第5章 运行时存储空间的组织和管理

术语

- 过程的活动

  过程的一次执行称为过程的一次活动

- 活动记录

  过程的活动需要可执行代码和存放所需信息的存储空间，所需的局部信息的存储空间称为活动记录

本章内容

- 讨论一个活动记录中的数据布局
- 程序执行过程中，所有活动记录的组织方式

第5章 运行时存储空间的组织和管理

- 影响存储分配策略的主要语言特征
  - 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动态地分配
  - 存储块是否必须显式地回收

5.1 局部存储分配

5.1.1 过程

语言概念：

过程定义、过程调用、形式参数、实在参数、活动的生存期

5.1 局部存储分配

5.1.2 名字的作用域和绑定

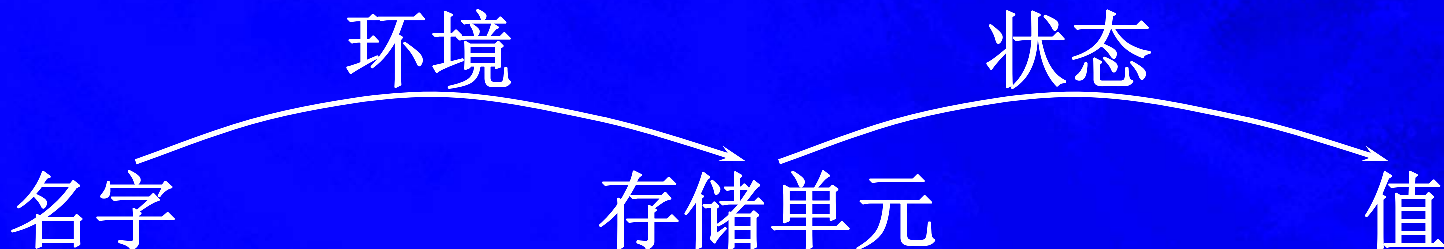
1、名字的作用域

- 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

5.1 局部存储分配

2、环境和状态

- 环境把名字映射到左值（存储单元），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



5.1 局部存储分配

3、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

5.1 局部存储分配

3、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

5.1 局部存储分配

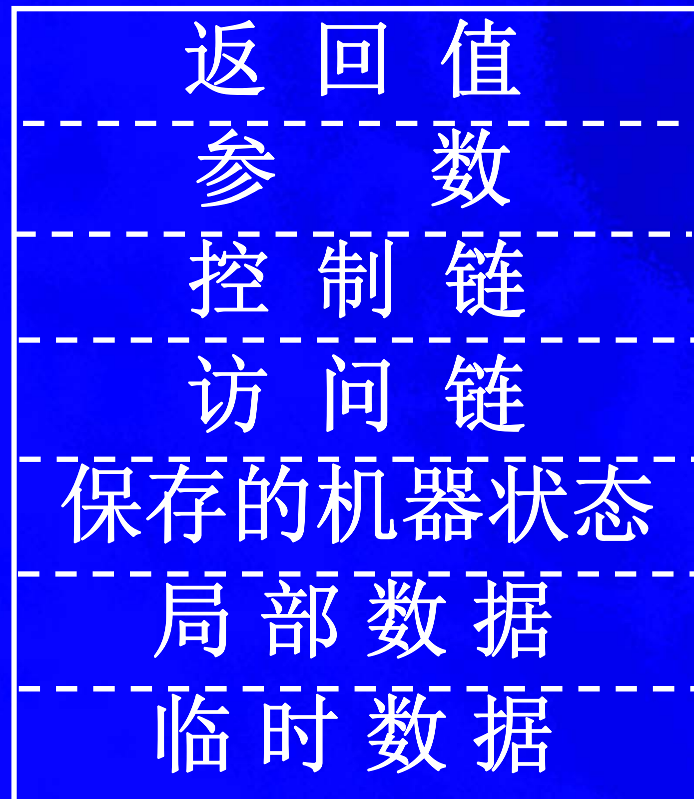
3、静态概念和动态概念的对应

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

5.1 局部存储分配

5.1.3 活动记录

活动记录的常见布局



5.1 局部存储分配

5.1.4 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它们声明时出现的次序，在活动记录的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的某个位置的地址来表示
- 数据的存储布局还有一个对齐问题

5.1 局部存储分配

- 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

5.1 局部存储分配

- 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

5.1 局部存储分配

- 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

5.1 局部存储分配

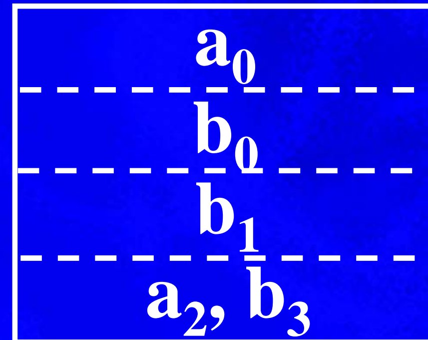
5.1.5 程序块

- 本身含有局部变量声明的语句
- 可以嵌套
- 作用域按最接近的嵌套规则给出
- 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

5.1 局部存储分配

```
main( ) / * 例 */
{ /* begin of B0 */
    int a = 0;
    int b = 0;
    { /* begin of B1 */
        int b = 1;
        {/* begin of B2 */
            int a = 2;
        }/* end of B2 */
        {/* begin of B3 */
            int b = 3;
        }/* end of B3 */
    }/* end of B1 */
}/* end of B0 */
```

