



中国科学技术大学
University of Science and Technology of China



《编译原理与技术》

运行时存储空间的组织和管理

计算机科学与技术学院

李 诚

2021-12-1



- 运行时存储空间组织管理概述
- 活动树与栈式空间分配
- 调用序列与返回序列
- 非局部数据的访问

由编译器、操作系统、目标机器共同完成
编译器视角：目标程序运行在逻辑地址空间
操作系统视角：将逻辑地址转换为物理地址
目标机器视角：真正执行指令，访问数据，同时限制了存储空间的组织和数据的访问



□编译器为目标程序创建并管理一个运行时环境，目标程序运行在该环境中。

- ❖对象存储位置和空间的分配
- ❖访问变量的机制
- ❖过程间的连接
- ❖参数传递机制



□ 编译器必须为源程序中出现的**一些数据对象**
分配运行时的存储空间

- ❖ 静态存储分配

- ❖ 动态存储分配

□ 对于那些在编译时刻就可以确定大小的数据对象，可以在编译时刻就为它们分配存储空间，这样的策略成为**静态存储分配**

- ❖ 比较简单，所以不展开



□如果**不能**在编译时刻**完全确定**数据对象的**大小**，就要采用**动态存储分配**的策略。即，在编译时刻仅产生各种必要的信息，而在运行时刻，再动态地分配存储空间。

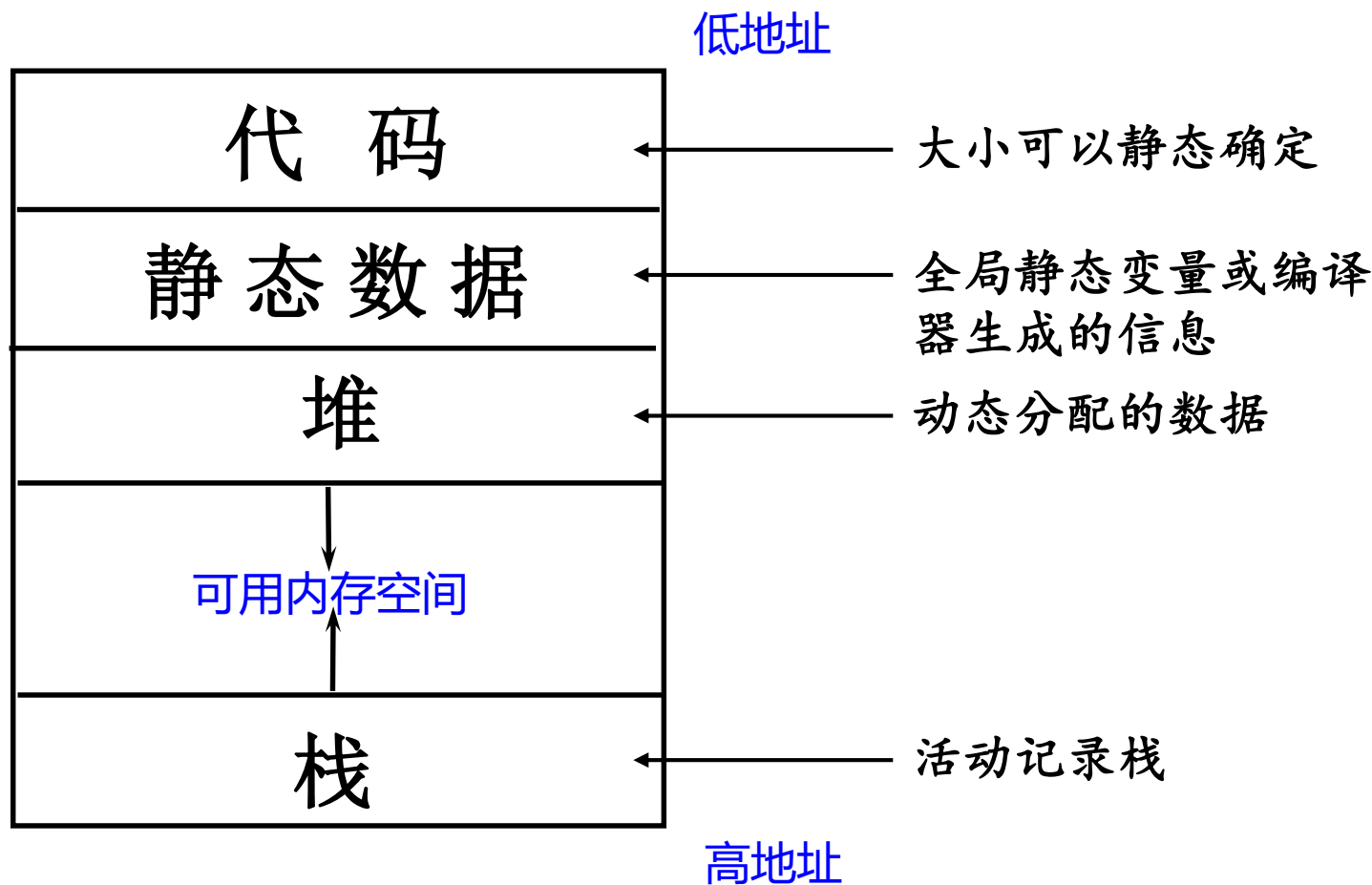
- ❖ 栈式存储分配

- ❖ 堆式存储分配

□**静态**和**动态**这两个概念分别对应**编译时刻**和**运行时刻**



程序的存储分配





- 过程能否递归
- 当控制从过程的活动返回时, 局部变量的值是否要保留
- 过程能否访问非局部变量
- 过程调用的参数传递方式
- 过程能否作为参数被传递
- 过程能否作为结果值传递
- 存储块能否在程序控制下被动态地分配
- 存储块是否必须被显式地释放



□过程

- FORTRAN的子例程(subroutine)
- PASCAL的过程/函数(procedure/function)
- C的函数

□过程的激活（调用）与终止（返回）

□过程的执行需要：

代码段 + 活动记录（过程运行所需的额外信息，如参数，局部数据，返回地址等）



□ **基本概念：作用域与生存期**

□ **活动记录的常见布局**

❖ 字节寻址、类型、次序、对齐

□ **程序块：同名情况的处理**



□名字的作用域

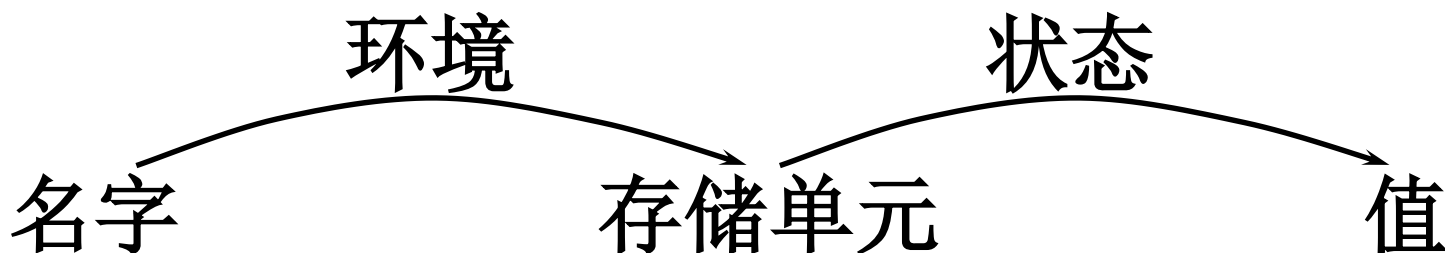
- ❖ 一个声明起作用的程序部分称为该声明的**作用域**
 - ❖ 即使一个名字在程序中只声明一次，该名字在程序运行时也可能表示不同的数据对象
- 如下图代码中的n

```
int f(int n){  
    if (n<0) error("arg<0");  
    else if (n==0) return 1;  
    else return n*f(n-1);  
}
```



□环境和状态

- ❖ 环境把名字映射到左值，而状态把左值映射到右值（即名字到值有两步映射）
- ❖ 赋值改变状态，但不改变环境
- ❖ 过程调用改变环境
- ❖ 如果环境将名字 x 映射到存储单元 s ，则说 x 被绑定到 s





□ 静态概念和动态概念的对应

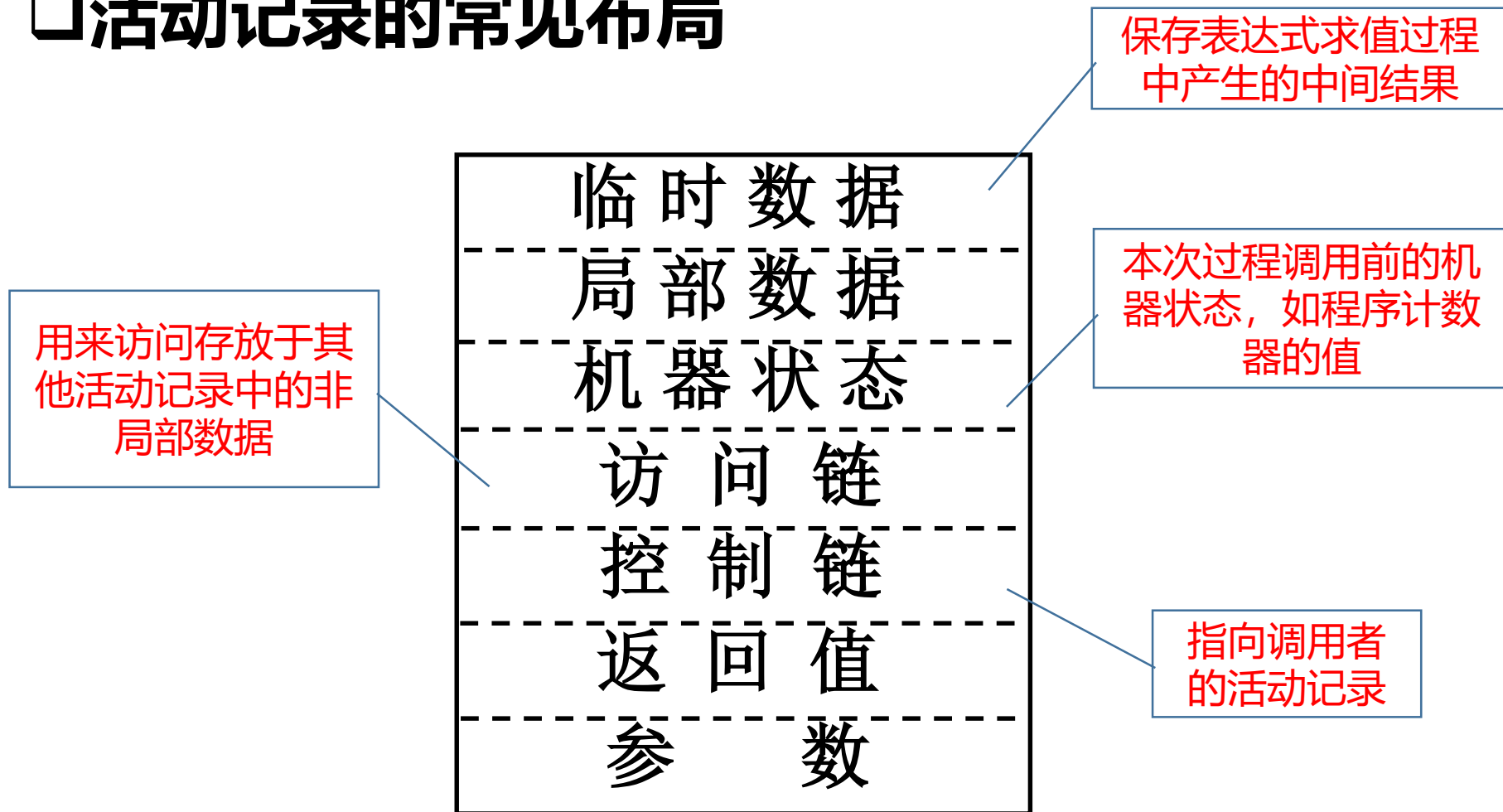
静 态 概 念	动 态 对 应
过程的定义	过程的活动
名字的声明	名字的绑定
声明的作用域	绑定的生存期



- 使用过程(或函数、方法)作为用户自定义动作的单元的语言，其编译器通常以过程为单位分配存储空间
- 过程体的每次执行成为该过程的一个活动
- 编译器为每一个活动分配一块连续存储区域，用来管理此次执行所需的信息，这片区域称为活动记录(activation record)



□活动记录的常见布局





□局部数据的布局

- ❖ 字节是可编址内存的最小单位
- ❖ 变量所需的存储空间可以根据其类型而静态确定
- ❖ 一个过程所声明的局部变量，按这些变量声明时出现的次序，在局部数据域中依次分配空间
- ❖ 局部数据的地址可以用相对于活动记录中某个位置的地址来表示
- ❖ 数据对象的存储布局还有一个对齐问题



□例 在SPARC/Solaris工作站上下面两个结构体的size分别是24和16，为什么不一样？

```
typedef struct _a{
```

```
    char  c1;
```

```
    long  i;
```

```
    char  c2;
```

```
    double f;
```

```
}a;
```

```
typedef struct _b{
```

```
    char c1;
```

```
    char c2;
```

```
    long i;
```

```
    double f;
```

```
}b;
```

对齐： char : 1, long : 4, double : 8



□例 在SPARC/Solaris工作站上下面两个结构体的size分别是24和16, 为什么不一样?

```
typedef struct _a{
```

```
    char  c1;    0
```

```
    long  i;     4
```

```
    char  c2;    8
```

```
    double f;   16
```

```
}a;
```

```
typedef struct _b{
```

```
    char c1;    0
```

```
    char c2;    1
```

```
    long i;     4
```

```
    double f;   8
```

```
}b;
```

对齐: char : 1, long : 4, double : 8



□例 在x86/Linux机器的结果和SPARC/Solaris工作站不一样，是20和16。

```
typedef struct _a{
```

```
    char  c1;    0
```

```
    long  i;     4
```

```
    char  c2;    8
```

```
    double f;   12
```

```
}a;
```

```
typedef struct _b{
```

```
    char c1;    0
```

```
    char c2;    1
```

```
    long i;     4
```

```
    double f;   8
```

```
}b;
```

对齐: char : 1, long : 4, double : 4



□程序块

- ❖ 本身含有局部变量声明的语句
- ❖ 可以嵌套
- ❖ 最接近的嵌套作用域规则
- ❖ 并列程序块不会同时活跃
- ❖ 并列程序块的变量可以重叠分配



```
main() /* 例 */
```

```
{
```

```
    int a = 0;
```

```
    int b = 0;
```

```
    {
```

```
        int b = 1;
```

```
        {
```

```
            int a = 2;
```

```
        }
```

```
        {
```

```
            int b = 3;
```

```
        }
```

```
    }
```

```
}
```

```
/* begin of  $B_0$  */
```

```
/* begin of  $B_1$  */
```

```
/* begin of  $B_2$  */
```

```
/* end of  $B_2$  */
```

```
/* begin of  $B_3$  */
```

```
/* end of  $B_3$  */
```

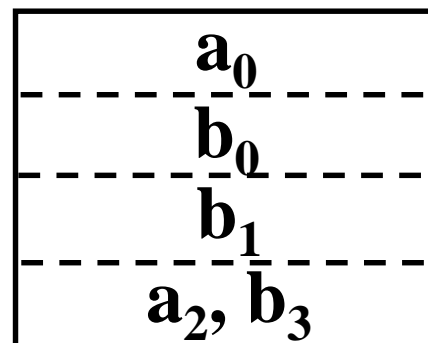
```
/* end of  $B_1$  */
```

```
/* end of  $B_0$  */
```



```
main() /* 例 */
{ /* begin of  $B_0$  */
  int a = 0;
  int b = 0;
  { /* begin of  $B_1$  */
    int b = 1;
    { /* begin of  $B_2$  */
      int a = 2;
    } /* end of  $B_2$  */
    { /* begin of  $B_3$  */
      int b = 3;
    } /* end of  $B_3$  */
  } /* end of  $B_1$  */
} /* end of  $B_0$  */
```

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





- 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



□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



□主要有两种策略

- ❖ 栈式存储：与过程调用返回有关，涉及过程的局部变量以及过程活动记录
- ❖ 堆存储：关系到部分生存周期较长的数据



- 运行时存储空间组织管理概述
- 活动树与栈式空间分配
- 调用序列与返回序列
- 非局部数据的访问



- 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**



- 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

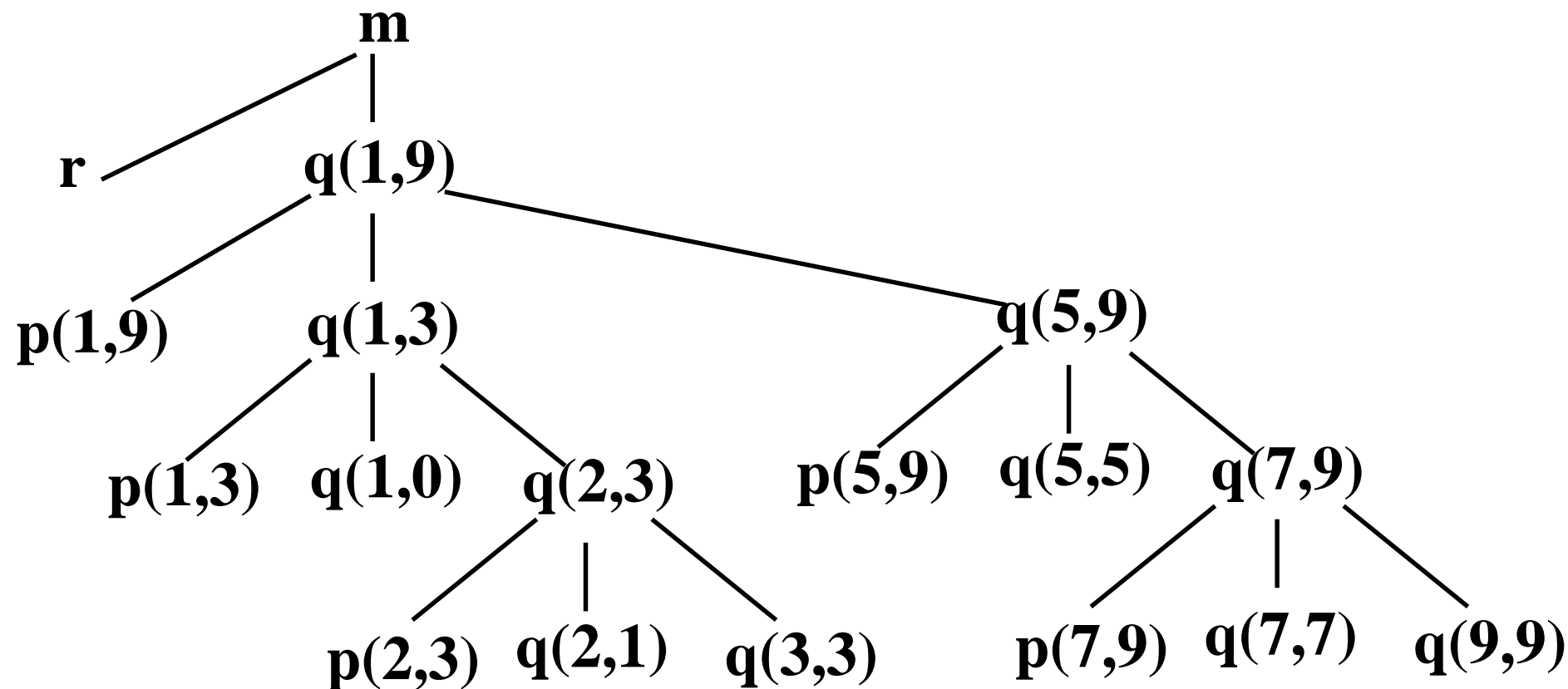


```
int a[11];
void readArray() /*将9个数读入a[1],...,a[9]中*/
{ int i; ...}
int partition(int m, int n)
{/*选择一个分割值v, 划分a[m,...,n], 使得a[m... p-1]小于v, a[p]=v, a[p+1...n]
大于v, 返回p*/
...}
void quicksort(int m, int n)
{ int i;
  if(n>m){
    i = partition(m, n);
    quicksort(m, i-1);
    quicksort(i+1, n);}
main(){
  readArray();
  a[0] = -9999;
  a[10] = 9999;
  quicksort(1,9);}
```



