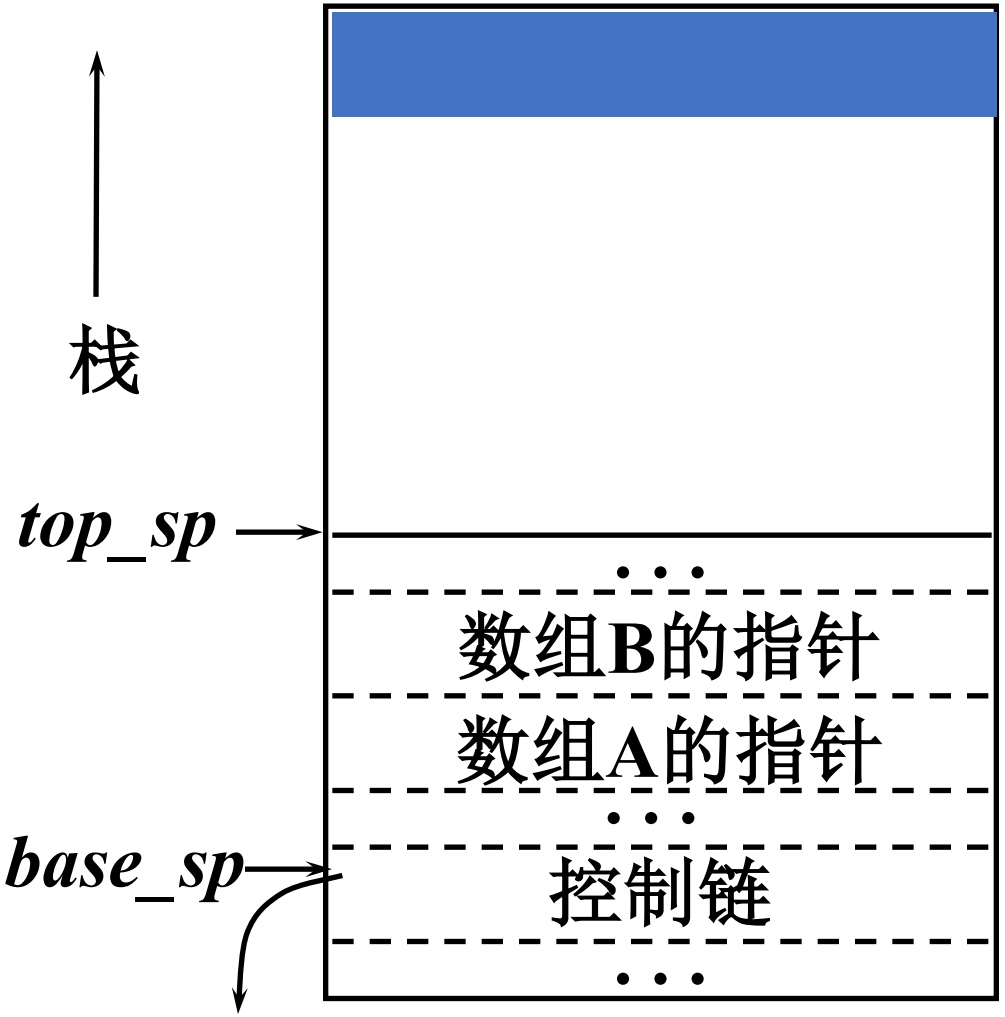
由于C语言采用逆序传递参数，格式串参数将被放在AR中的“固定”位置，即bp+8。而由此参数即可确定待输入值的参数（变量）的个数。从而适应参数个数变化的情况。

全局栈式存储分配

- **栈上可变长数据**
  - 数据对象**的长度在编译时不能确定的情况**
  - 但仅仅为该活动运行过程使用
  - 避免垃圾回收开销
  - **例：局部数组的大小要等到过程激活时才能确定**
- **如何在栈上布局可变长的数组?**
  - 先分配存放数组指针的单元，对数组的访问通过指针间接访问
  - 运行时，这些指针指向分配在栈顶的数组存储空间