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



重叠分配存储单元

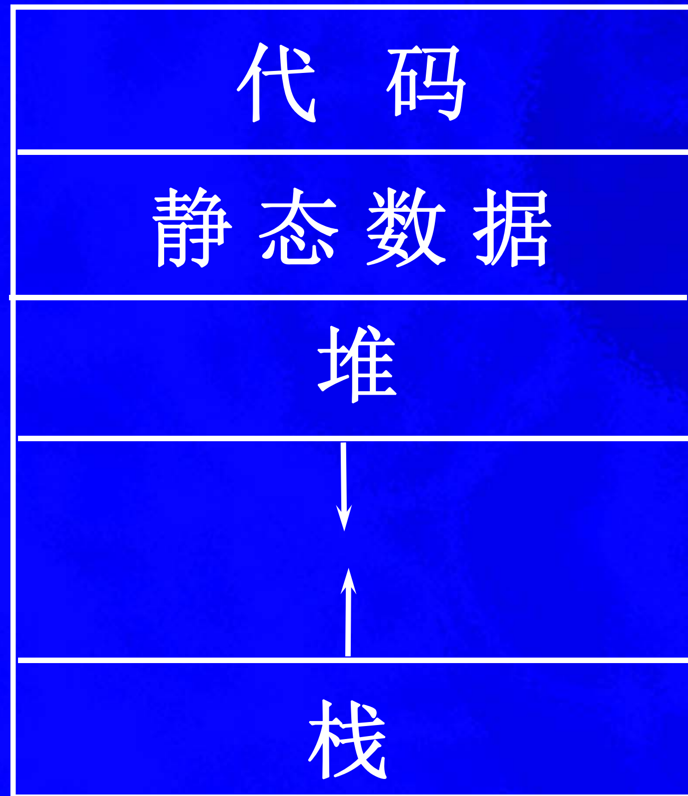
5.2 全局栈式存储分配

本节介绍

- 介绍程序运行时所需的各个活动记录在存储空间的组织
- 描述过程的目标代码怎样访问绑定到局部名字的存储单元
- 介绍三种分配策略
  - 静态分配策略
  - 栈式分配策略
  - 堆式分配策略

5.2 全局栈式存储分配

5.2.1 运行时内存的划分



5.2 全局栈式存储分配

1、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

5.2 全局栈式存储分配

2、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

5.2 全局栈式存储分配

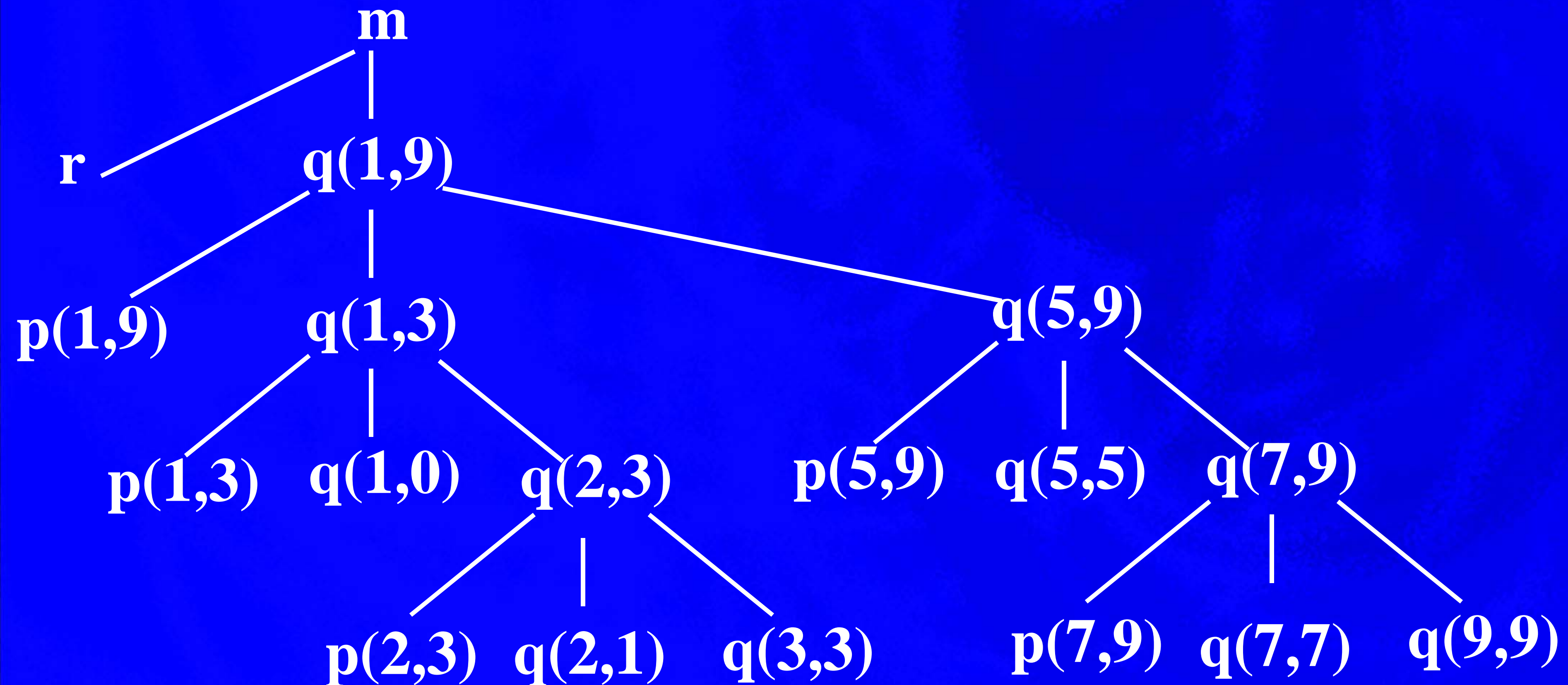
- 例 C程序的外部变量、静态局部变量以及程序中出现的常量都可以静态分配
- 声明在函数外面
  - 外部变量 —— 静态分配
  - 静态外部变量 —— 静态分配
- 声明在函数里面
  - 静态局部变量 —— 静态分配
  - 自动变量 —— 不能静态分配