□活动树

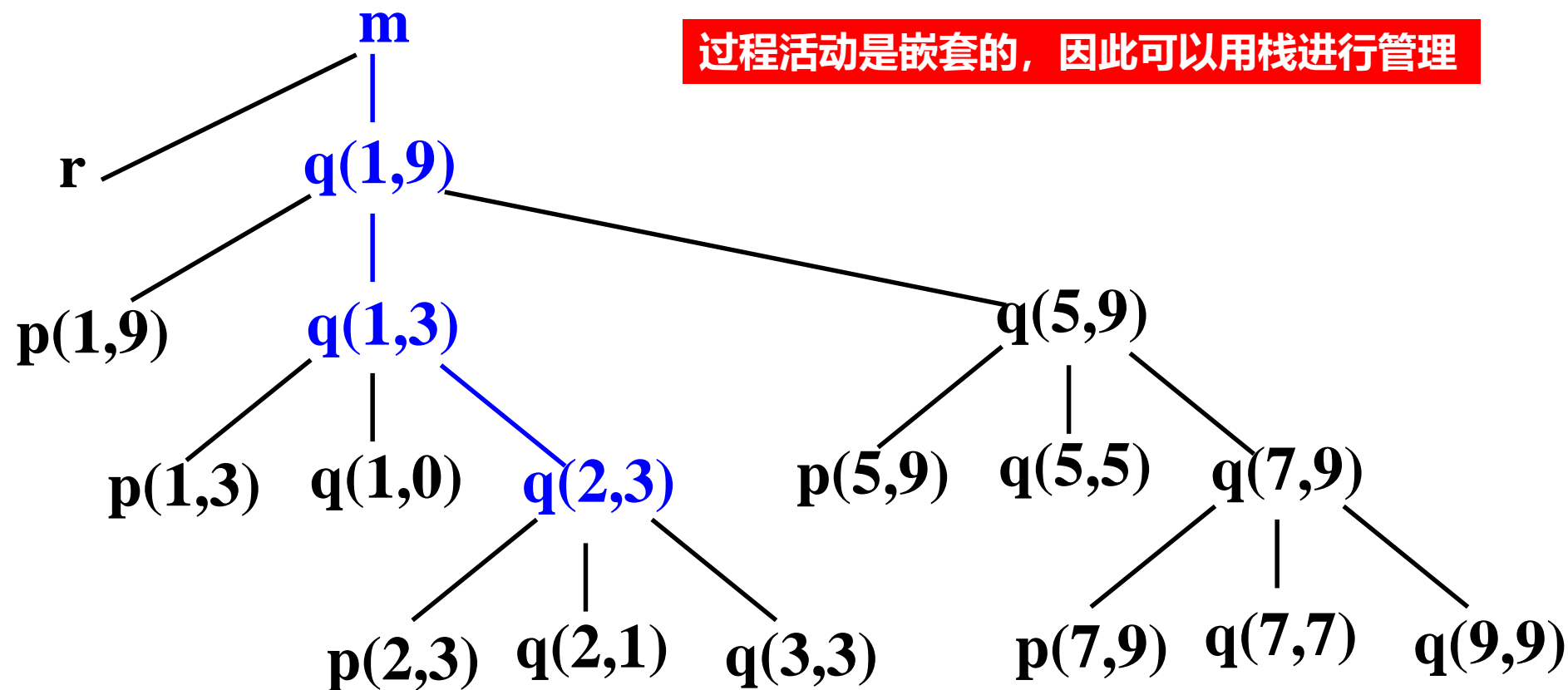
❖ 用树来描绘控制进入和离开活动的方式





□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： $m, q(1, 9), q(1, 3), q(2, 3)$

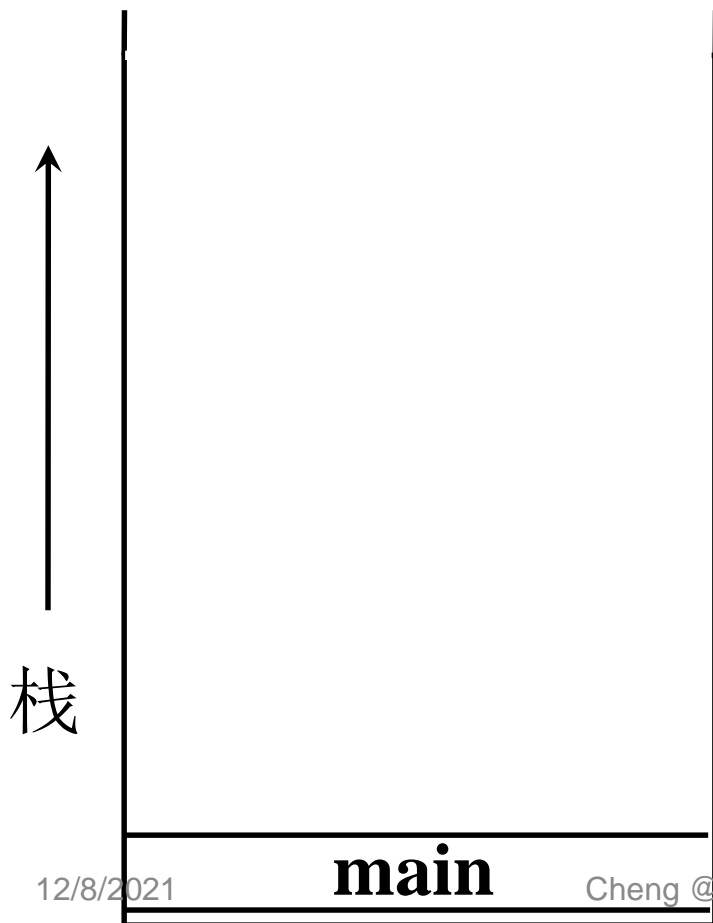




□运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



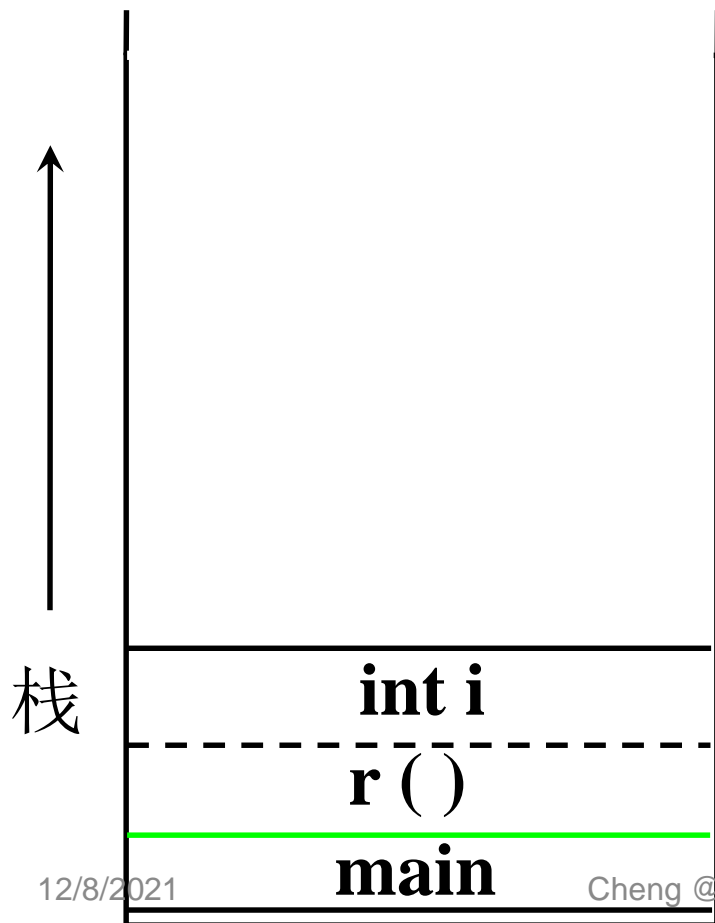
□运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



函数调用关系树
main

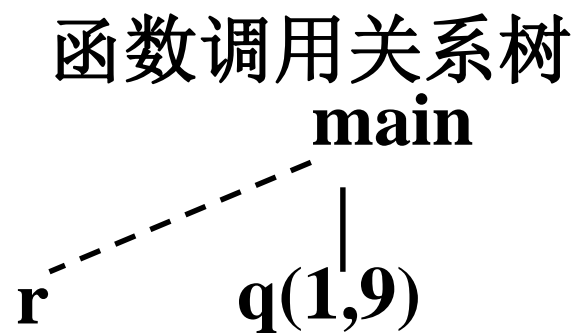
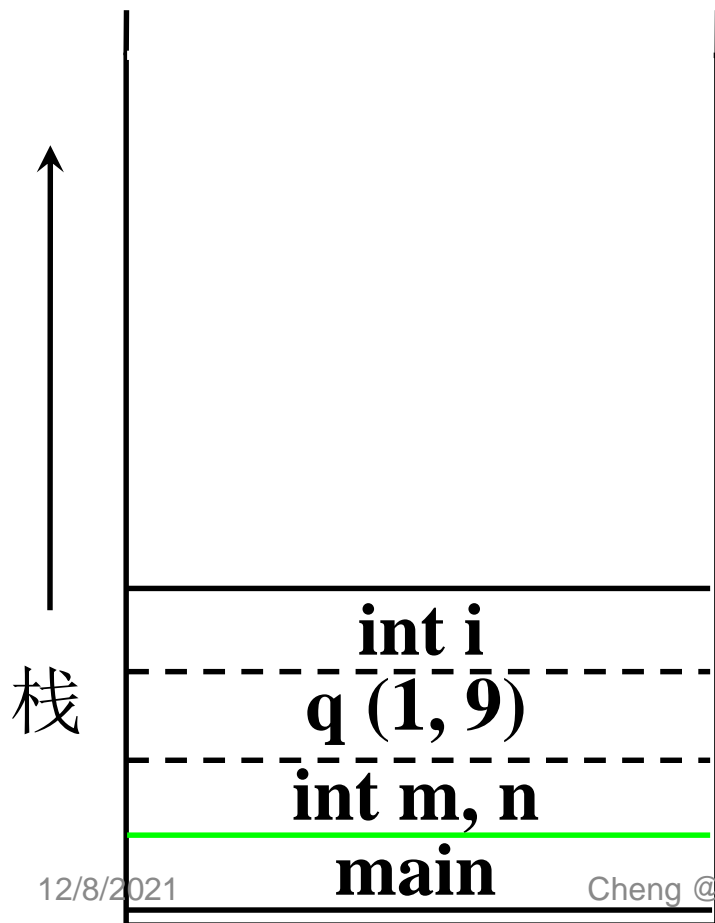


□运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



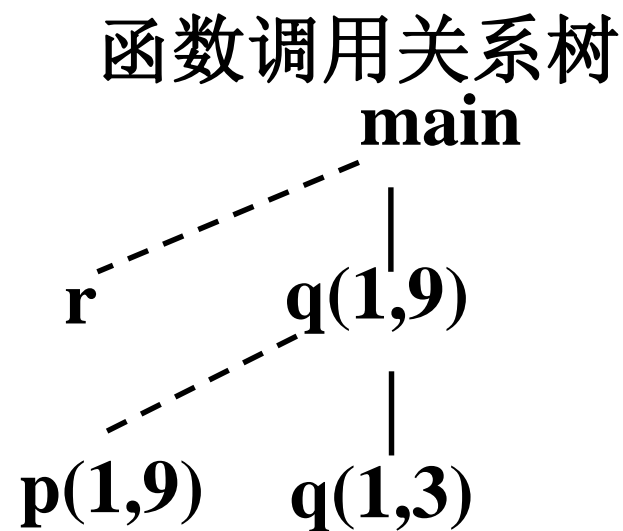
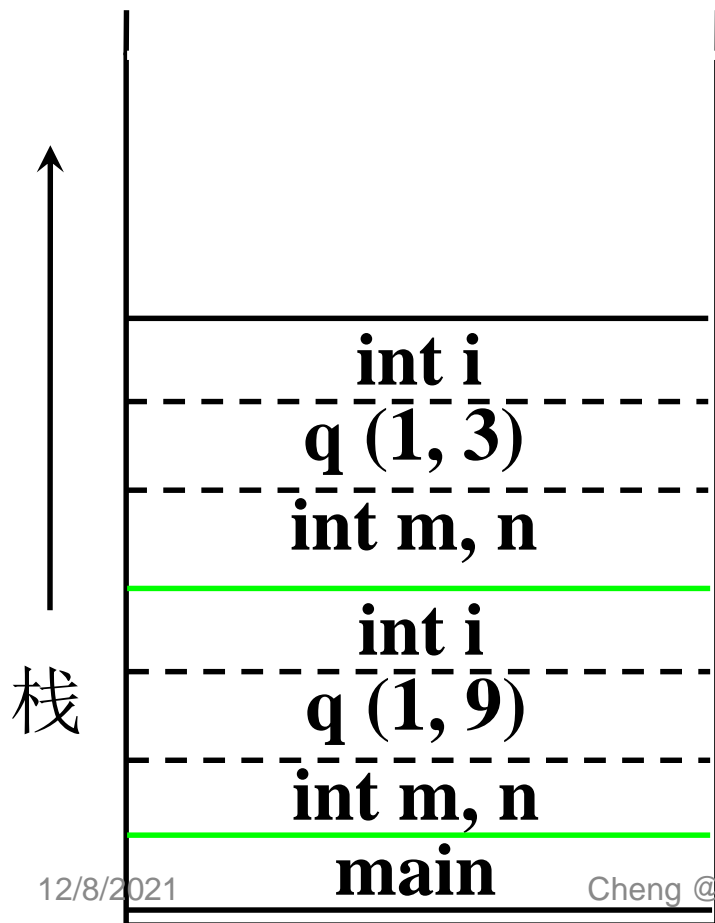


□运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



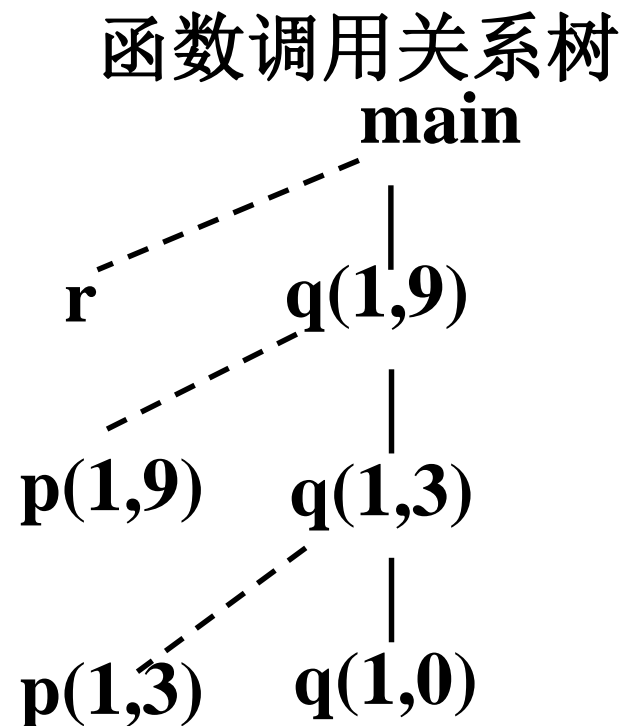
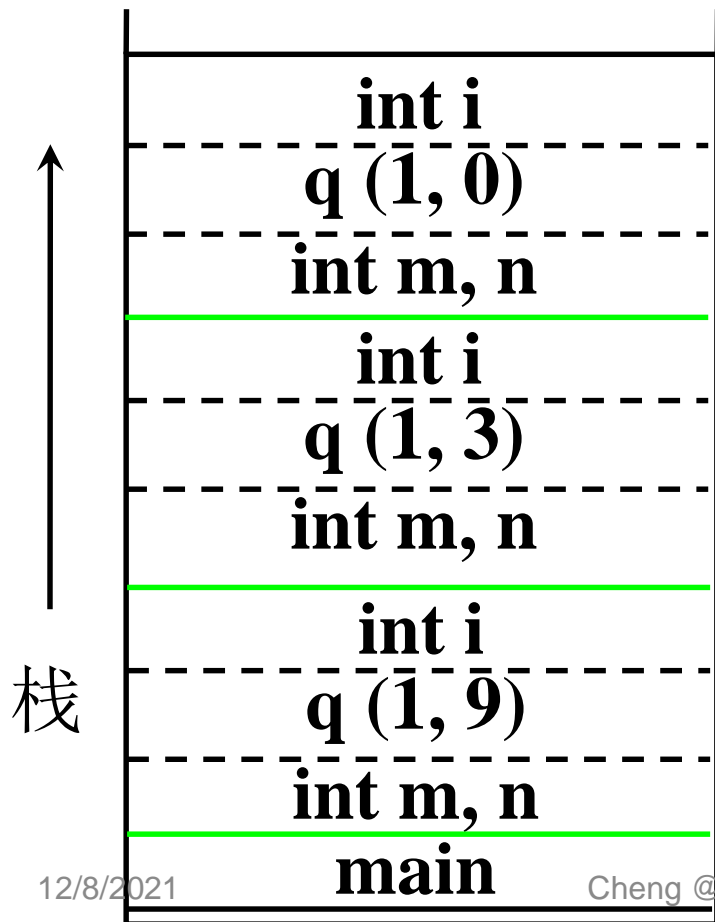