全局栈式存储分配

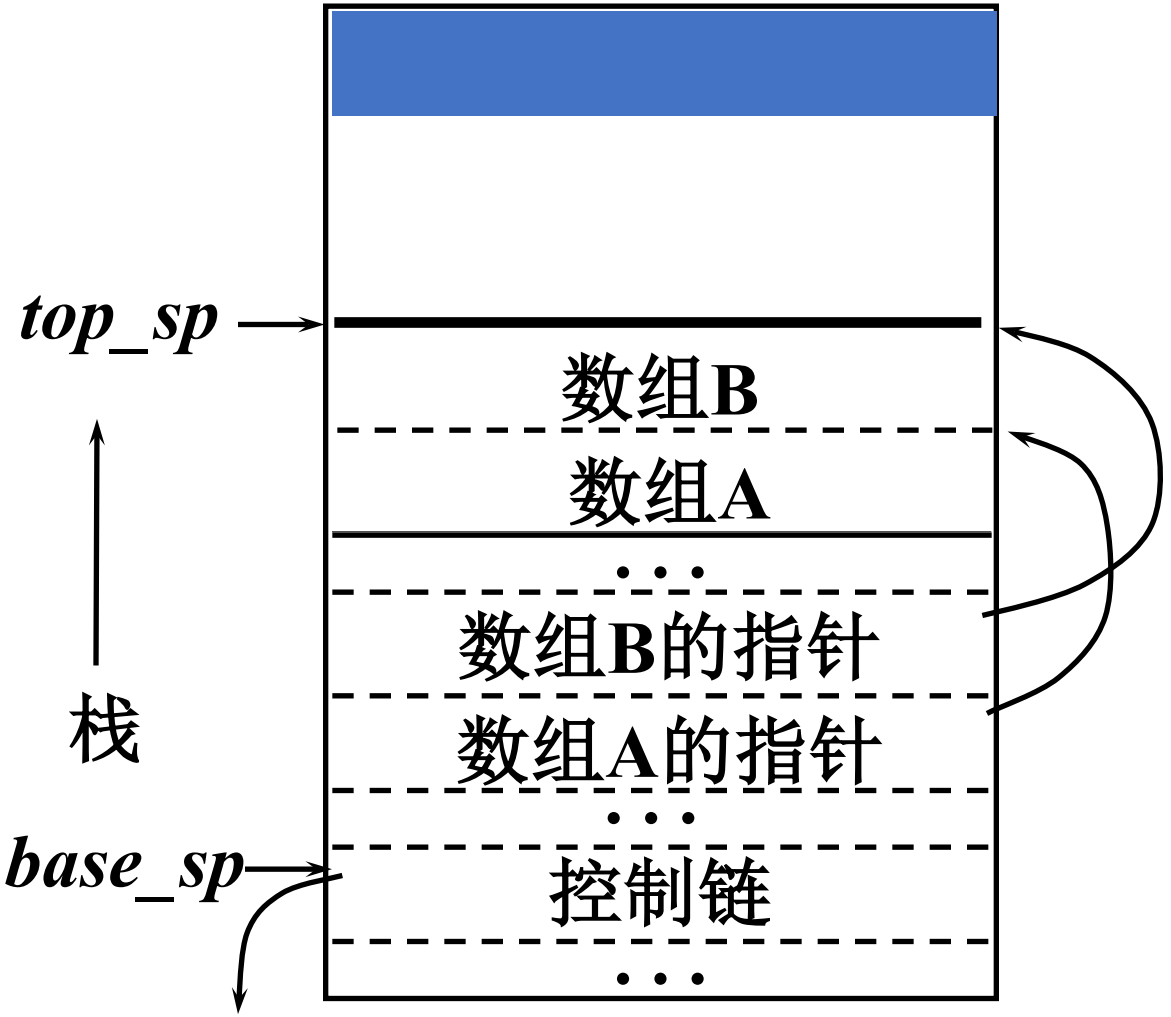
- 访问动态分配的数组



(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

全局栈式存储分配

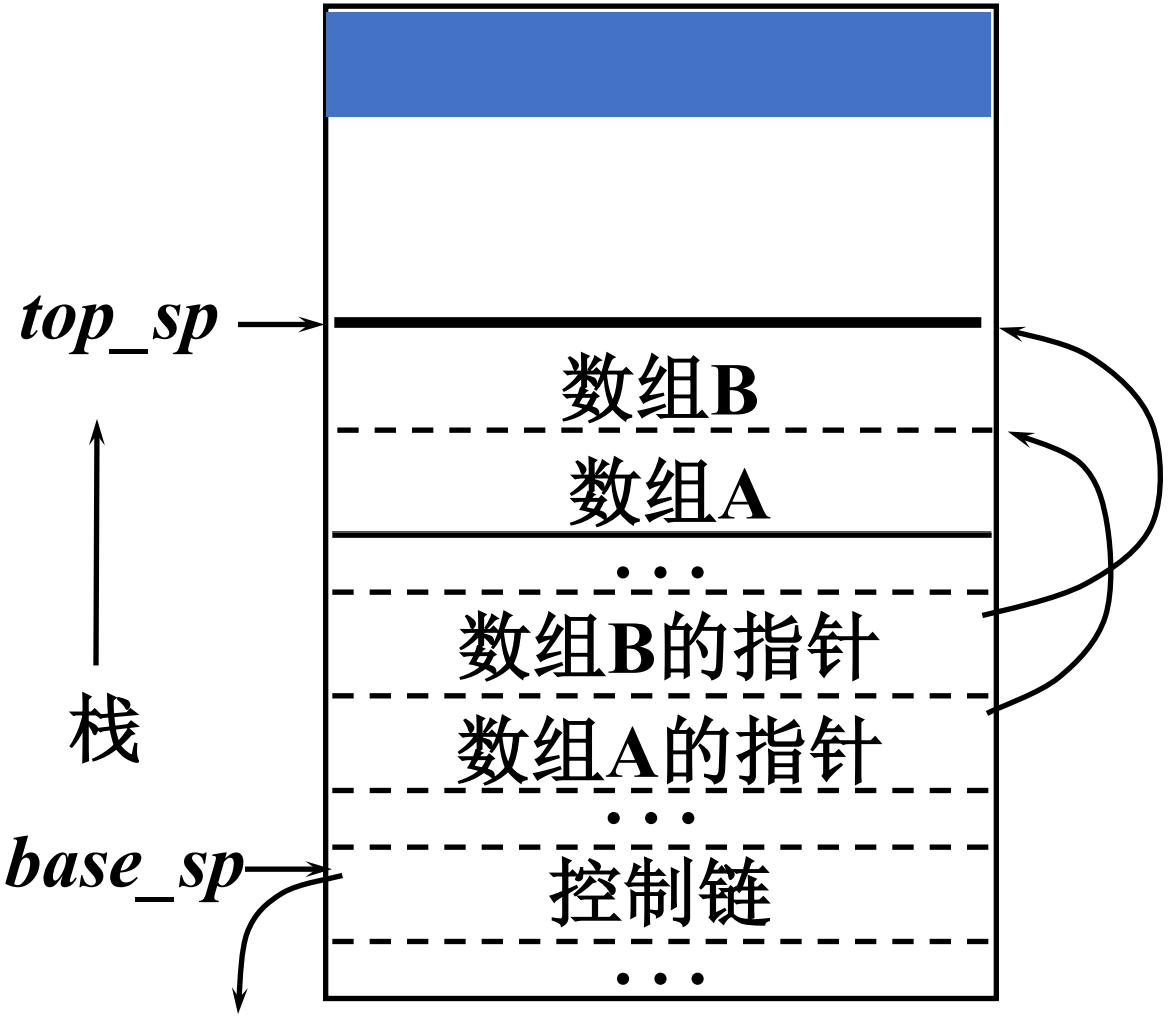
- 访问动态分配的数组