5.2 全局栈式存储分配

5.2.2 活动树和运行栈

1、活动树

– 用树来描绘控制进入和离开活动的次序

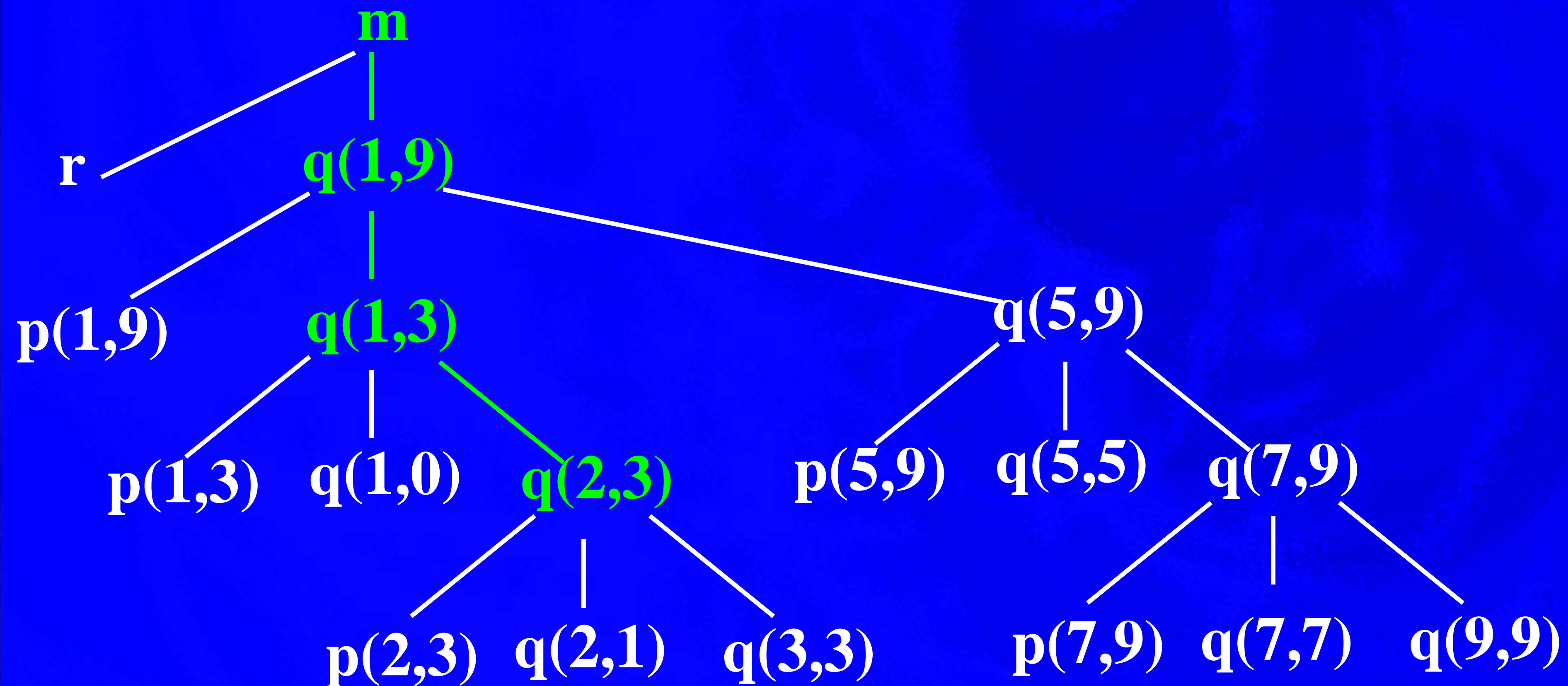


5.2 全局栈式存储分配

- 活动树的特点
  - 每个结点代表某过程的一个活动
  - 根结点代表主过程的活动
  - 结点*a*是结点*b*的父结点，当且仅当控制流从活动*a*进入活动*b*
  - 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

5.2 全局栈式存储分配

- 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)**

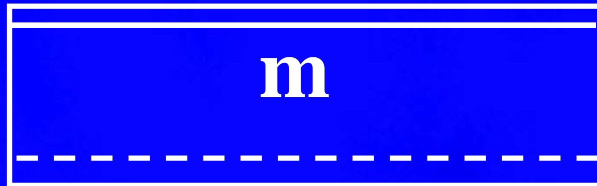


5.2 全局栈式存储分配

2、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）

5.2 全局栈式存储分配

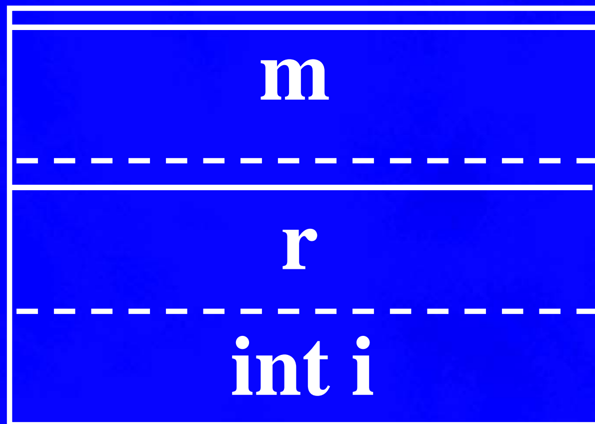
2、运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



m

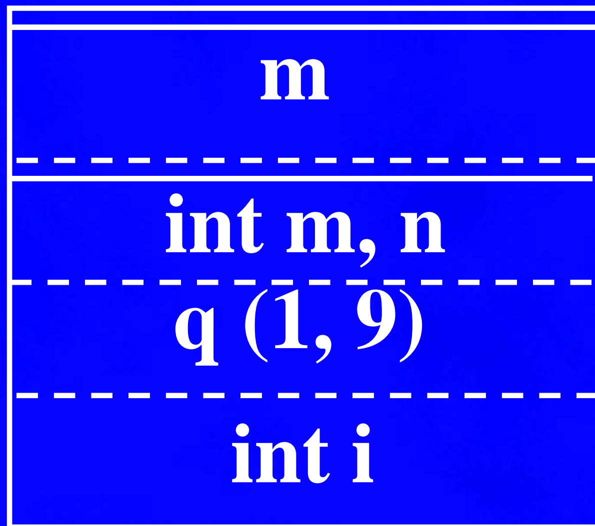
5.2 全局栈式存储分配

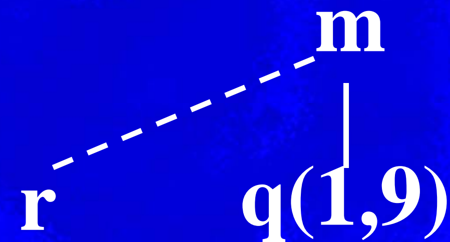
2、运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