□运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



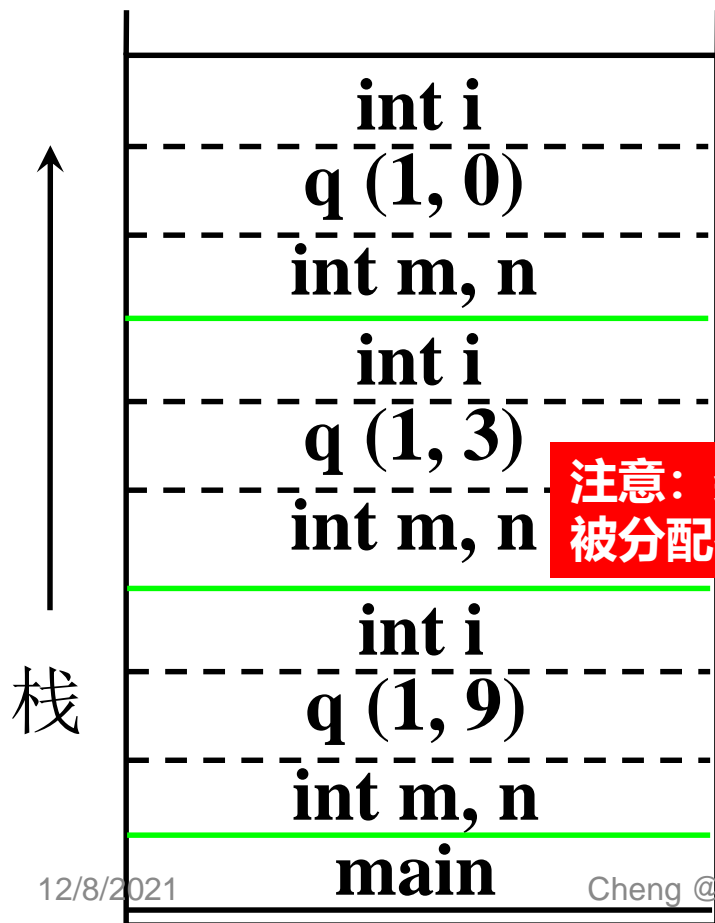


□运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）

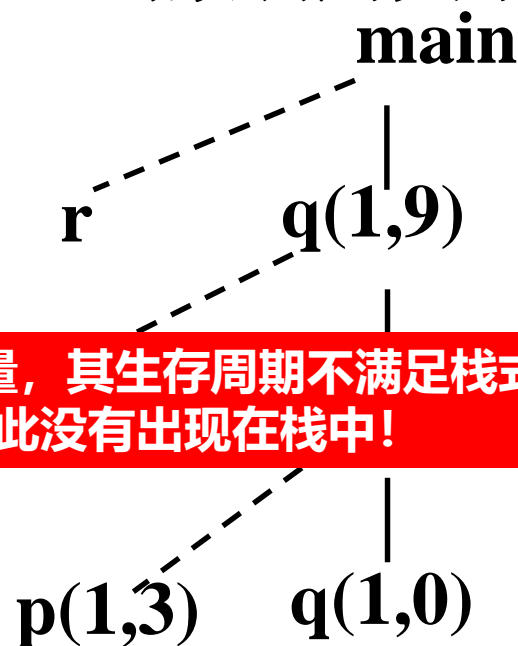




□运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



函数调用关系树



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！



- 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**



- 运行时存储空间组织管理概述
- 活动树与栈式空间分配
- 调用序列与返回序列
- 非局部数据的访问



□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



□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



□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**



□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





□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



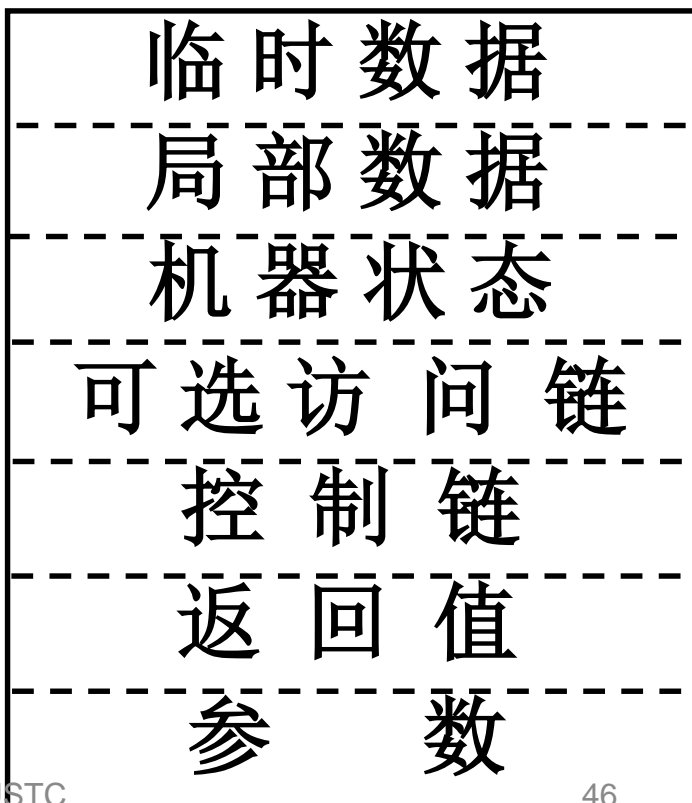


□即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异

□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



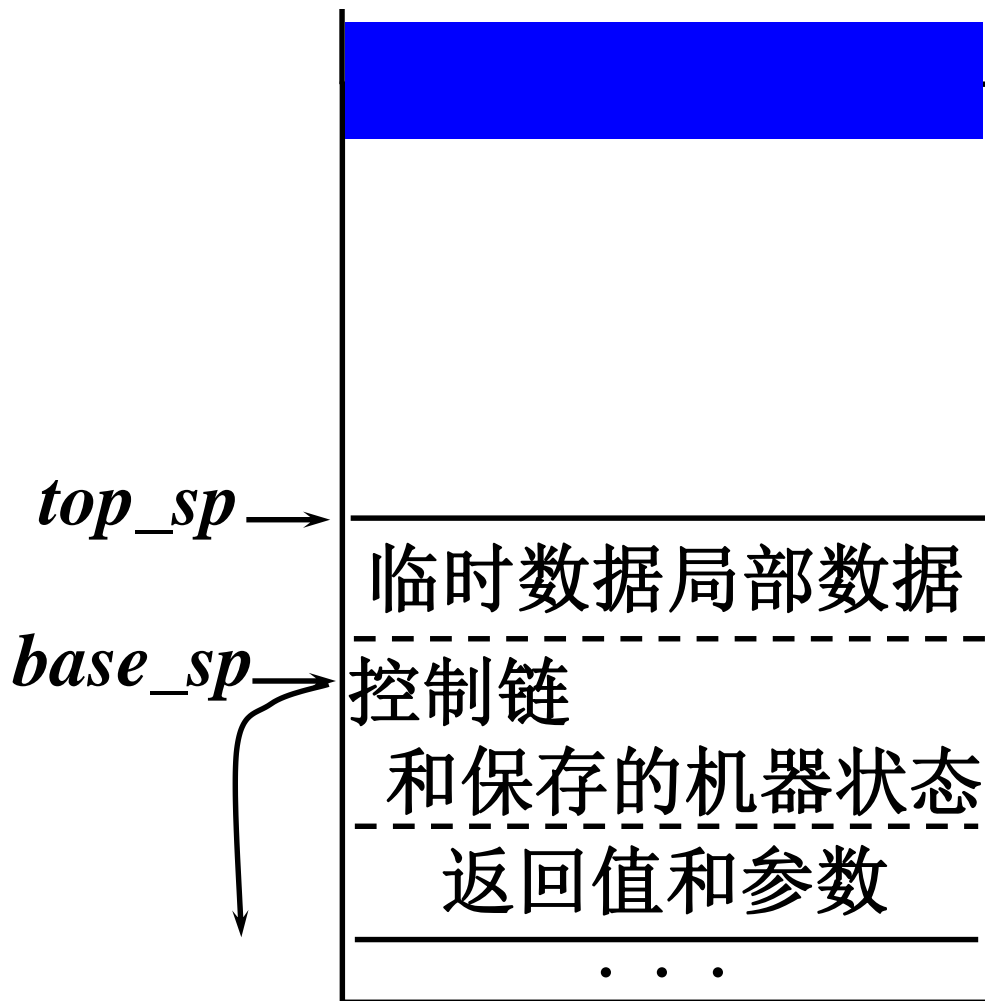


□即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异

□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

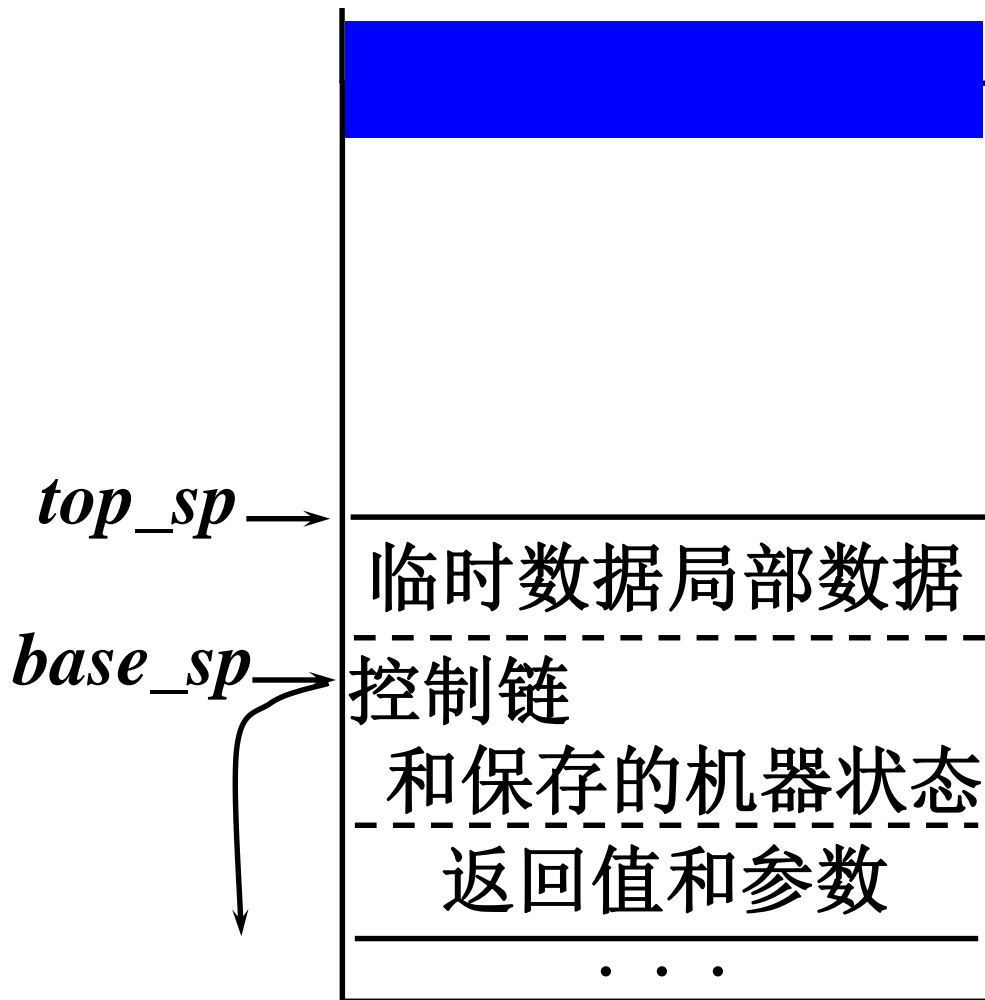




- ❖ **top_sp**: 栈顶寄存器，如esp、rsp。指向栈顶活动记录的末端
- ❖ **base_sp**: 基址寄存器，如ebp、rbp。指向栈顶活动记录中控制链所在位置。

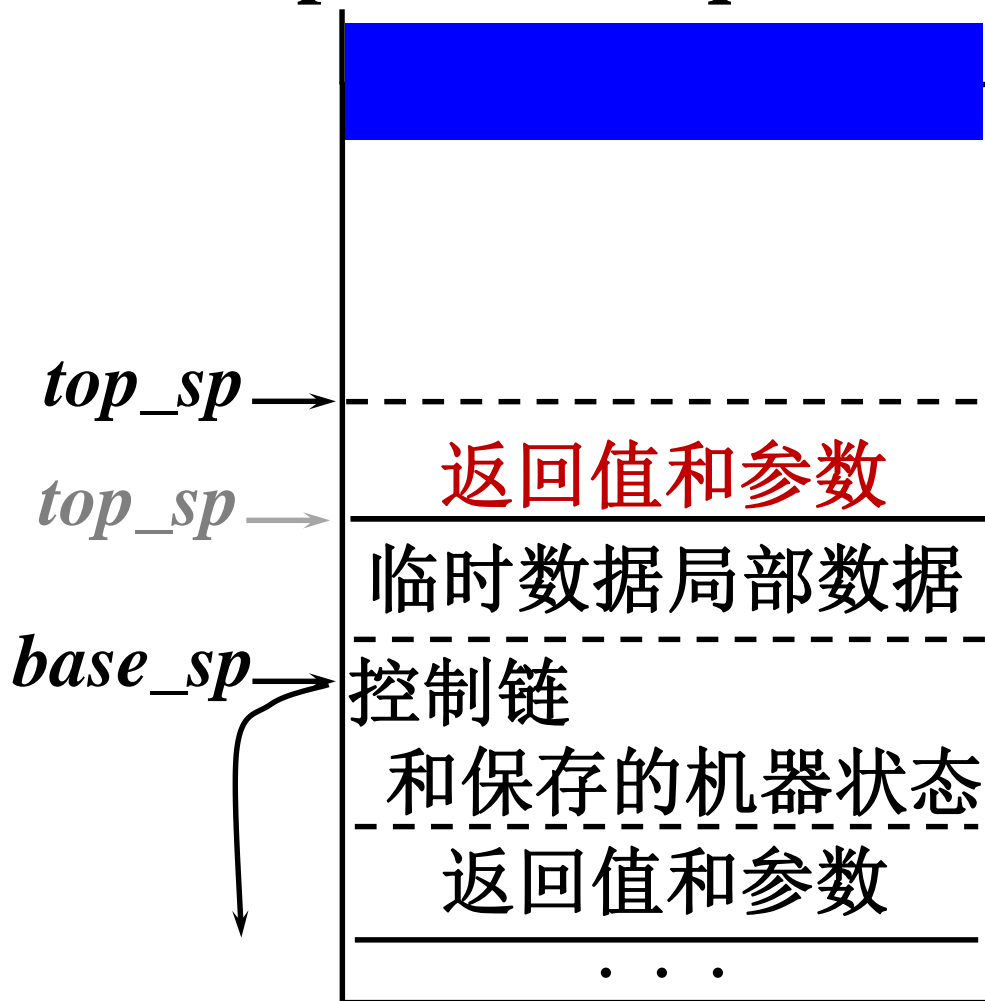


□ 过程p调用过程q的调用序列(栈往上增长)





□ 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



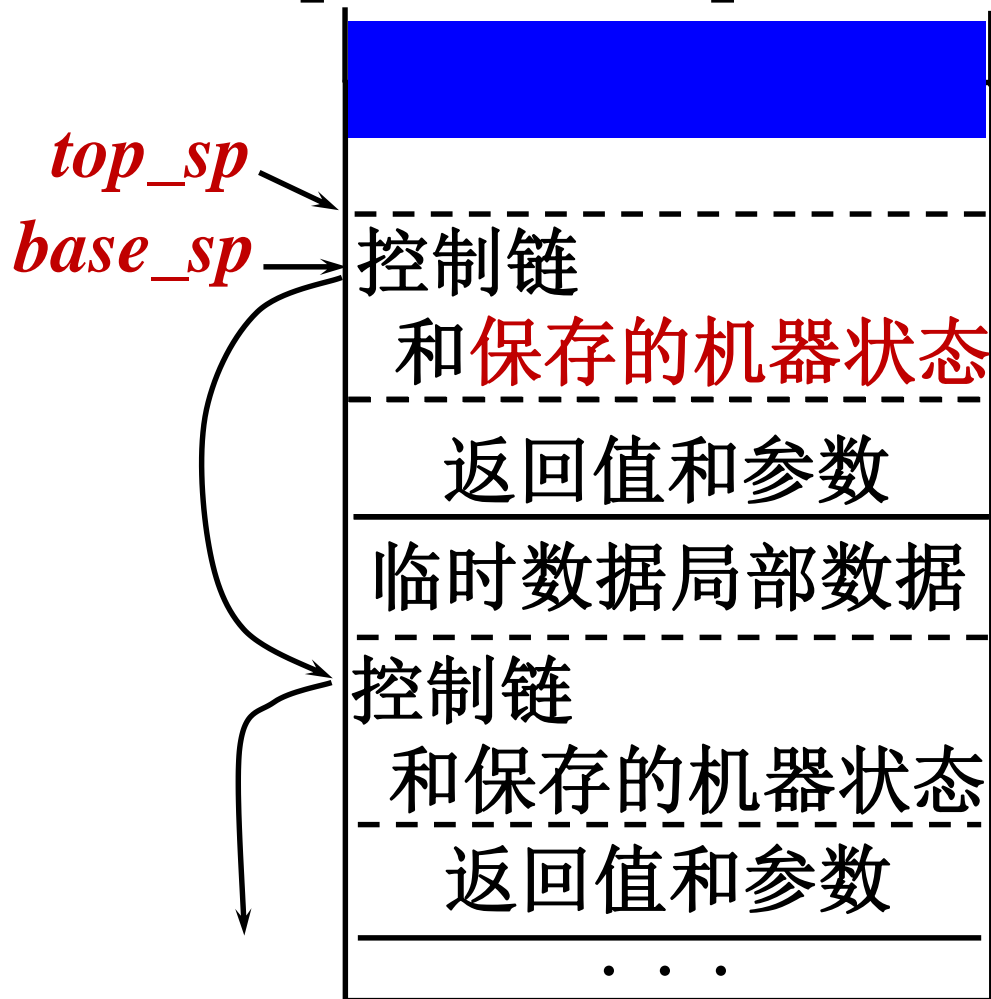
□过程p调用过程q的调用序列



(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



□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



□ 过程p调用过程q的调用序列

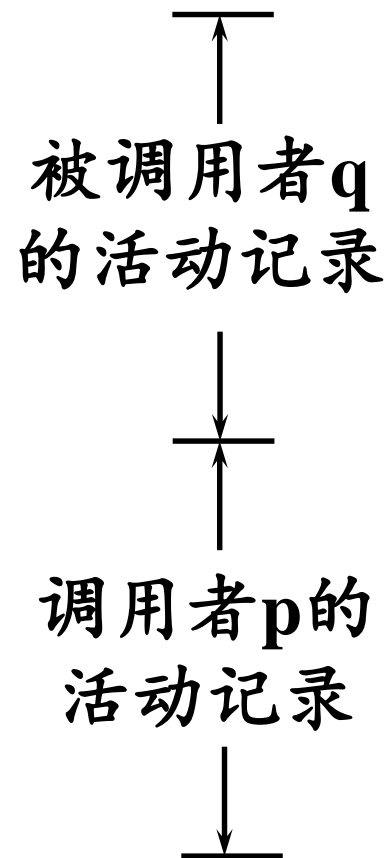
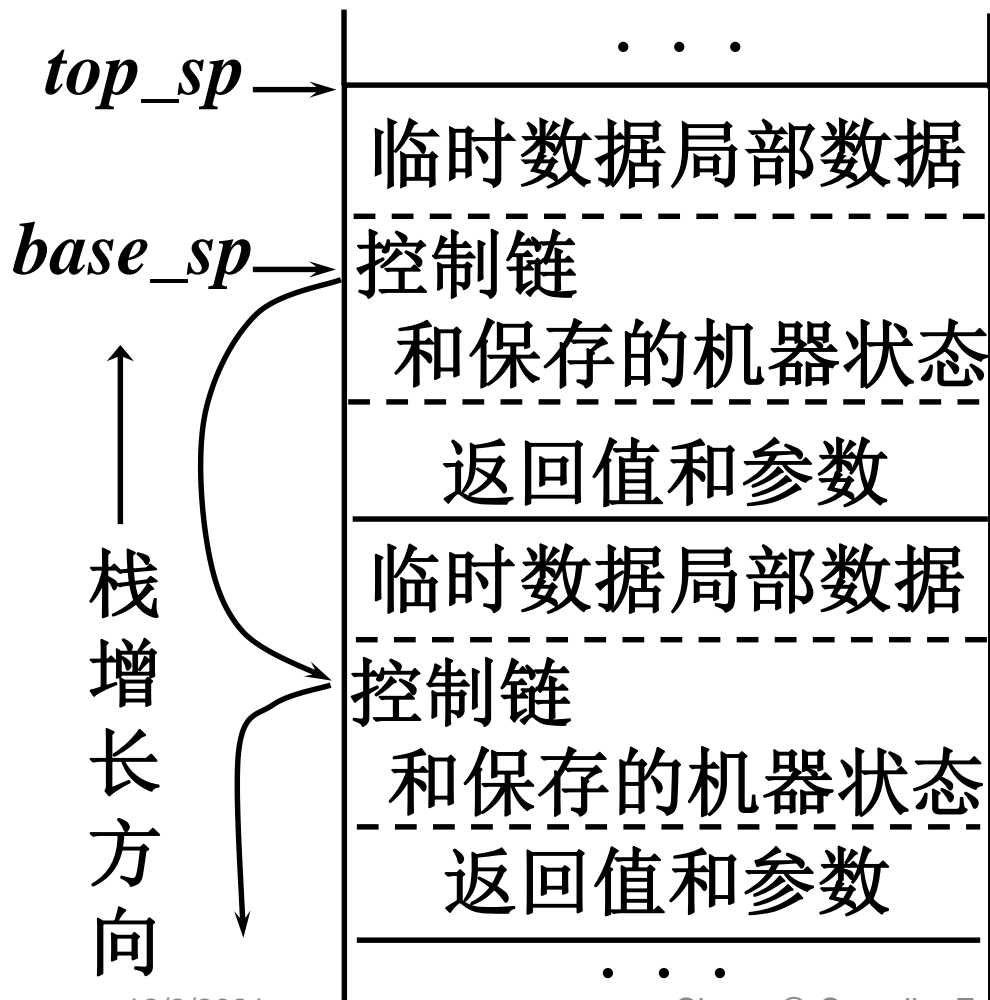