(2) 运行时，这些指针指向分配在栈顶的数组存储空间

全局栈式存储分配

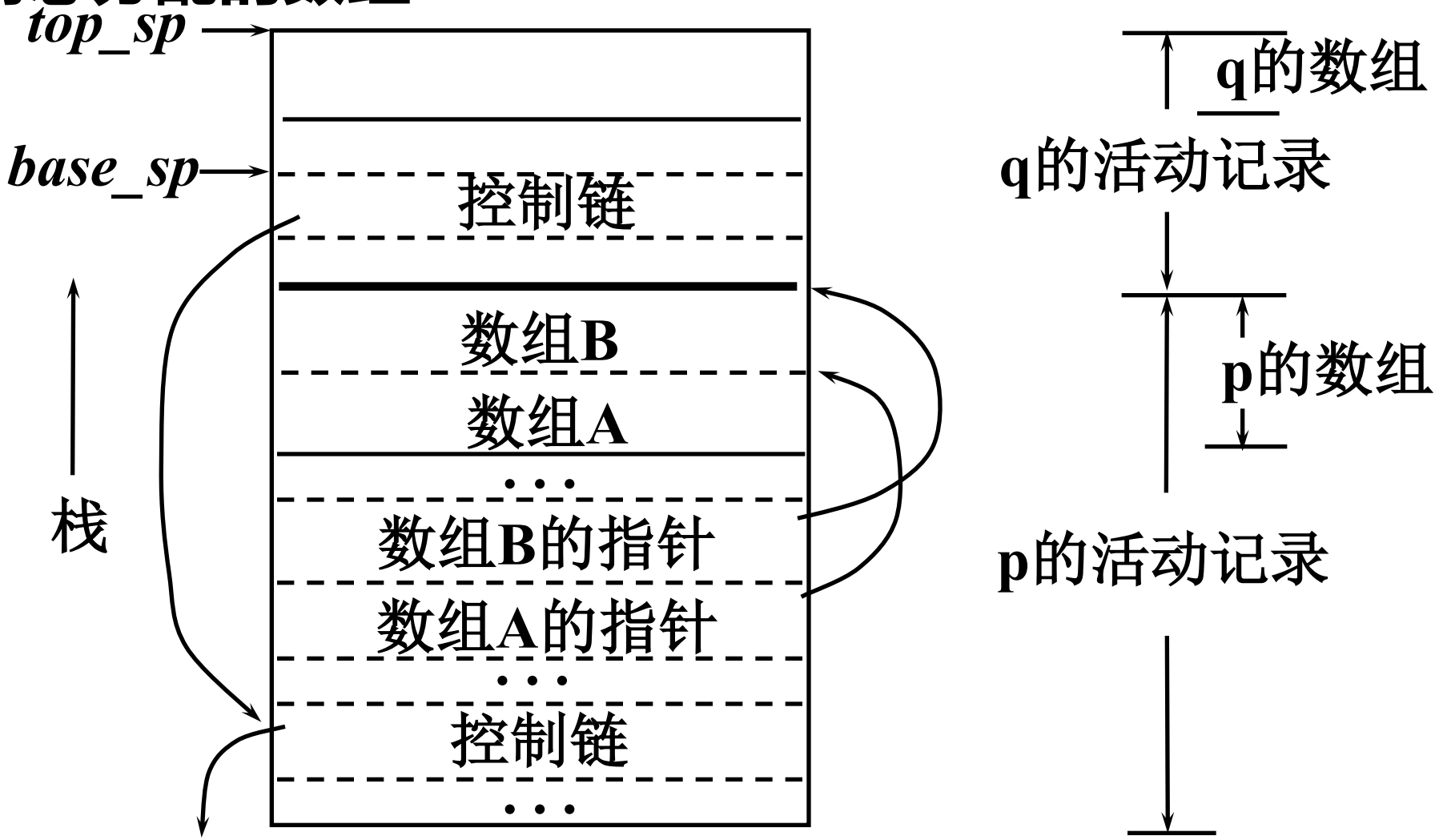
- 访问动态分配的数组



(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

全局栈式存储分配

- 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

例：main中引用p指向的对象

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

一起努力
打造国产基础软硬件体系！

李 诚

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2024年11月04日