5.2 全局栈式存储分配

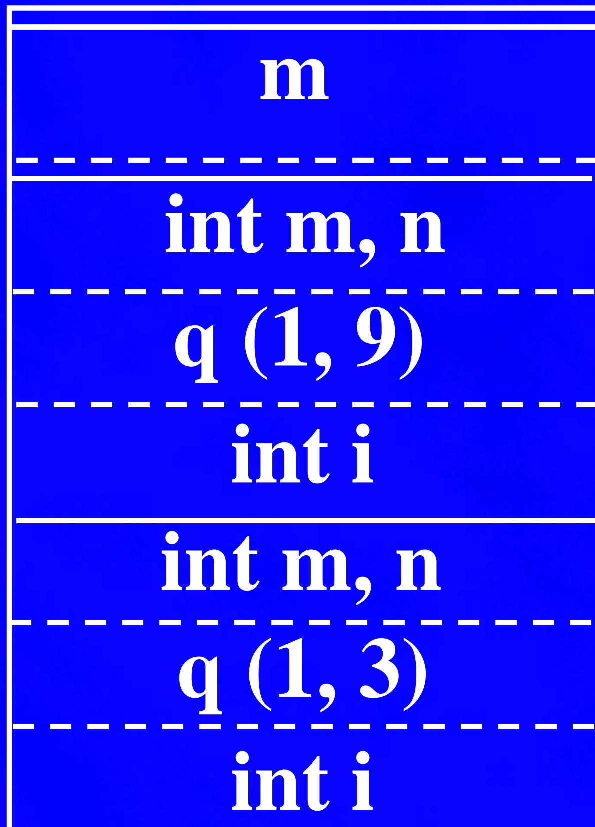
2、运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）

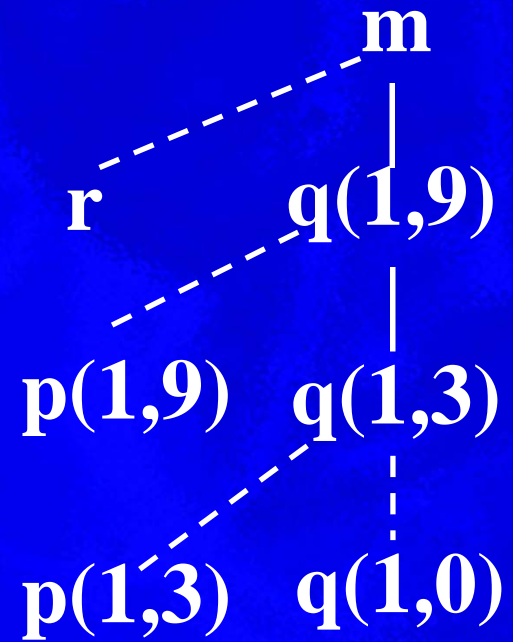




5.2 全局栈式存储分配

2、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）





5.2 全局栈式存储分配

5.2.3 调用序列

- 过程调用和过程返回都需要执行一些代码来管理运行栈，保存或恢复机器状态等
- 过程调用序列

  在过程调用时执行的分配活动记录，以及把信息填入其域中的代码
- 过程返回序列

  在过程返回时执行的恢复机器状态、回收活动记录，以及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，分别位于调用过程和被调用过程中

5.2 全局栈式存储分配

- 即使是同一种语言，过程调用序列、返回序列和活动记录的布局也会因实现而异
- 设计这些序列和活动记录布局的一些原则
  - 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

5.2 全局栈式存储分配

- 即使是同一种语言，过程调用序列、返回序列和活动记录的布局也会因实现而异
- 设计这些序列和活动记录布局的一些原则
  - 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

5.2 全局栈式存储分配

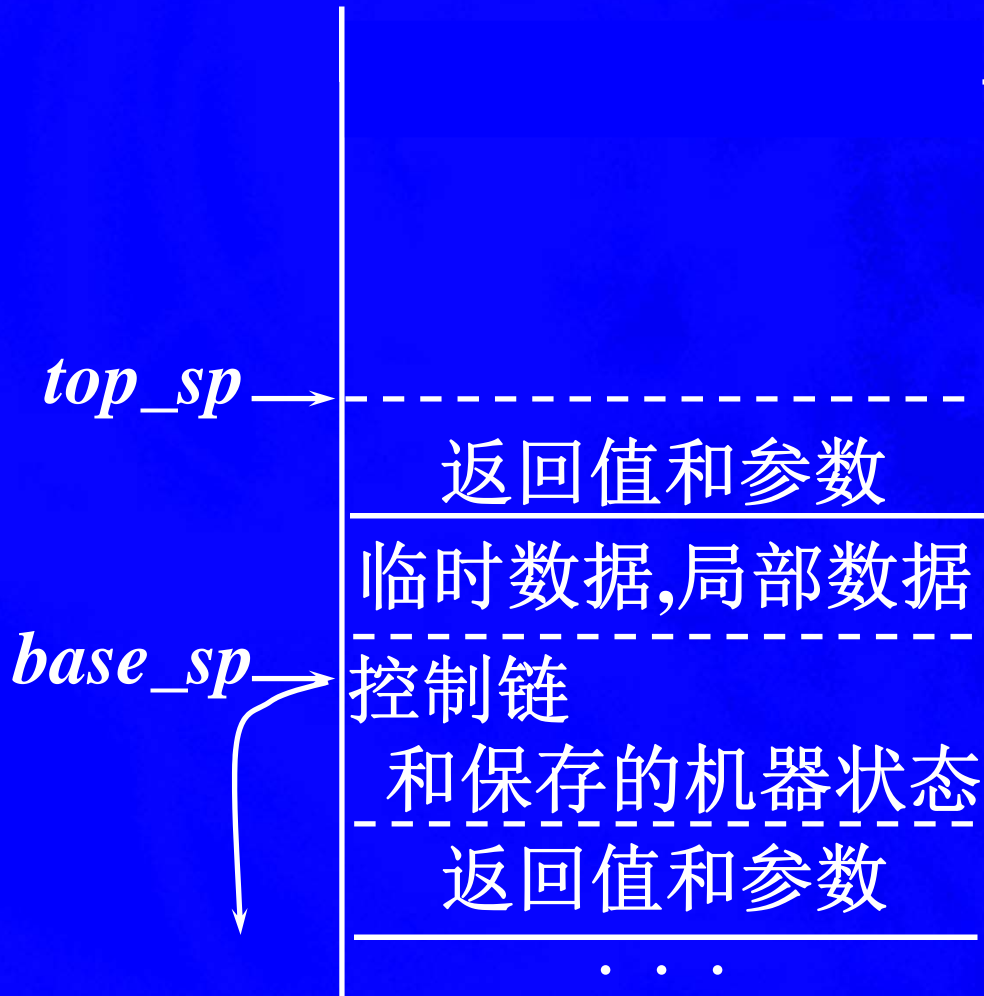
- 即使是同一种语言，过程调用序列、返回序列和活动记录的布局也会因实现而异
- 设计这些序列和活动记录布局的一些原则
  - 用同样的代码来执行各个活动的保存和恢复