base_sp不变, 指向活动记录中间

(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**

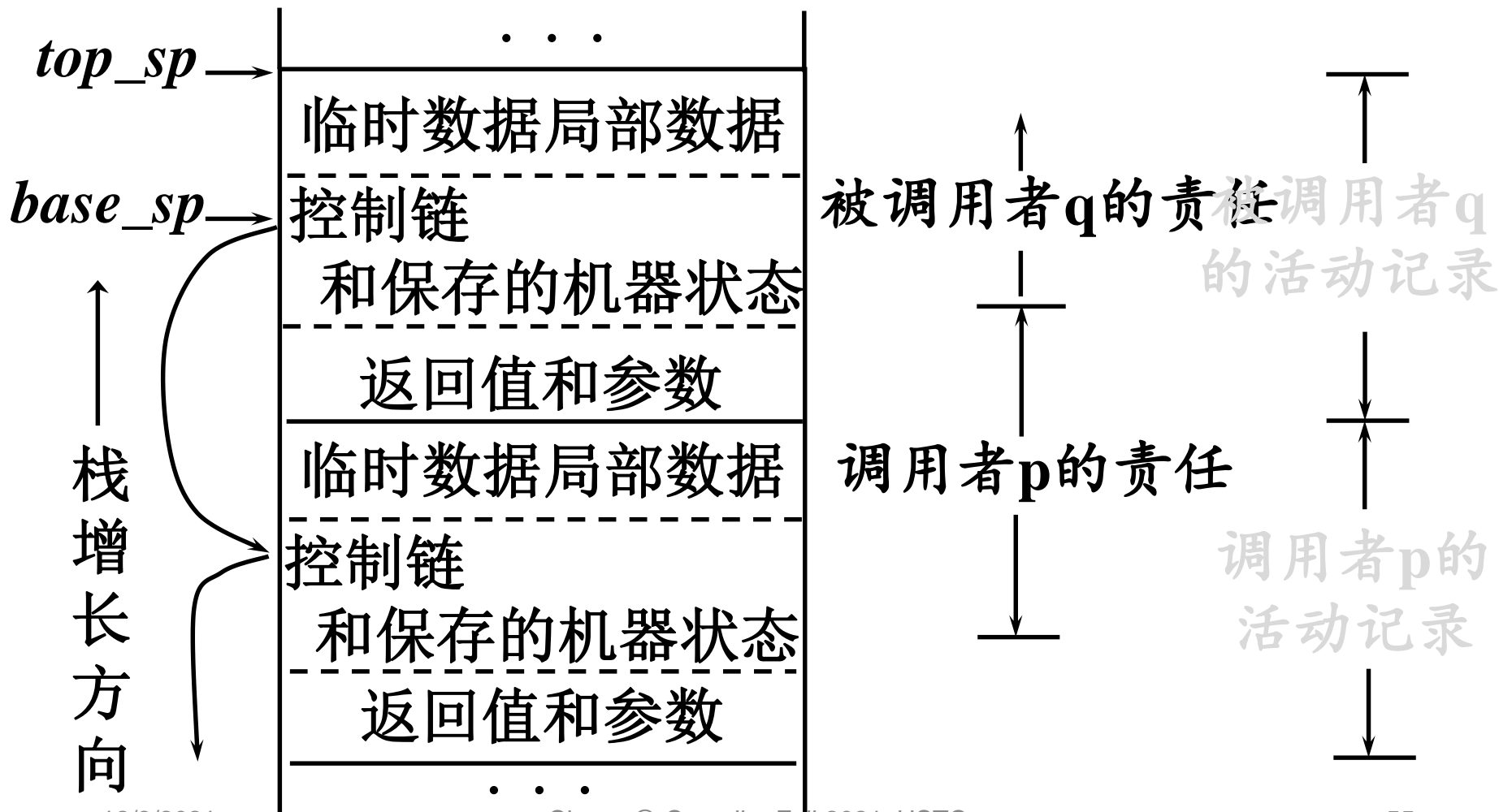


□调用者p和被调用者q之间的任务划分



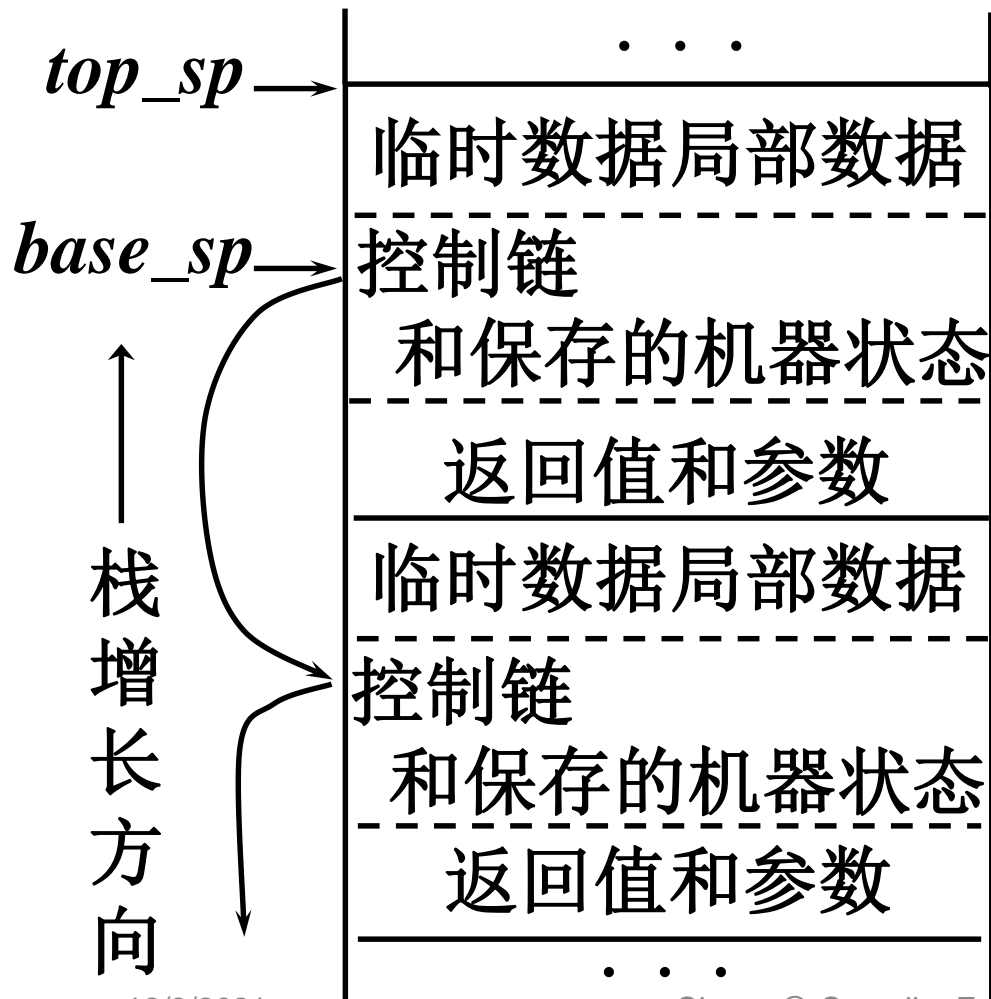


□调用者p和被调用者q之间的任务划分



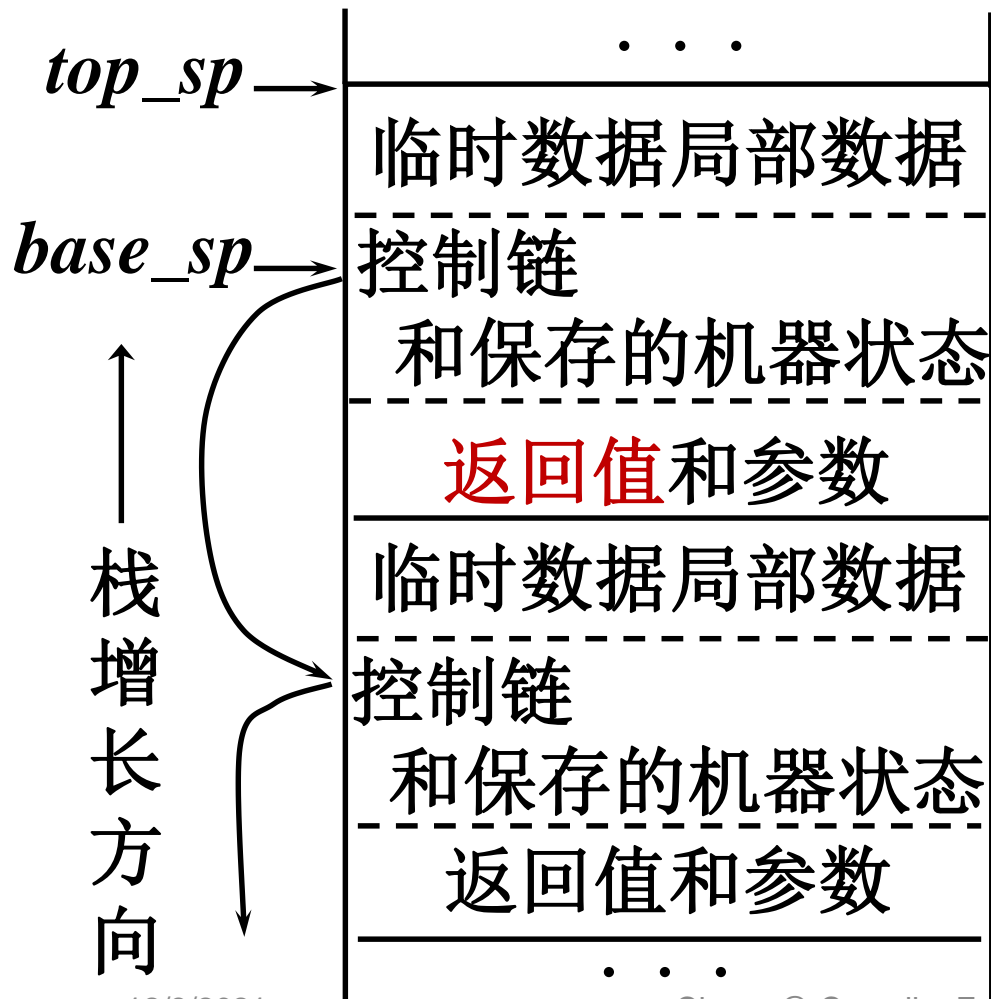


□ 过程p调用过程q的返回序列





□过程p调用过程q的返回序列

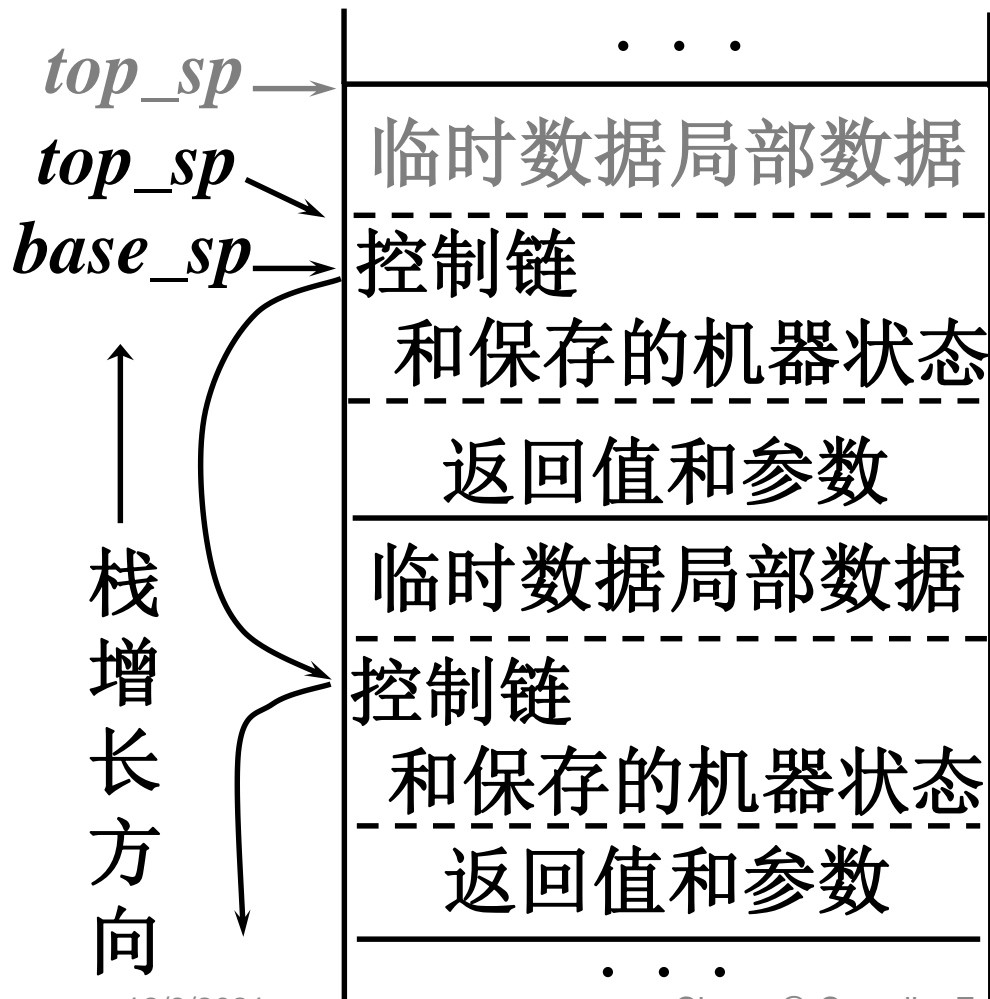


(1) q把返回值置入邻近p的活动记录的地方

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



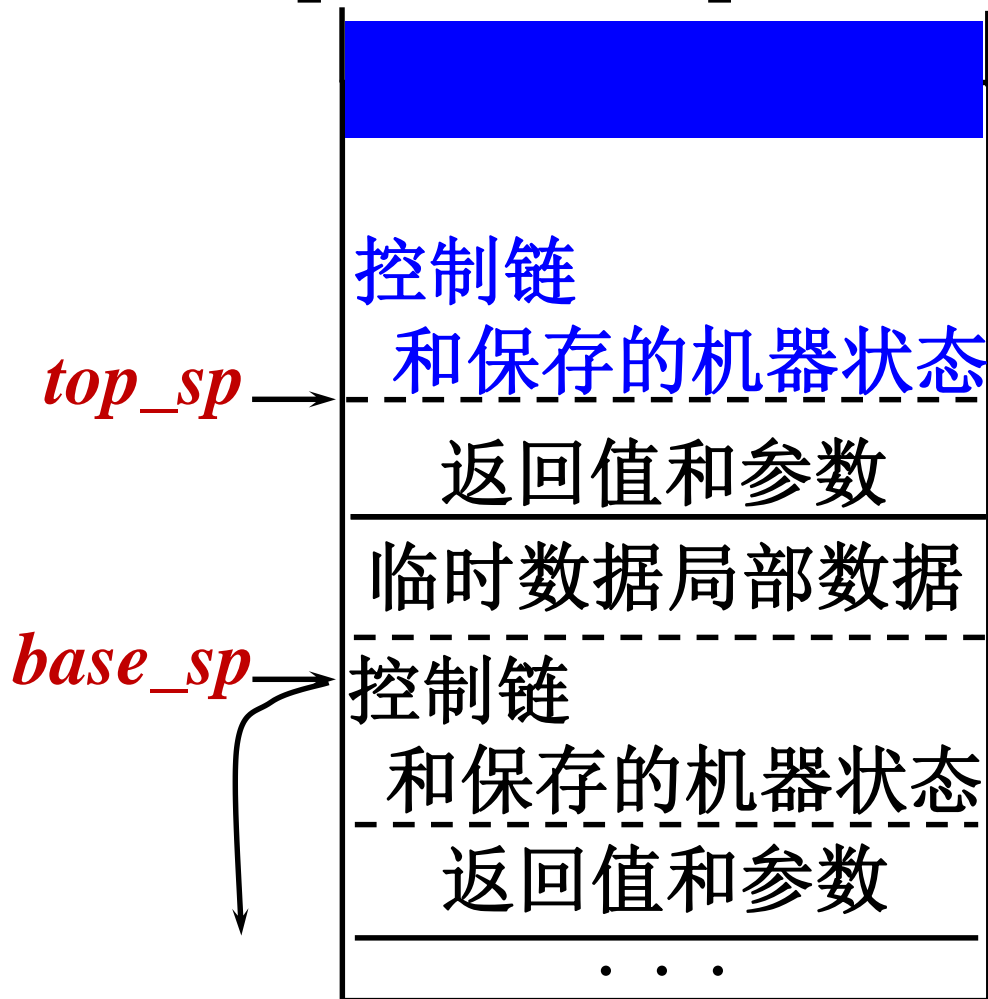
□ 过程p调用过程q的返回序列



(2) q对应调用序列的步骤(4), 增加 top_sp 的值



□ 过程p调用过程q的返回序列

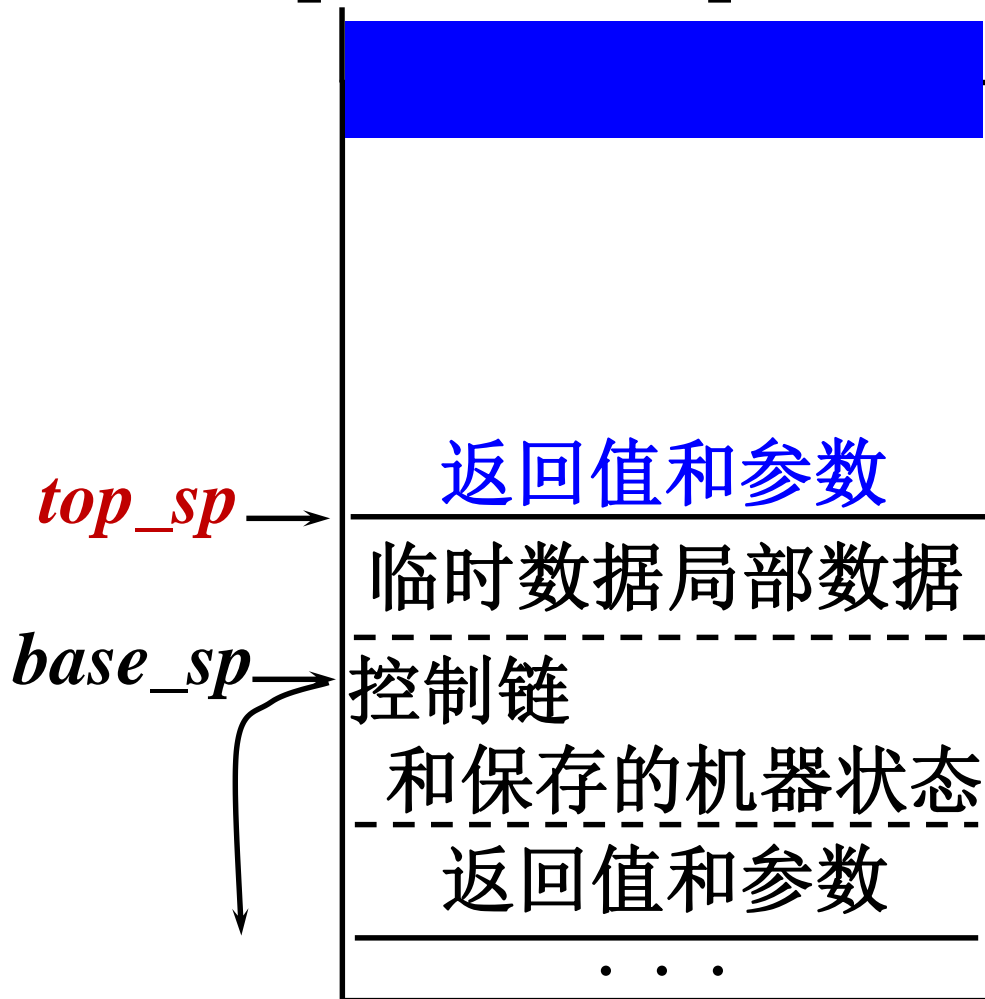


(3) q恢复寄存器(包括 *base_sp*)和机器状态, 返回p

控制权转到p



□ 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



中国科学技术大学
University of Science and Technology of China

华夏英才班学术讲座 (No.8)

龙芯指令集的设计与优化



报告人：张福新（中科院计算所微处理器中心副主任、
龙芯实验室主任）

时 间：2021年12月8日 16:00

地 点：线上 腾讯会议：451-428-981

报告摘要：

本报告介绍我国自主设计的龙芯指令集体系结构的情况，包括它的设计原则、设计特色、设计过程以及一些性能分析和优化工作。

报告人简介：

张福新，博士，男，福建永定人，1976年6月生，中国科学院计算技术研究所正高工、博士生导师。长期致力于国产龙芯 CPU 芯片的研发和产业化工作，是龙芯团队骨干人员之一。曾先后主持承担江苏省重大科技成果转化专项资金项目、国家科技部 863 重点项目、发改委工业项目和国务院“核高基”重大专项等科研课题。曾荣获江苏省十大杰出青年、江苏省首届高层次创新创业引进人才、中科院第九届杰出青年、国务院特殊津贴专家和中国青年五四奖章等荣誉。

主办单位：计算机科学与技术学院

欢迎全校师生参加！



有C程序如下：

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

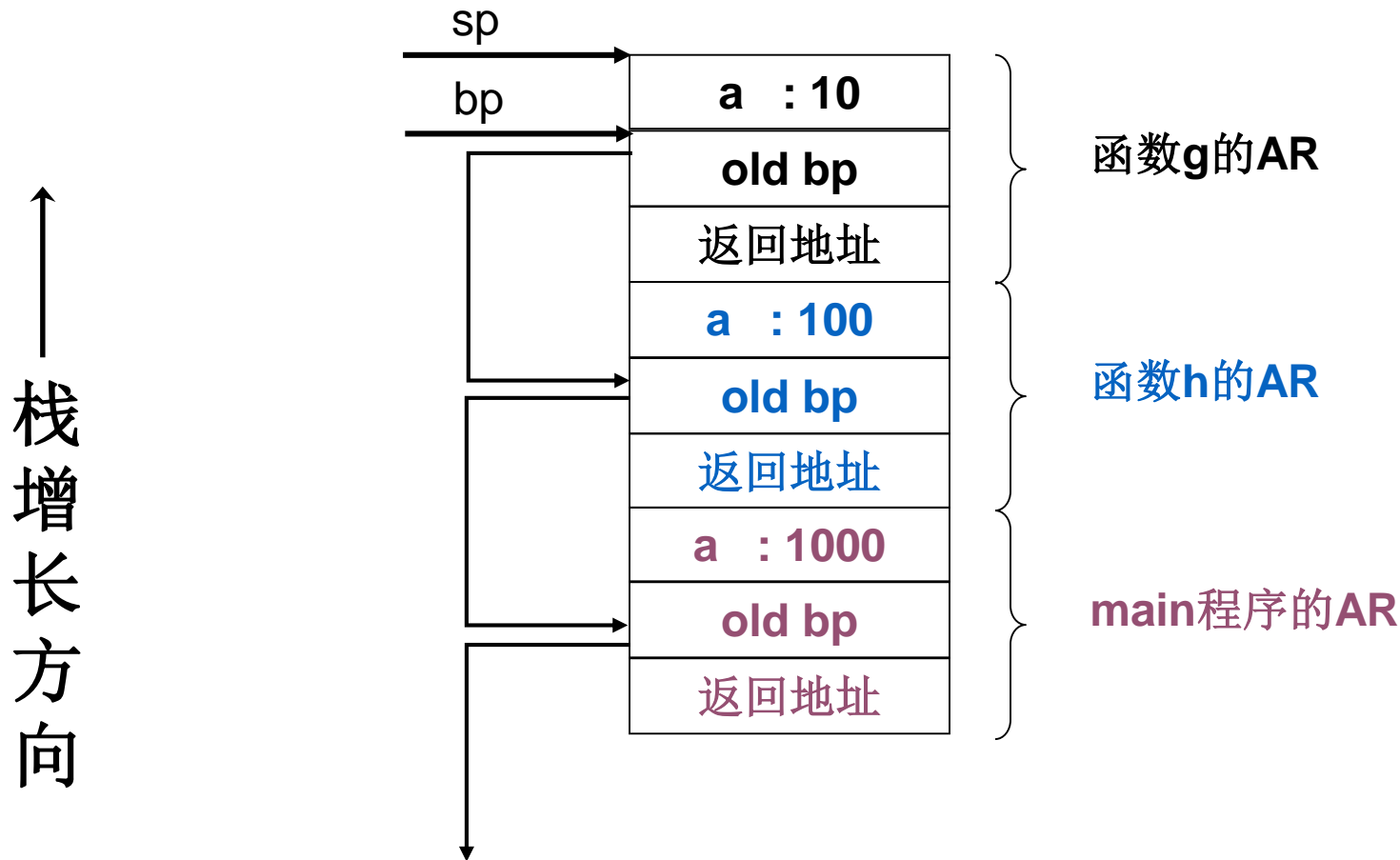
```
void h() { int a ; a = 100; g(); }
```

```
main()
```

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

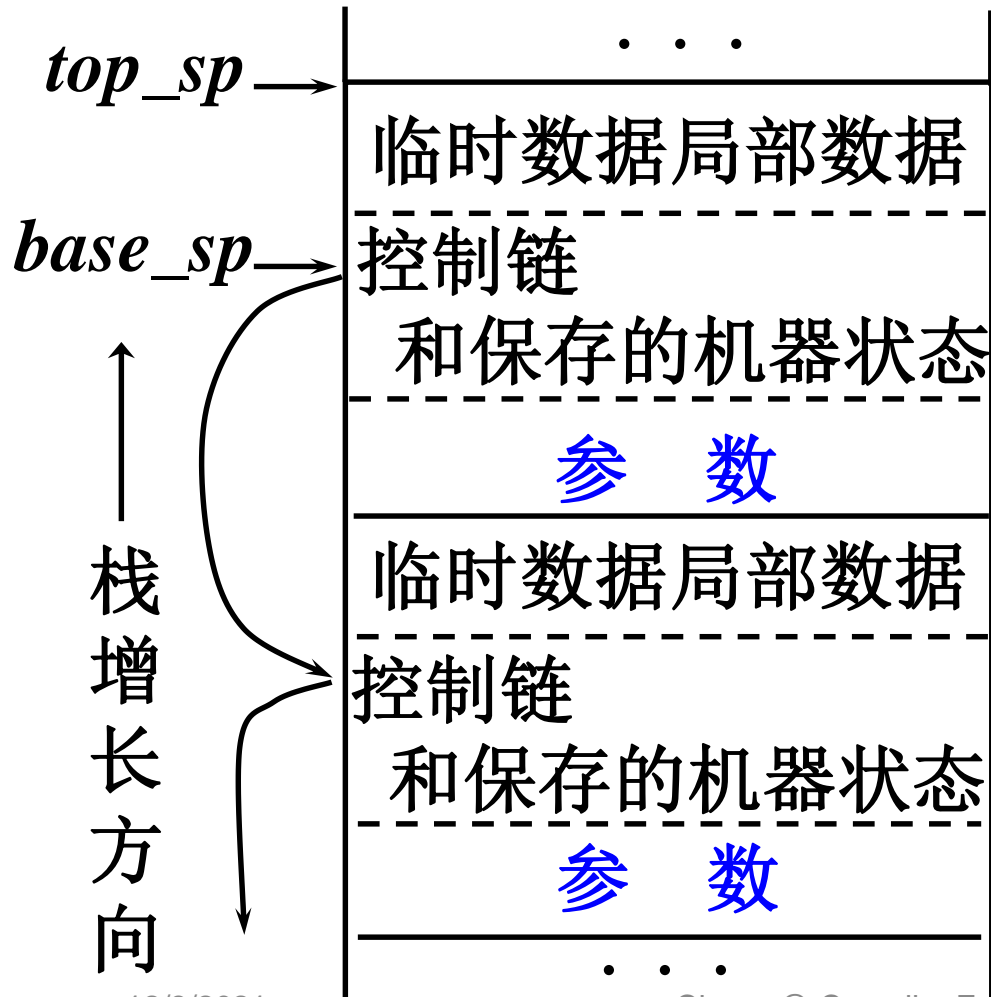


□过程g被调用时，活动记录栈的（大致）内容





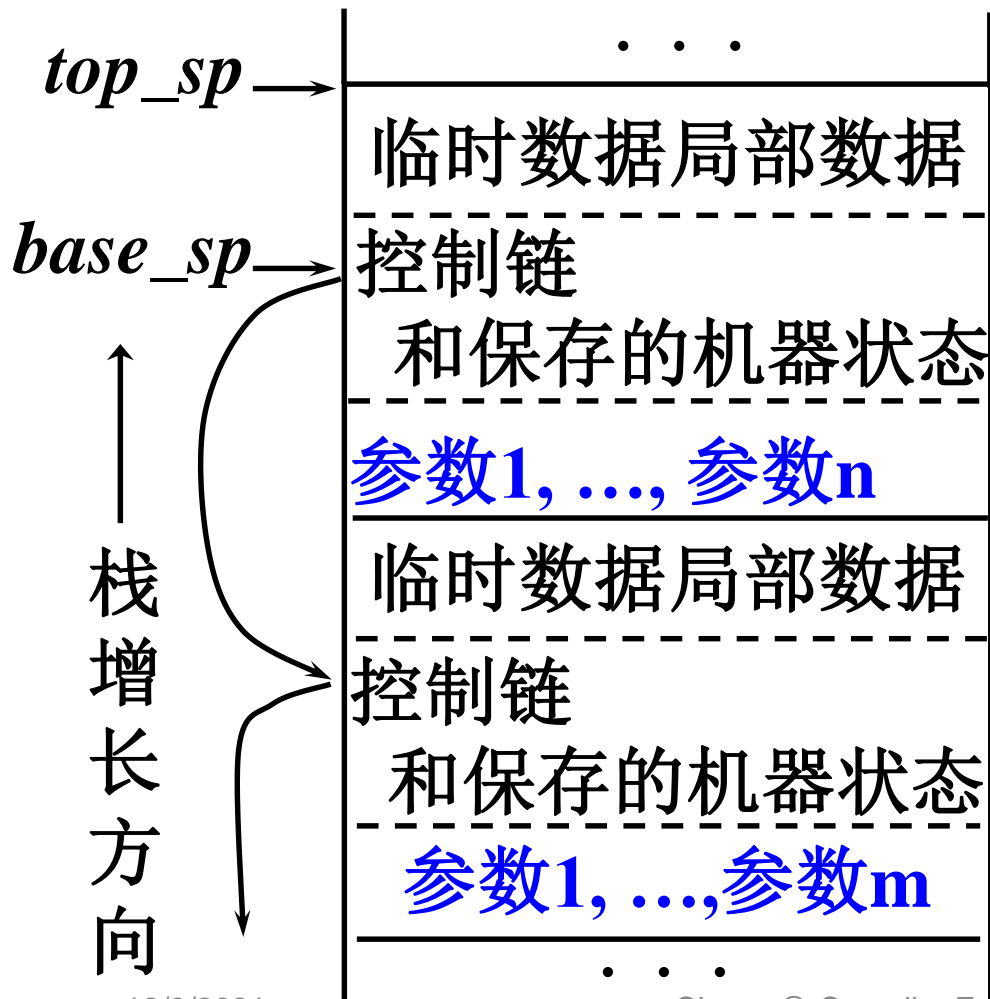
□过程的参数个数可变的情况



(1) 函数返回值改成
用寄存器传递



□过程的参数个数可变的情况

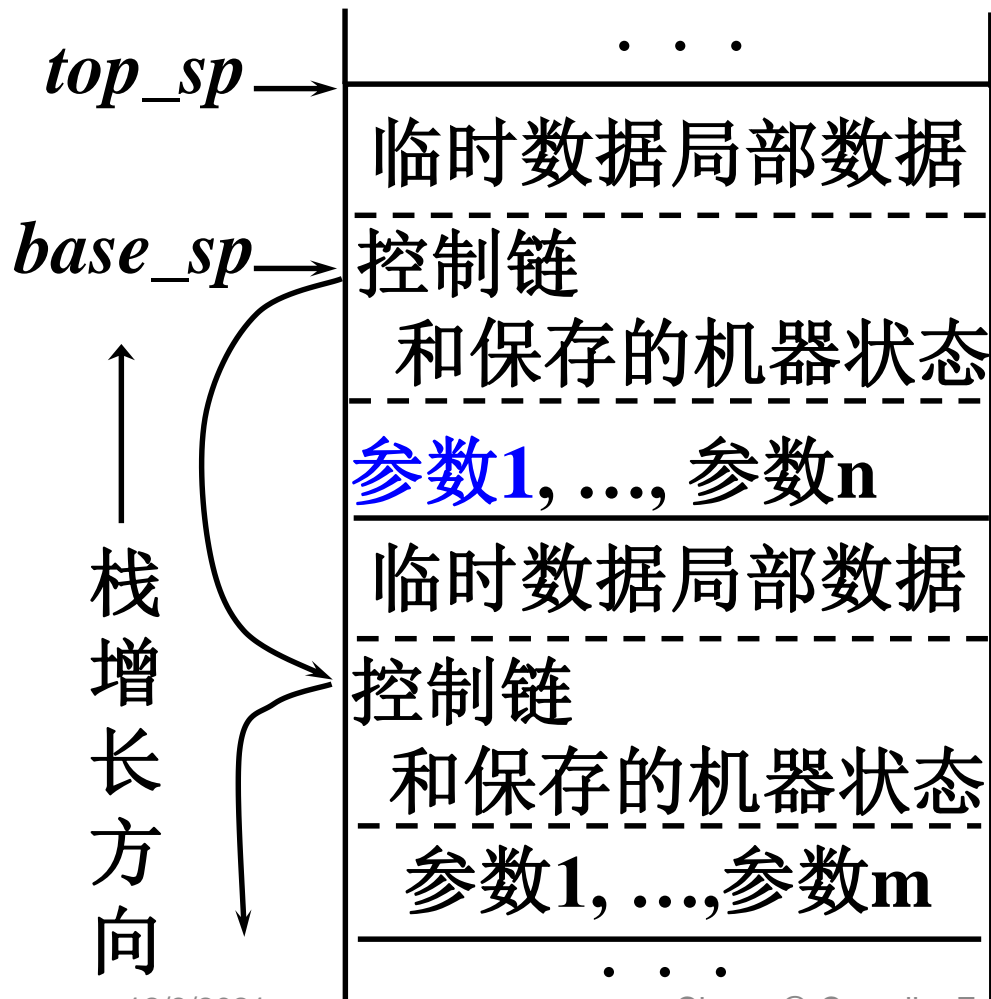


(2) 编译器产生将实参表达式逆序计算并将结果进栈的代码

自上而下依次是参数1, ..., 参数n



□过程的参数个数可变的情况

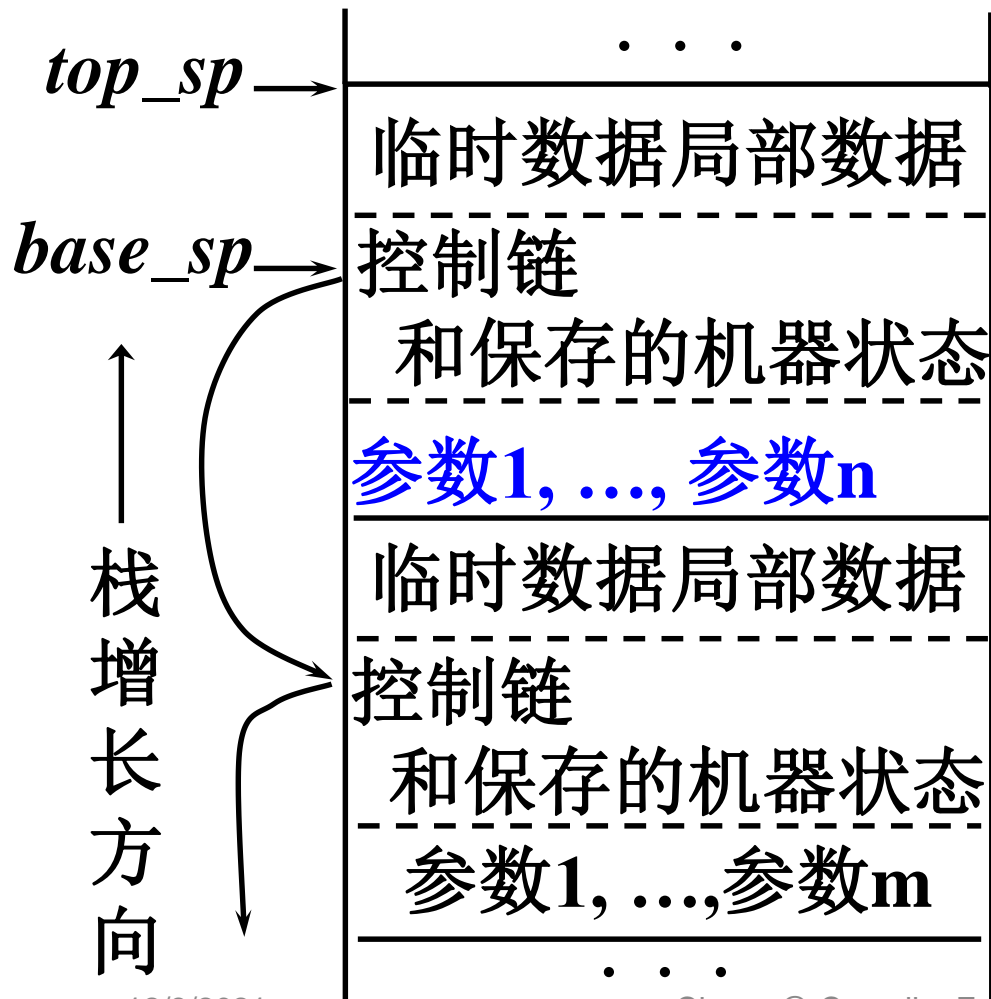


(3) 被调用函数能准确地知道第一个参数的位置

But why?