5.2 全局栈式存储分配

1、过程p调用过程q的调用序列

5.2 全局栈式存储分配

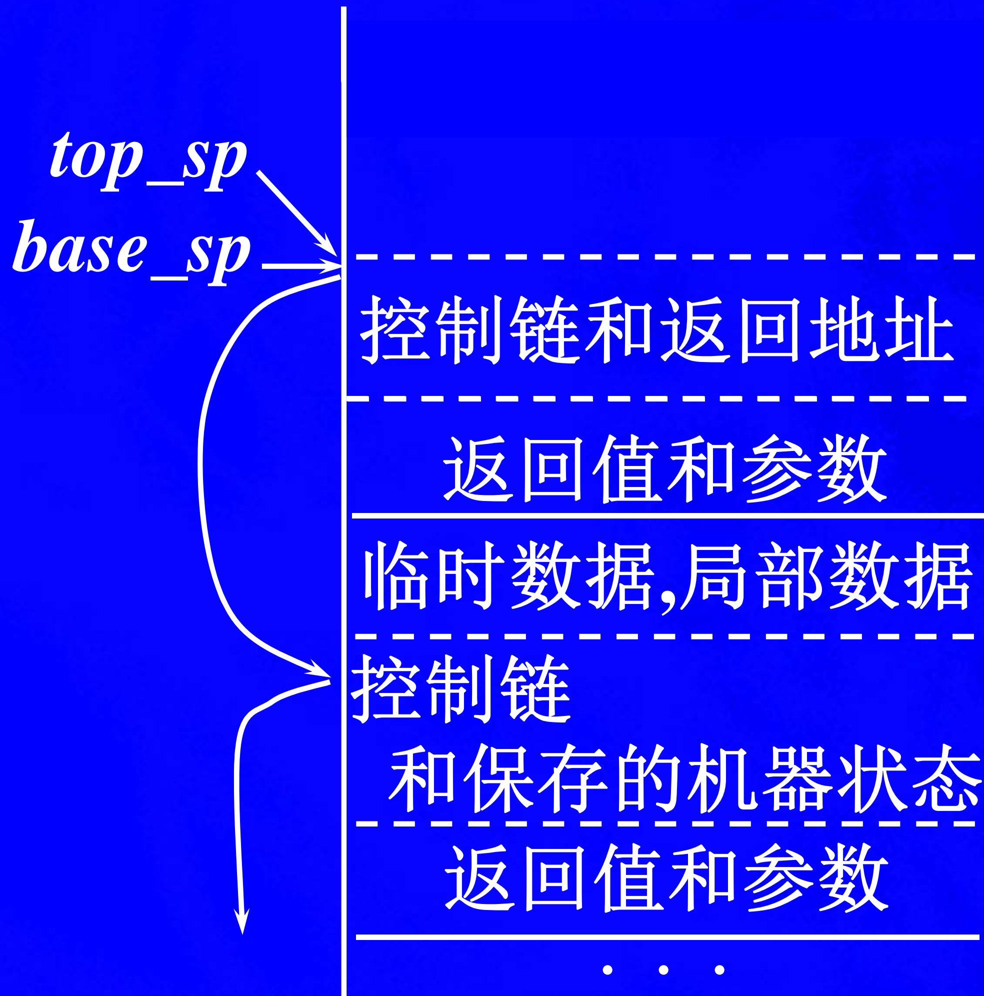
1、过程p调用过程q的调用序列



(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

5.2 全局栈式存储分配

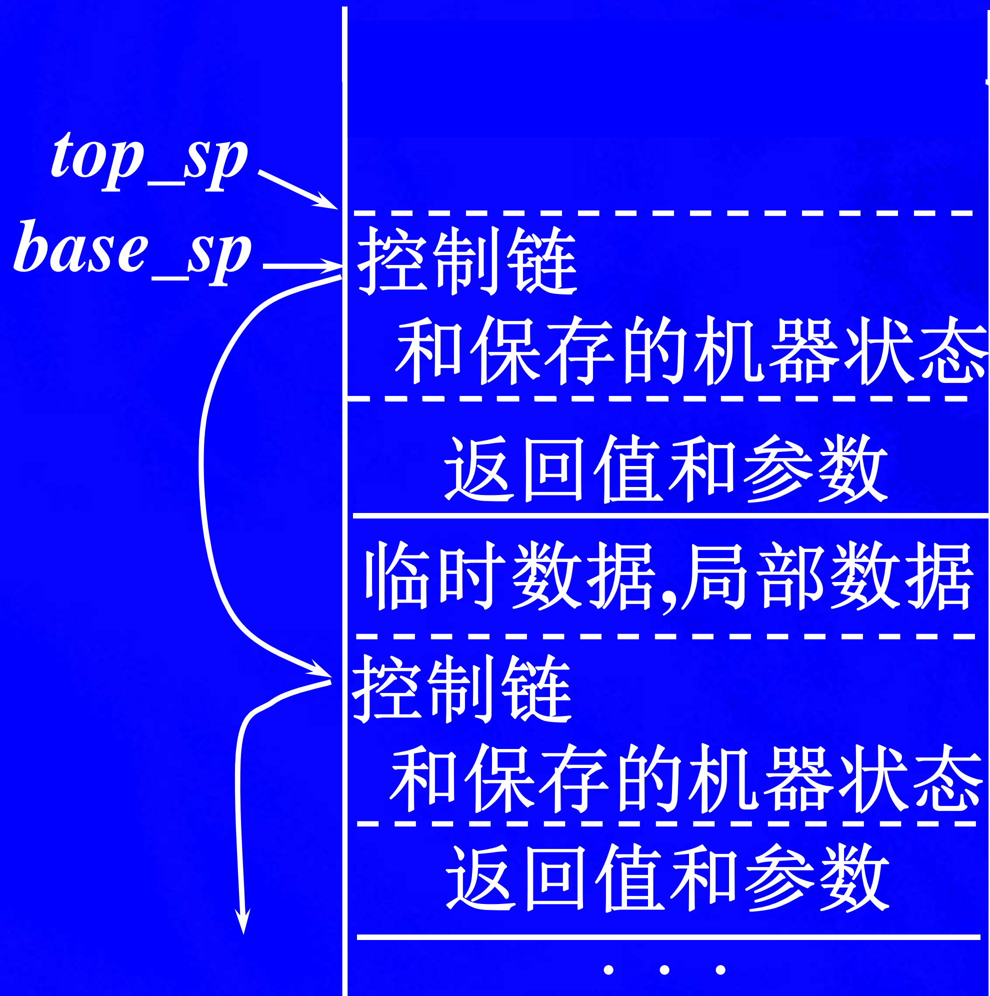
1、过程p调用过程q的调用序列



(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

5.2 全局栈式存储分配

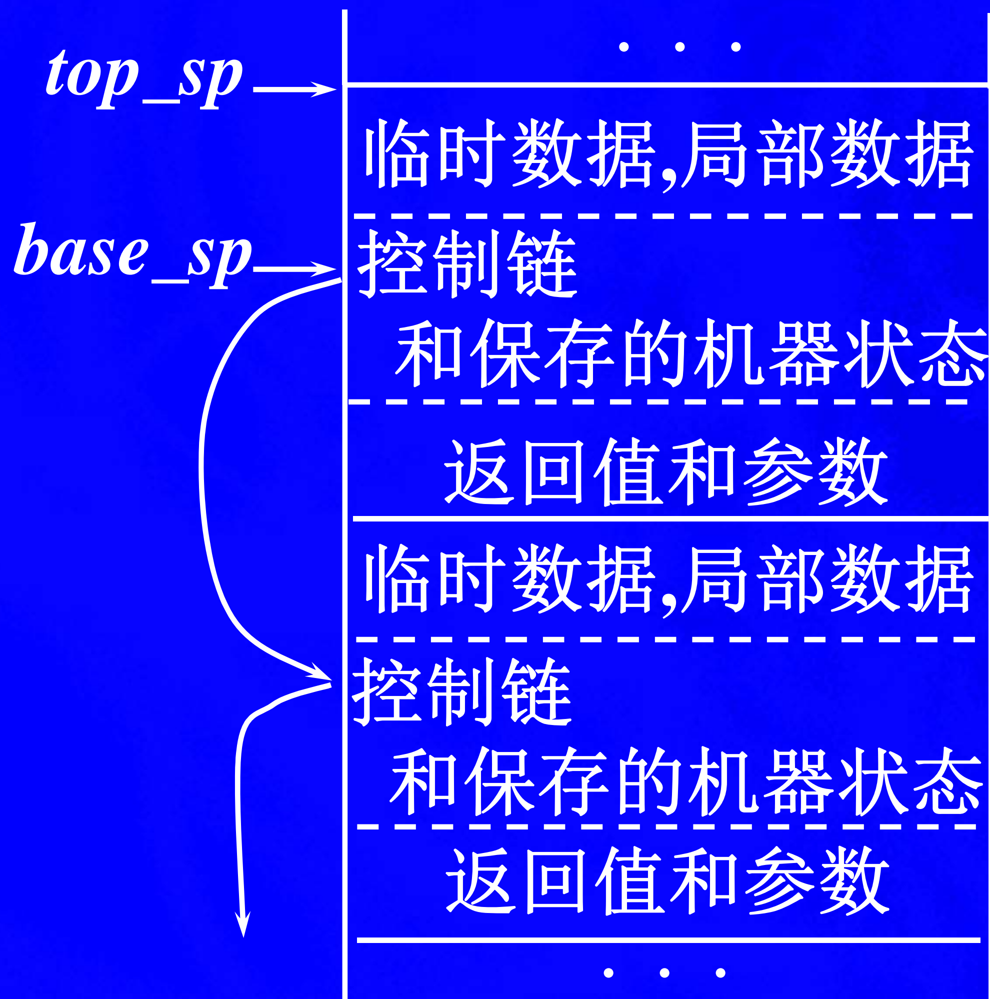