□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



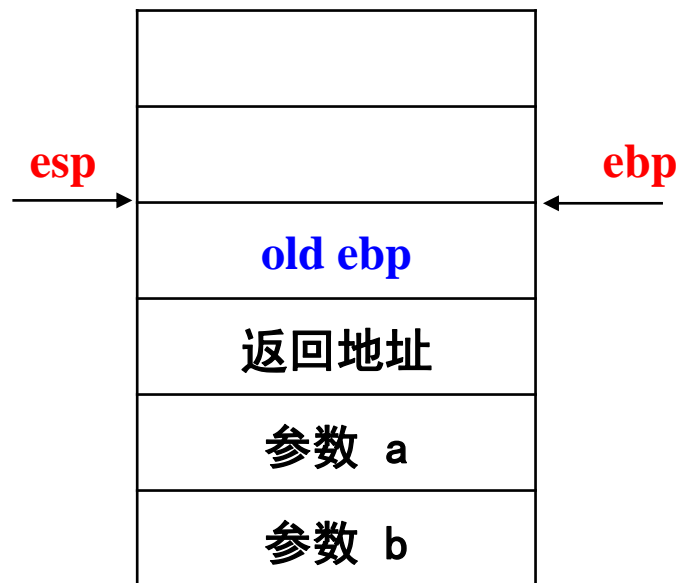
```
void func( int a , int b )  
{  
    int c , d;  
    c = a;  
    d = b;  
}
```



```
.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
```



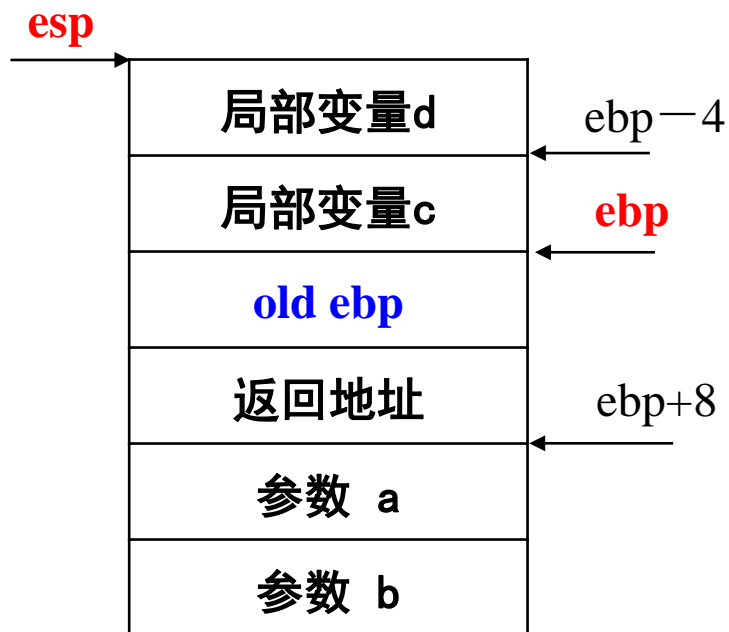
```
void func( int a , int b )  
{  
    int c , d;  
    c = a;  
    d = b;  
}
```



```
.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
```



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



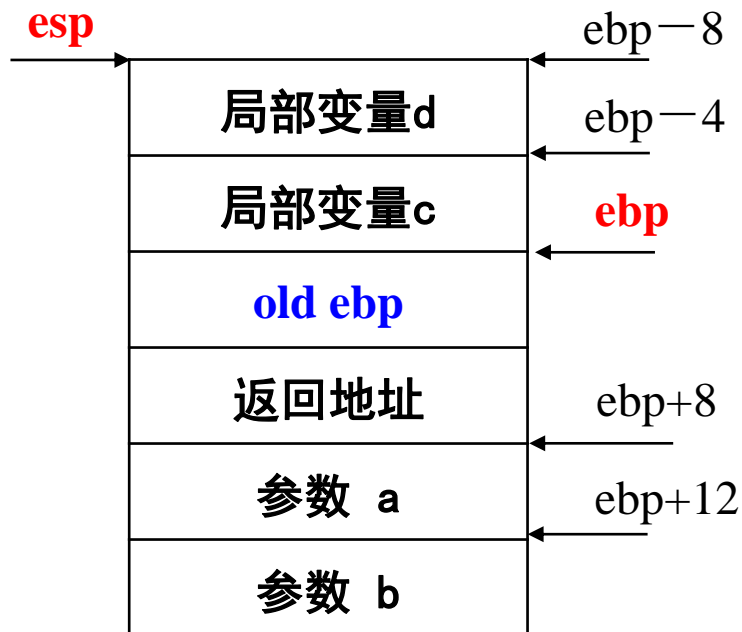
```
.file "ar.c"  
.text  
.globl func  
.type func,@function  
func:  
    pushl %ebp //老基地址压栈  
    movl %esp, %ebp //基地址指针=栈顶指针  
    subl $8, %esp //分配c,d局部变量空间  
    movl 8(%ebp), %eax //将a值放进寄存器  
    movl %eax, -4(%ebp) //将a值赋给c  
    movl 12(%ebp), %eax  
    movl %eax, -8(%ebp)  
    leave 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

有如下C程序：

```
main()
```

```
{
```

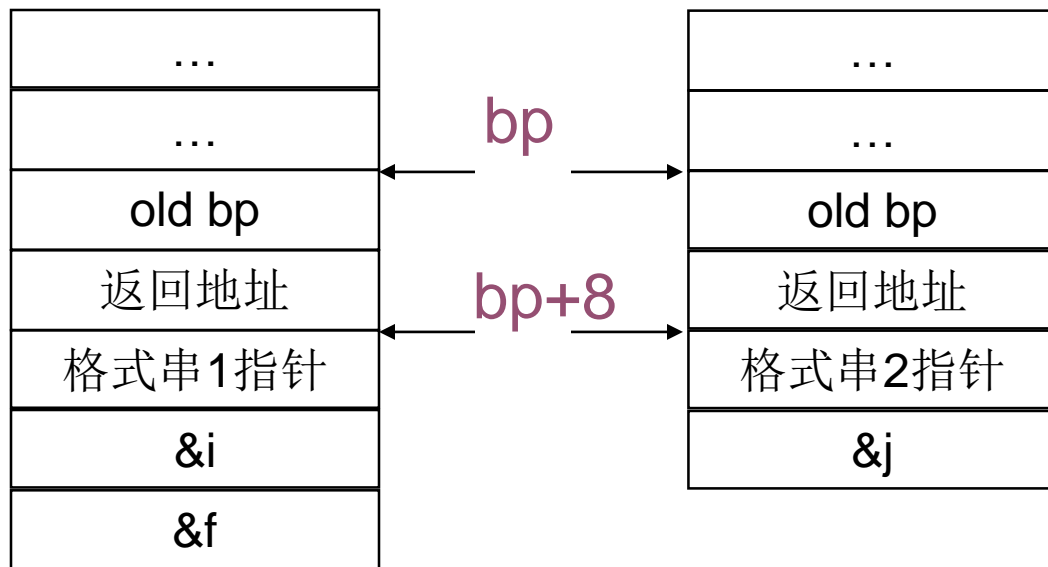
```
    int i ; float f; int j ;
```

```
    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。



□ 栈上可变长数据

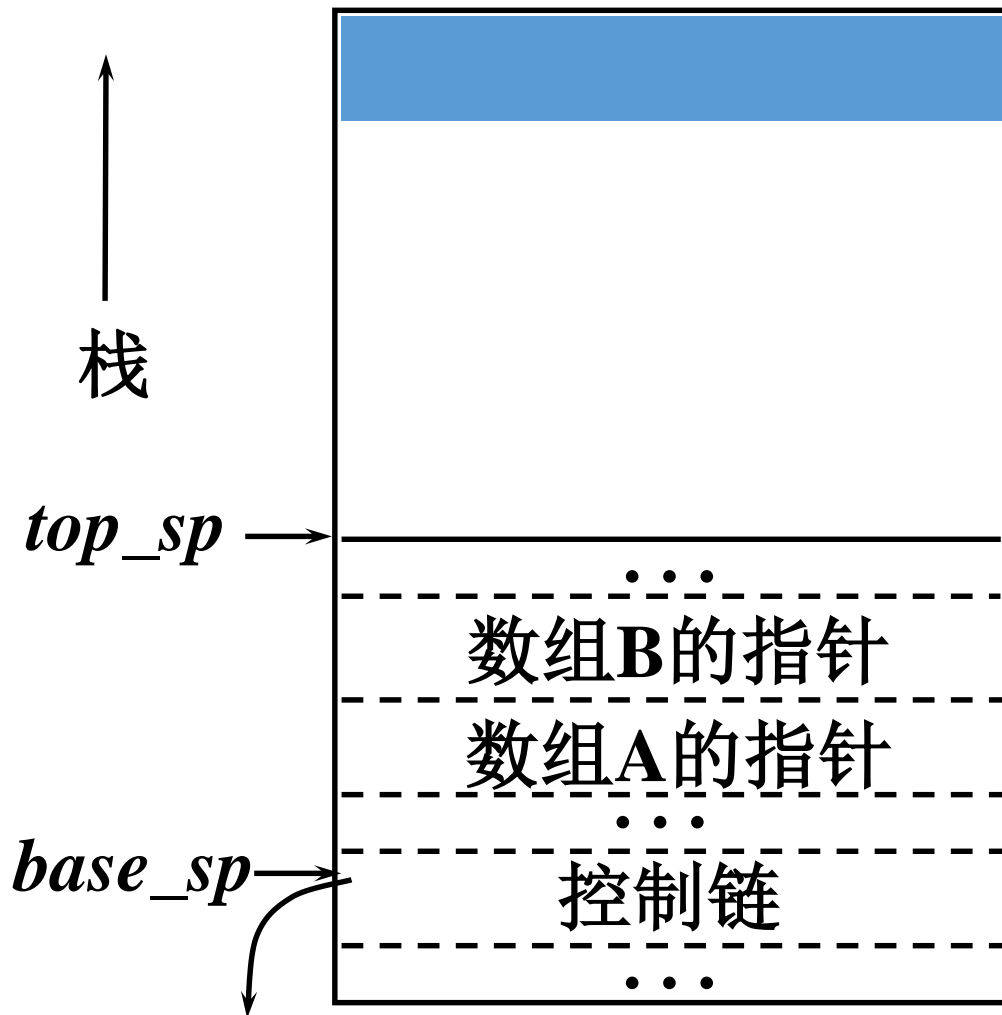
- ❖ 数据对象的长度在编译时不能确定的情况
- ❖ 但仅仅为该活动运行过程使用
- ❖ 避免垃圾回收开销
- ❖ 例：局部数组的大小要等到过程激活时才能确定