1、过程p调用过程q的调用序列



(3) q保存寄存器的值和其他机器状态信息

5.2 全局栈式存储分配

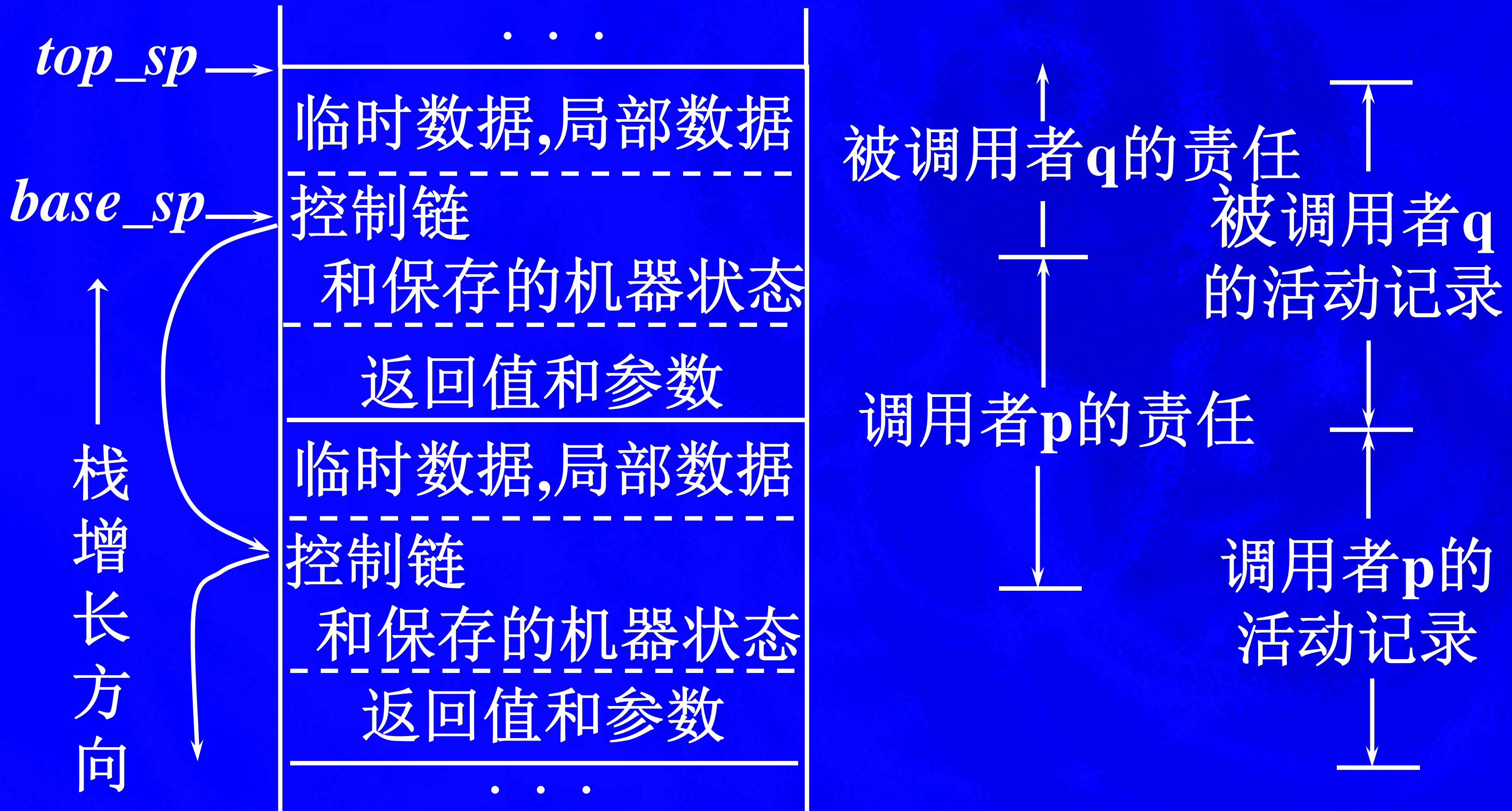
1、过程p调用过程q的调用序列



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

5.2 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



5.2 全局栈式存储分配

2、过程p调用过程q的返回序列

5.2 全局栈式存储分配

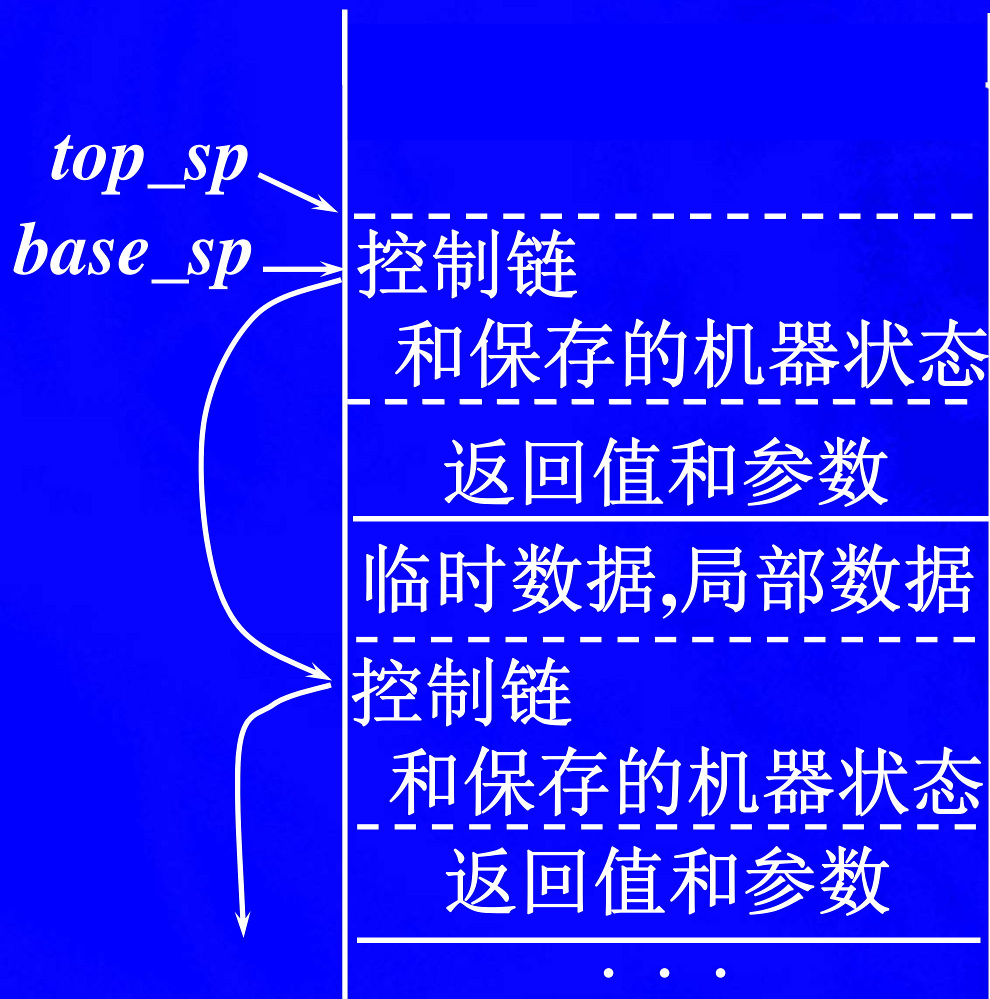
2、过程p调用过程q的返回序列