□ 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



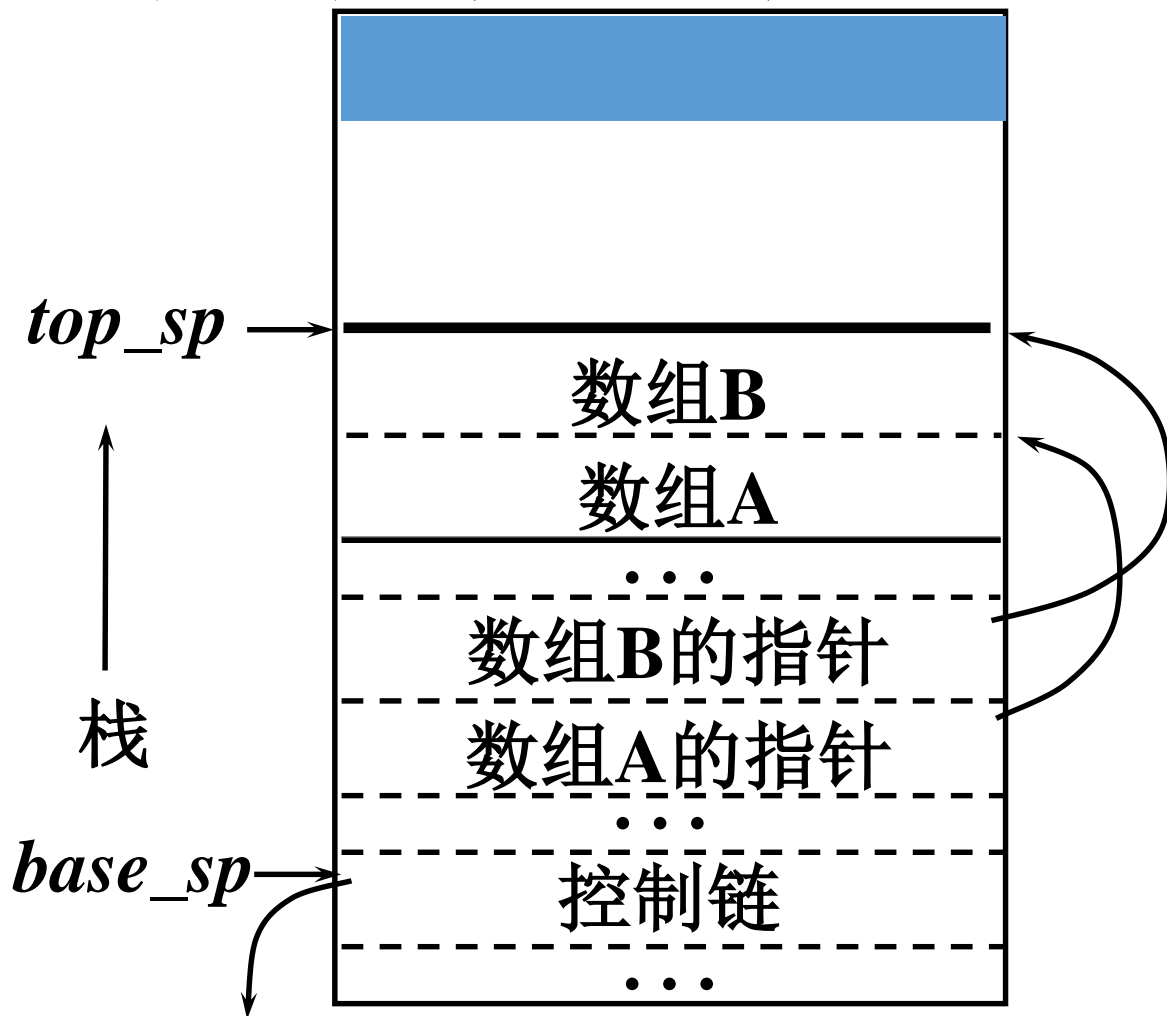
□访问动态分配的数组



(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



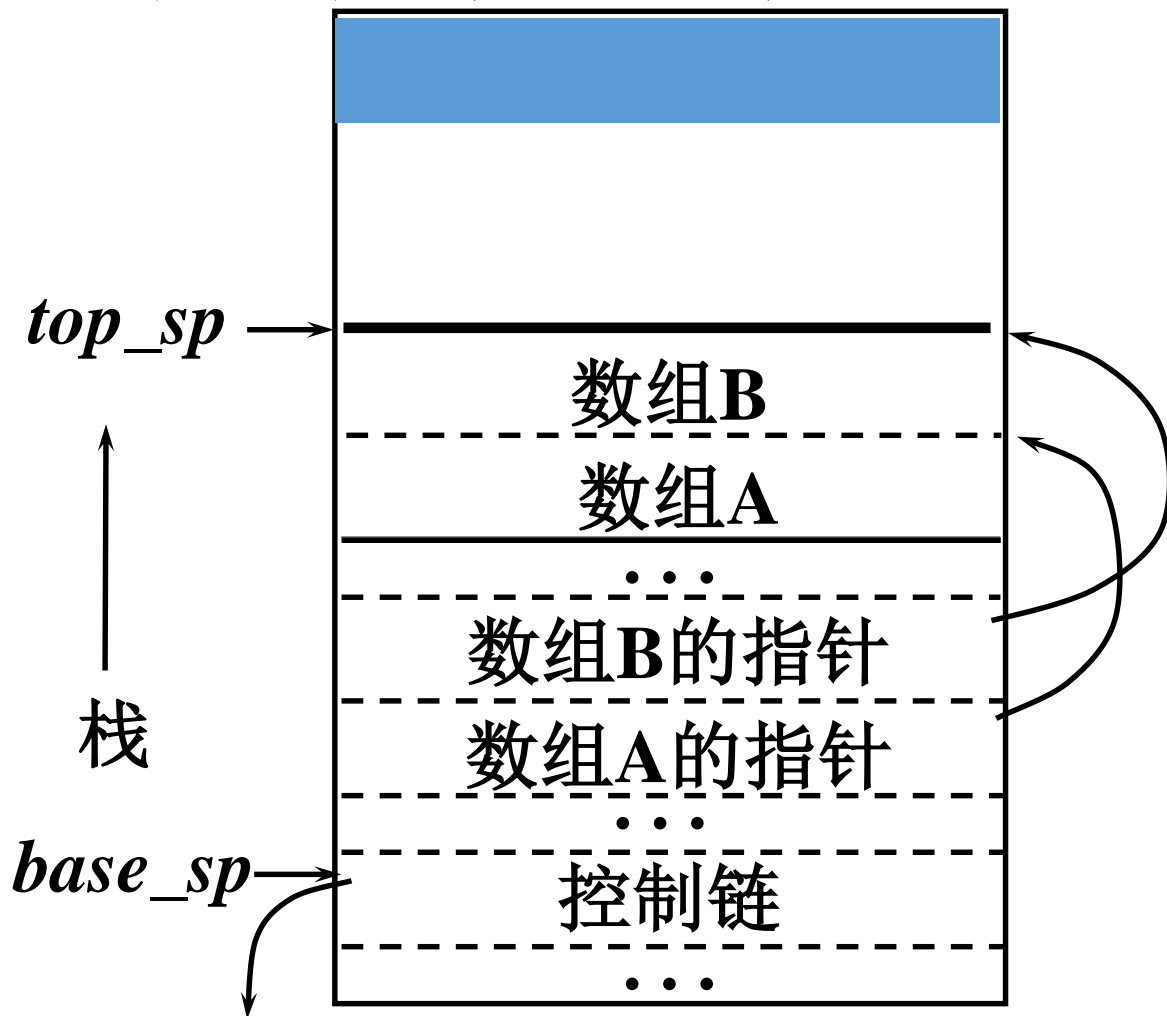
□访问动态分配的数组



(2) 运行时，这些指针指向分配在栈顶的数组存储空间



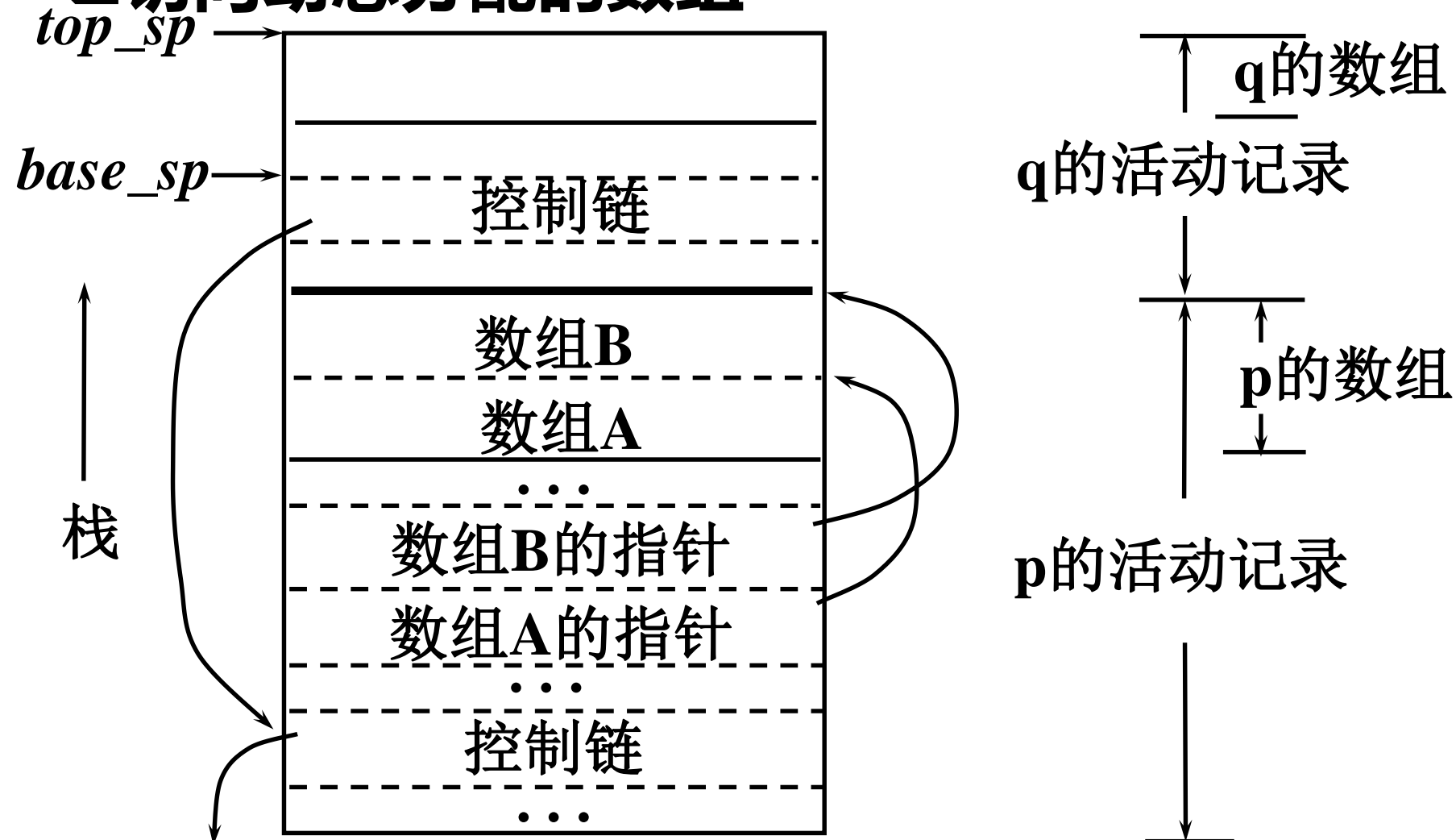
□访问动态分配的数组



(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



□访问动态分配的数组





□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;
}	 	}



- 无过程嵌套 (C语言)
- 有过程嵌套 (Pascal语言)



□无过程嵌套时的数据访问

- ❖ 过程体中的非局部引用可以直接使用静态确定的地址（非局部数据此时就是全局数据）
- ❖ 局部变量在栈顶的活动记录中，可以通过`base_sp`指针来访问
- ❖ 无须深入栈中取数据，无须访问链
 - 因此，访问链是optional的



□ 有过程嵌套的静态作用域

- ❖ 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
- ❖ 然而，当B的代码激活执行时，如何找到A的活动记录呢？
 - 这里需要建立访问链

```
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：

A的活动记录
C的活动记录
B的活动记录

当A直接调用B时：

A的活动记录
B的活动记录



□ 有过程嵌套的静态作用域

sort

readarray

exchange

quicksort

partition

该例子参考紫书P186页



□ 有过程嵌套的静态作用域

❖ 过程嵌套深度

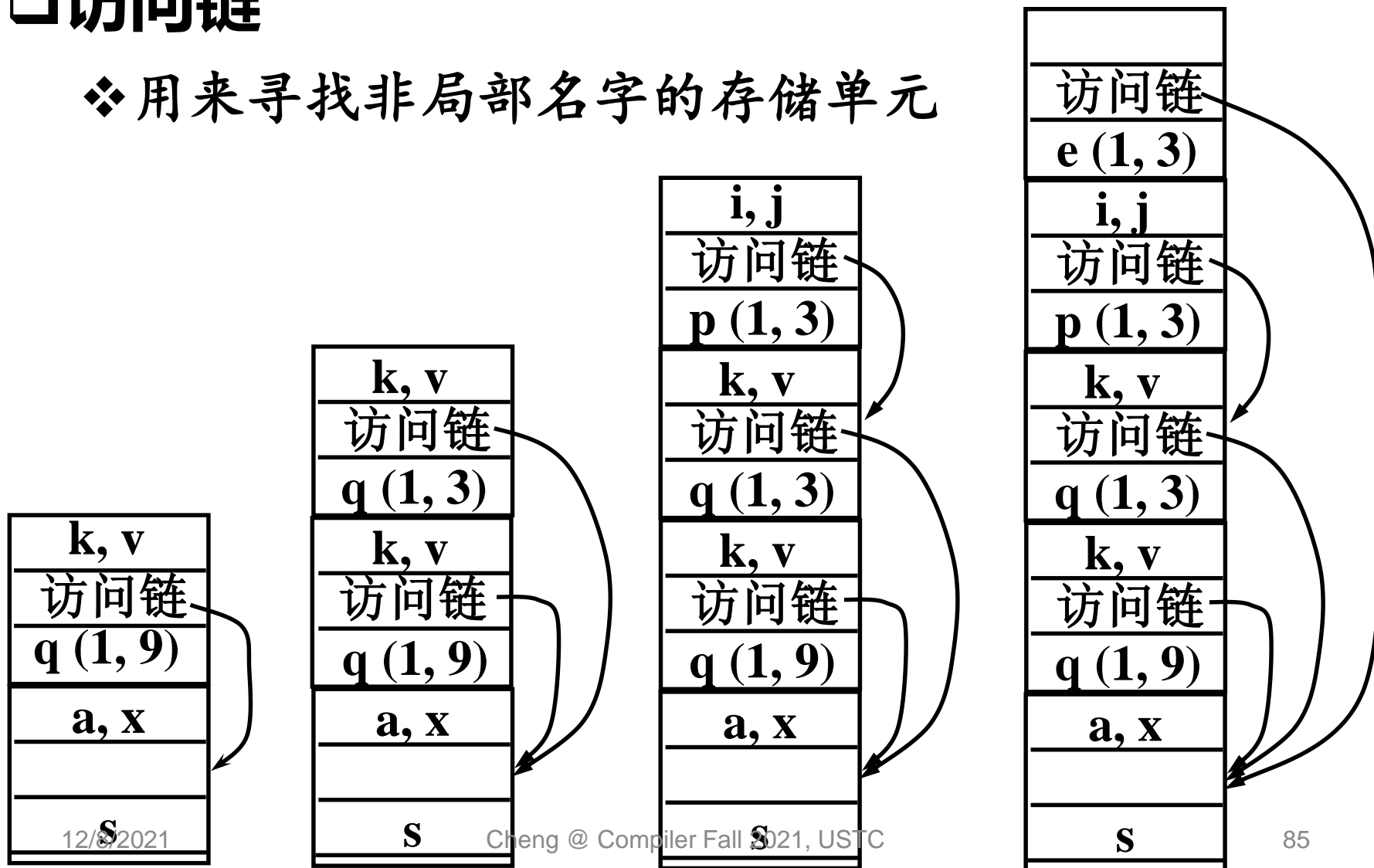
❖ 变量的嵌套深度：它的声明所在过程的嵌套深度作为该名字的嵌套深度

sort	1
readarray	2
exchange	2
quicksort	2
partition	3



访问链

❖ 用来寻找非局部名字的存储单元





□两个关键问题需要解决：

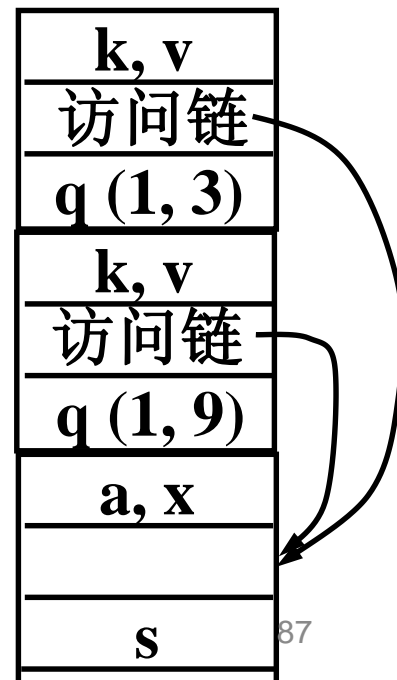
- ❖通过访问链访问非局部引用
- ❖访问链的建立



□访问非局部名字的存储单元

❖假定过程 p 的嵌套深度为 n_p ，它引用嵌套深度为 n_a 的变量 a ， $n_a \leq n_p$ ，如何访问 a 的存储单元

sort	1
readarray	2
exchange	2
quicksort	2
partition	3



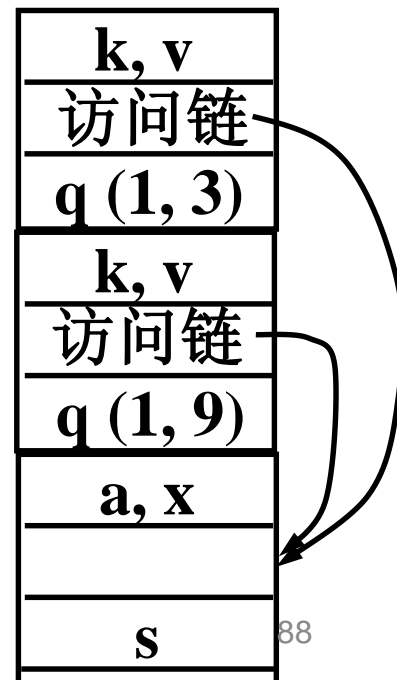


□访问非局部名字的存储单元

❖假定过程 p 的嵌套深度为 n_p ，它引用嵌套深度为 n_a 的变量 a ， $n_a \leq n_p$ ，如何访问 a 的存储单元

- 从栈顶的活动记录开始，追踪访问链 $n_p - n_a$ 次
- 到达 a 的声明所在过程的活动记录
- 访问链的追踪用间接操作就可完成

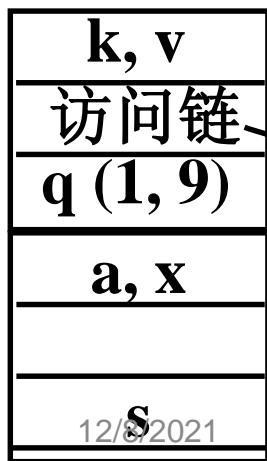
sort	1
readarray	2
exchange	2
quicksort	2
partition	3





访问非局部名字的存储单元

sort
readarray
exchange
quicksort
partition





□建立访问链(过程调用序列代码的一部分)

❖假定嵌套深度为 n_p 的过程 p 调用嵌套深度为 n_x 的过程 x

(1) $n_p < n_x$ 的情况

sort	1
readarray	2
exchange	2
quicksort	2
partition	3

这时 x 肯定就
声明在 p 中



□建立访问链

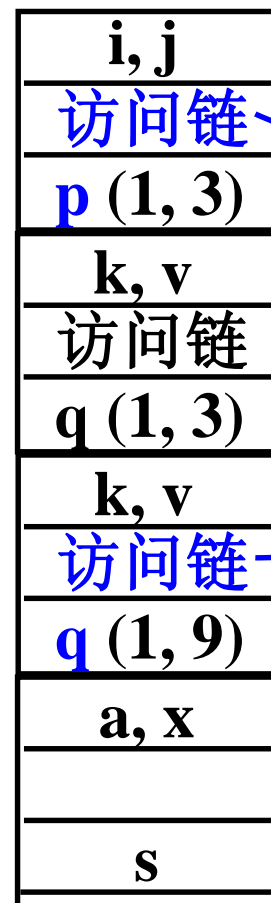
❖假定嵌套深度为 n_p 的过程 p 调用嵌套深度为 n_x 的过程 x

(1) $n_p < n_x$ 的情况

❖这时 x 肯定就声明在 p 中(嵌套)

❖被调用过程的访问链必须指向调用过程的活动记录(次栈顶)的访问链

❖sort调用quicksort、quicksort调用partition





□建立访问链

❖假定嵌套深度为 n_p 的过程p调用嵌套深度为 n_x 的过程x

(2) $n_p \geq n_x$ 的情况

sort	1
readarray	2
exchange	2
quicksort	2
partition	3

这时p和x的
嵌套深度分别
为1, 2, ...,
 $n_x - 1$ 的外围过
程肯定相同



□建立访问链

❖ 假定嵌套深度为 n_p 的过程 p 调用嵌套深度为 n_x 的过程 x

(2) $n_p \geq n_x$ 的情况

❖ 沿着访问链追踪 $n_p - n_x + 1$ 次，到达了静态包围 x 和 p 的且离它们最近的那个过程的最新活动记录

❖ 所到达的活动记录就是 x 的活动记录中的访问链应该指向的那个活动记录

❖ partition 调用 exchange, quicksort 调用自身





□ 实参与形参

- 存储单元（左值）
- 存储内容（右值）

根据所传递的实参的“内容”，参数传递可分为：

- 传值调用：传递实参的右值到形参单元；
- 引用调用：传递实参的左值到形参单元。



```
procedure swap( a , b )
```

```
  a, b : int; temp : int;
```

```
begin
```

```
  temp := a ;
```

```
  a := b;
```

```
  b := temp;
```

```
end.
```

讨论下面程序在不同参数
传递方式下输出：

```
x := 10 ; y := 20;
```

```
  swap( x,y );
```

```
  print ( x, y );
```



讨论下面程序在不同参数传递方式下输出：

1) `x := 10 ; y := 20;`

`swap(x,y);`

`print (x, y);`

5000: x → 10

4000: y → 20

2004: a →

2000: b →

1990: temp →

栈

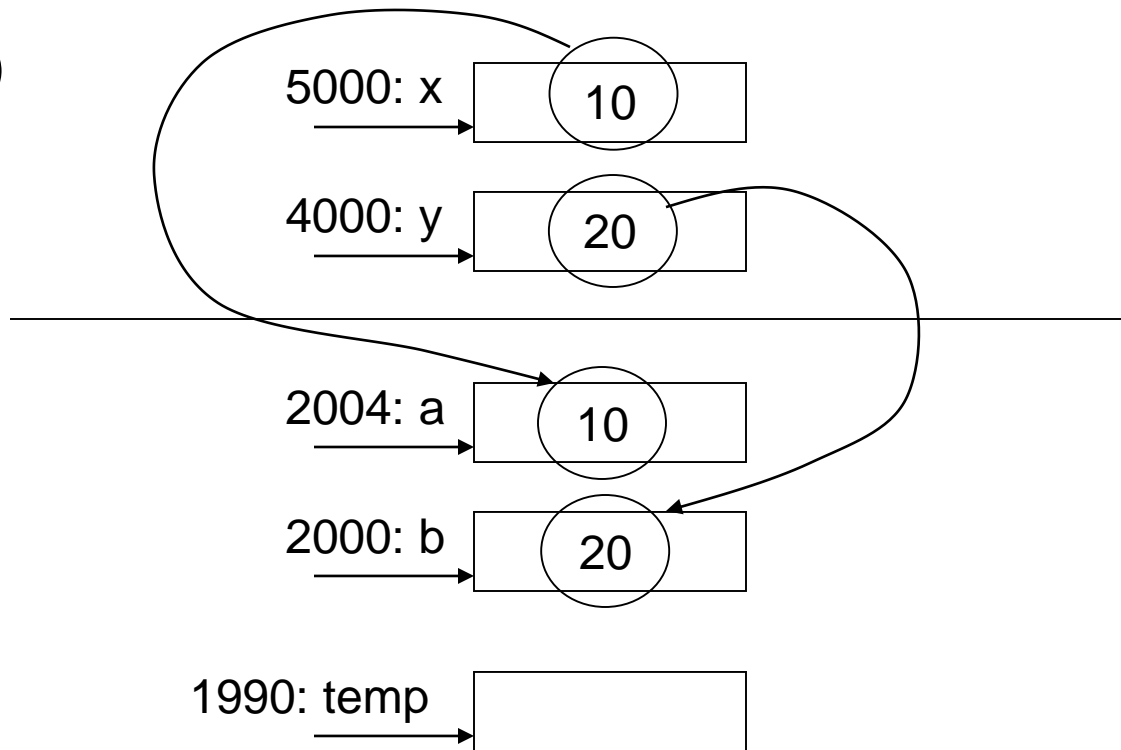


实参`x,y`和过程`swap`中形参`a,b`,和局部数据`temp`的存储分布示意



过程调用—swap(x,y)

- 传参一形、实结合





过程调用 — swap(x,y)

- 过程执行

temp := a

5000: x →

10

4000: y →

20

2004: a →

10

2000: b →

20

1990: temp →

10



过程调用 — swap(x,y)

- 过程执行

a := b

5000: x →

10

4000: y →

20

2004: a →

20

2000: b →

20

1990: temp →

10



过程调用 — swap(x,y)

- 过程执行

b := temp

5000: x →

10

4000: y →

20

2004: a →

20

2000: b →

10

1990: temp →

10



过程swap(x,y)执行后

- print(x, y)

5000: x →

10

4000: y →

20

10, 20

2004: →

20

2000: →

10

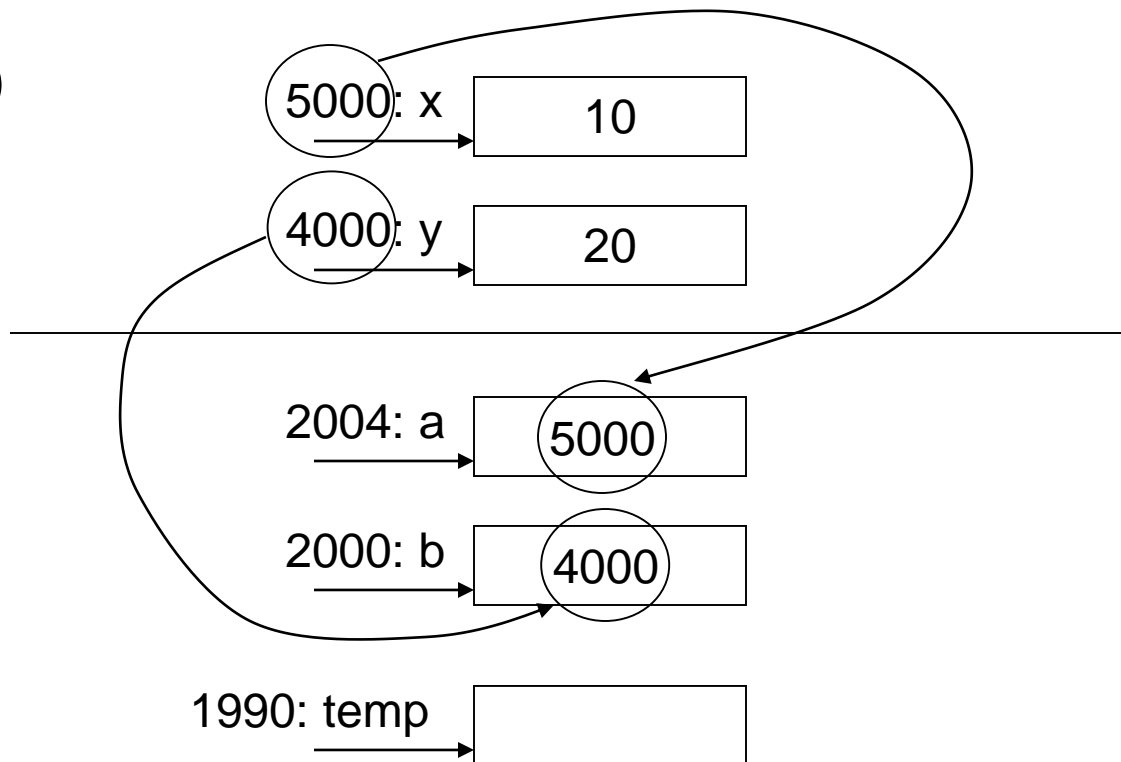
1990: →

10



过程调用—swap(x,y)

- 传参一形、实结合

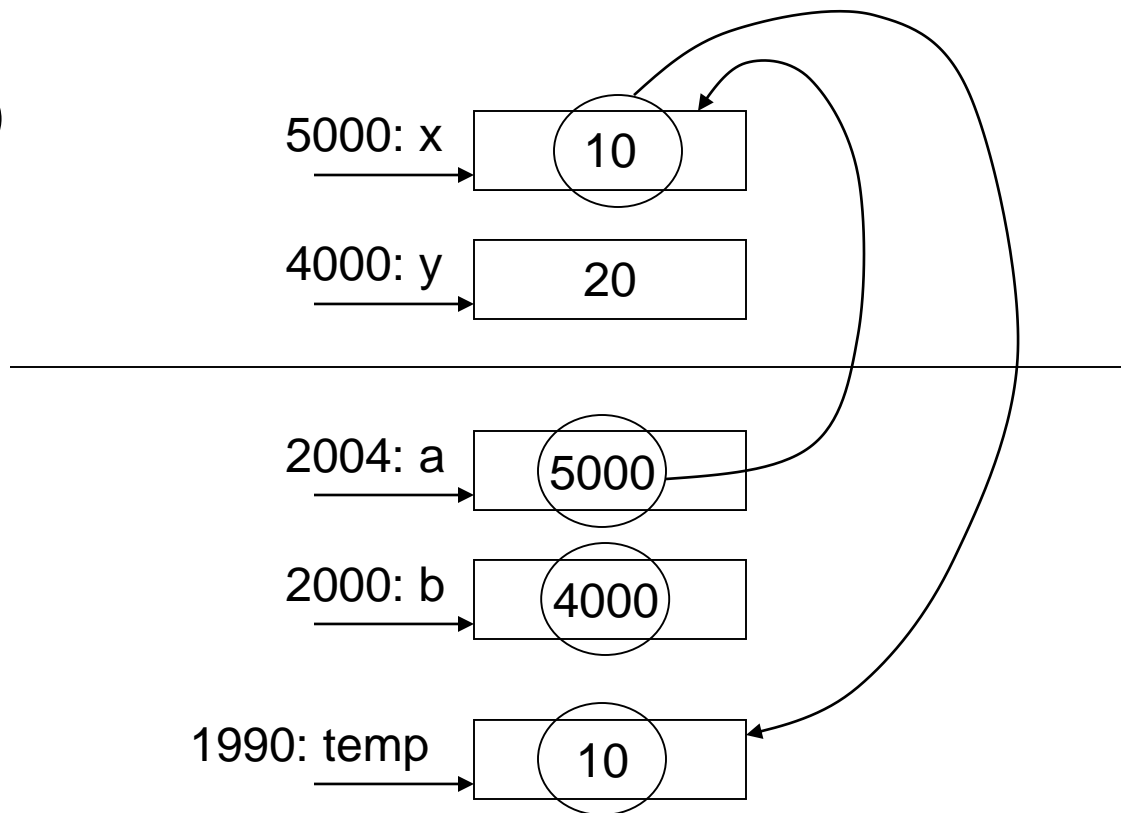




过程调用 — swap(x,y)

- 过程执行

temp := a

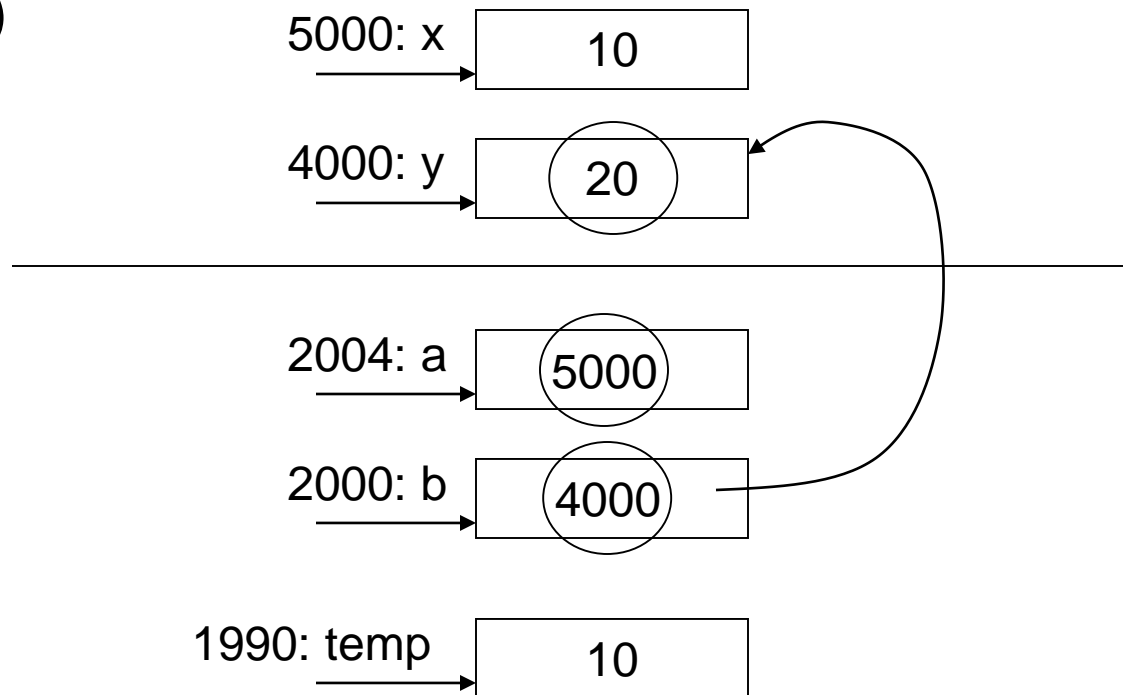




过程调用—swap(x,y)

- 过程执行

$a := b$

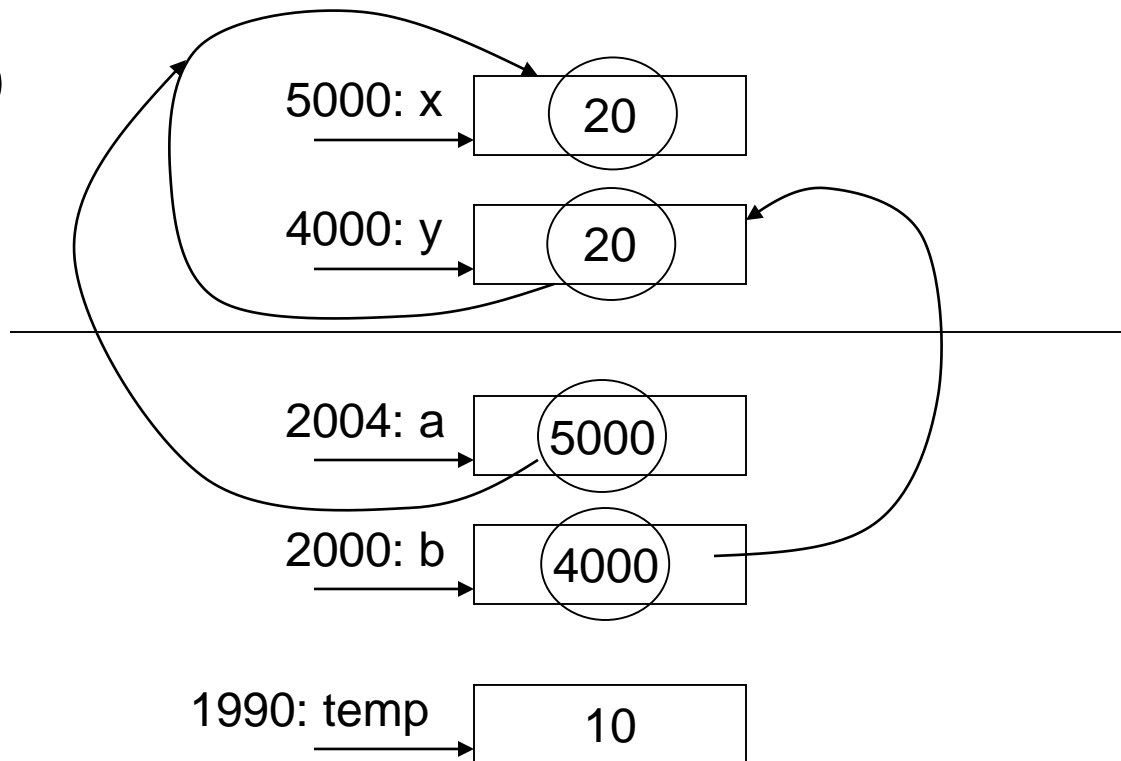




过程调用—swap(x,y)

- 过程执行

$a := b$

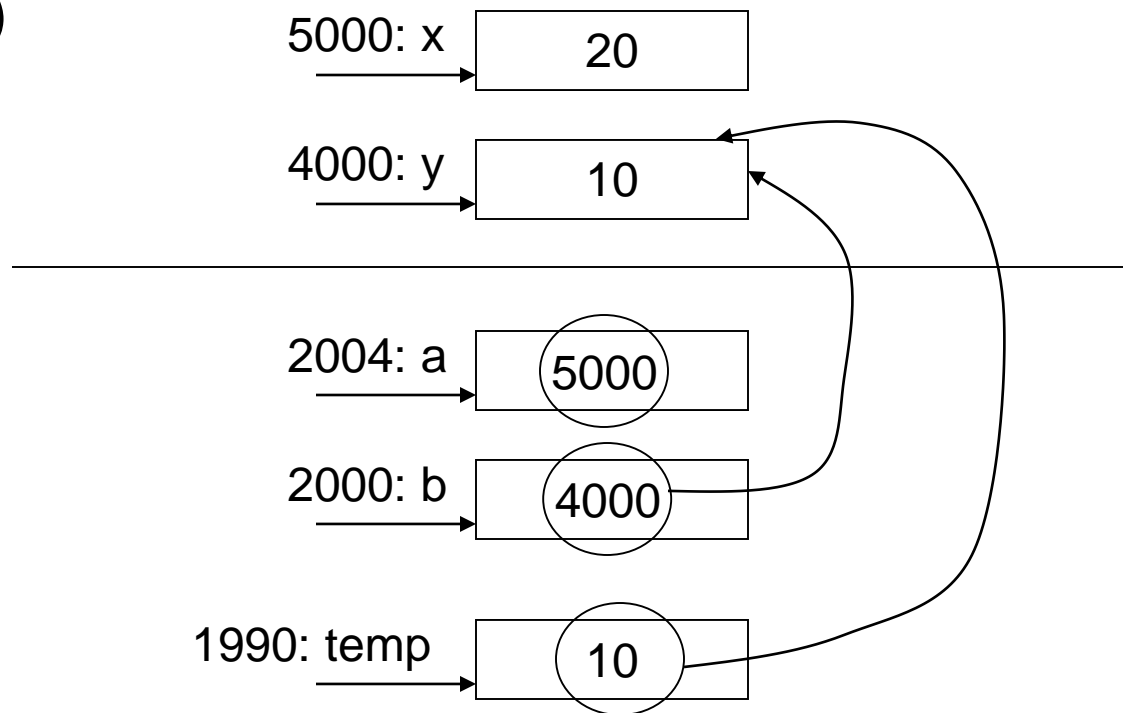




过程调用 — swap(x,y)

- 过程执行

$b := temp$





过程swap(x,y)执行后

- print(x, y)

5000: x →

4000: y →

20, 10

2004: →

2000: →

1990: →



以下c程序的输出是什么?

```
void func(char *s){s = (char*)malloc(10);}  
  
int main()  
{  
    char *p = NULL;  
    func(p);  
    if(!p)printf("error\n");else printf("ok\n");  
    return 0;  
}
```



```
void swap1(int p,int q)
```

```
{
```

```
    int temp;
```

```
    temp = p;
```

```
    p = q;
```

```
    q = temp;
```

```
}
```

```
    pushl %ebp
```

```
    movl  %esp, %ebp
```

```
    subl  $4, %esp
```

```
    movl  8(%ebp), %eax
```

```
    movl  %eax, -4(%ebp)
```

```
    movl  12(%ebp), %eax
```

```
    movl  %eax, 8(%ebp)
```

```
    movl  -4(%ebp), %eax
```

```
    movl  %eax, 12(%ebp)
```

```
    leave
```

```
    ret
```



```
void swap2(int *p,int *q)
```

```
{  
    int    temp;  
    temp = *p;  
    *p     = *q;  
    *q     = temp;  
}  
  
    pushl %ebp  
    movl  %esp, ebp  
    subl  $4, %esp  
    movl  8(%ebp), %eax  
    movl  (%eax), %eax  
    movl  %eax, -4(%ebp)  
    movl  8(%ebp), %edx  
    movl  12(%ebp), %eax  
    movl  (%eax), %eax  
    movl  %eax, (%edx)  
    movl  12(%ebp), %edx  
    movl  -4(%ebp), %eax  
    movl  %eax, (%edx)  
    leave  
    ret
```



```
void swap3(int *p, int *q)
```

```
{
```

```
    int *temp    ;
```

```
    temp = p    ;
```

```
    p     = q    ;
```

```
    q     = temp ;
```

```
}
```

```
    pushl %ebp
```

```
    movl  %esp, ebp
```

```
    subl  $4, %esp
```

```
    movl  8(%ebp), %eax
```

```
    movl  %eax, -4(%ebp)
```

```
    movl  12(%ebp), %eax
```

```
    movl  %eax, 8(%ebp)
```

```
    movl  -4(%ebp), %eax
```

```
    movl  %eax, 12(%ebp)
```

```
    leave
```

```
    ret
```

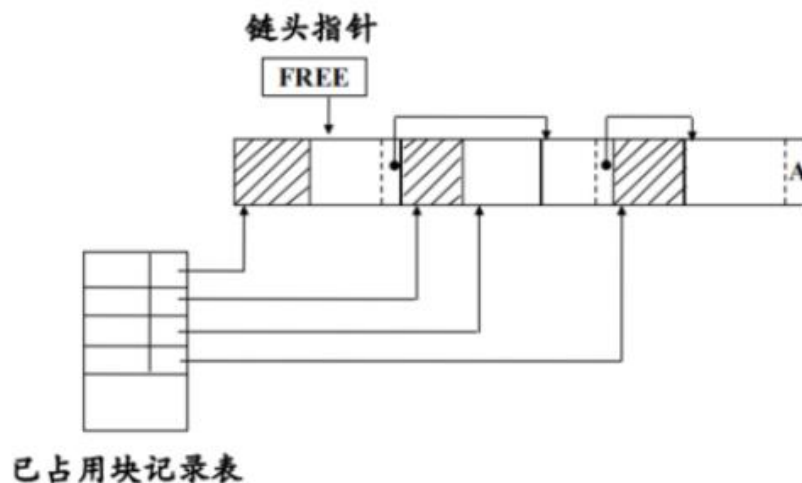
```
void swap4(int &p, int &q)
```

```
{  
    int temp;  
    temp = p;  
    p     = q;  
    q     = temp;  
}  
  
    pushl %ebp  
    movl  %esp, %ebp  
    subl  $4, %esp  
    movl  8(%ebp), %eax  
    movl  (%eax), %eax  
    movl  %eax, -4(%ebp)  
    movl  8(%ebp), %edx  
    movl  12(%ebp), %eax  
    movl  (%eax), %eax  
    movl  %eax, (%edx)  
    movl  12(%ebp), %edx  
    movl  -4(%ebp), %eax  
    movl  %eax, (%edx)  
    leave  
    ret
```




□堆空间

- ❖ 用于存放生命周期不确定、或生存到被明确删除为止的数据对象
- ❖ e.g., 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





□分配/回收堆空间的子系统

❖分配：为内存请求分配一段连续、适当大小的堆空间

- 首先从空闲的堆空间选择
- 如内存紧张，可以回收一部分内存
 - C、C++需要手动回收空间
 - Java可以自动回收空间

❖评价指标

- 空间效率
- 程序效率
- 管理效率



中国科学技术大学
University of Science and Technology of China



《编译原理与技术》

运行时存储空间的组织和管理

The end~