(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

5.2 全局栈式存储分配

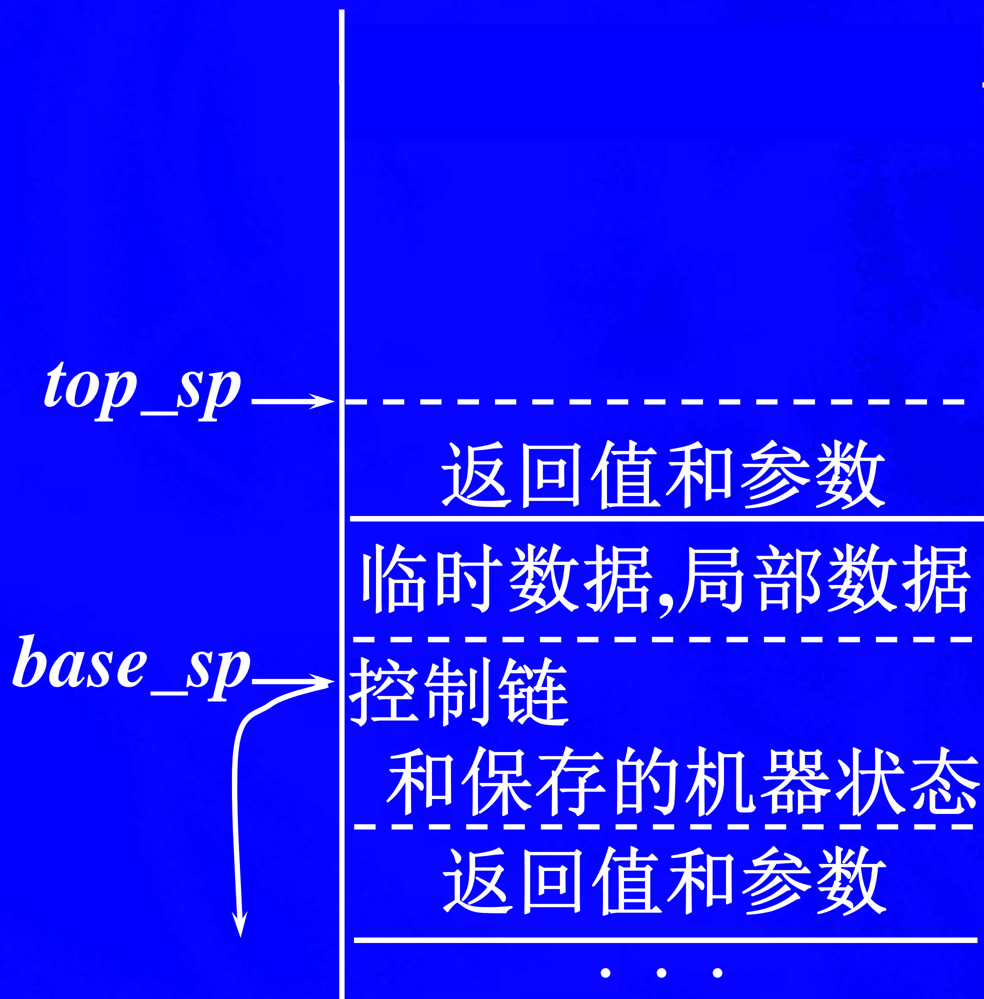
2、过程p调用过程q的返回序列



(2) 与调用序列的步骤(4)相对应，q增加*top_sp*的值

5.2 全局栈式存储分配

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，把控制转到p

5.2 全局栈式存储分配

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

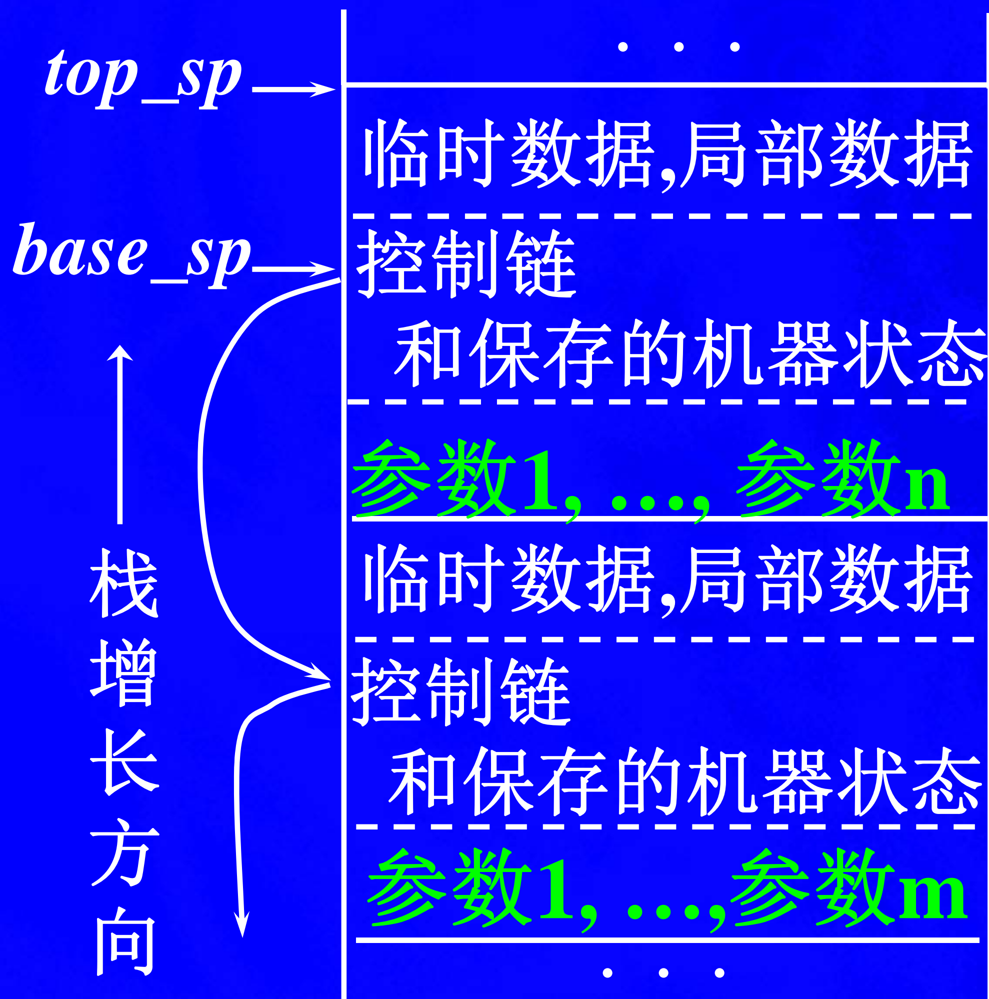
5.2 全局栈式存储分配

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

5.2 全局栈式存储分配

3、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

5.2 全局栈式存储分配

3、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1个参数的位置

5.2 全局栈式存储分配

3、过程的参数个数可变的情况

	. . .
top_sp →	临时数据,局部数据
base_sp →	控制链 和保存的机器状态
	参数1, …, 参数n
	临时数据,局部数据
	控制链 和保存的机器状态
	参数1, …,参数m
	. . .

↑ 栈增长方向

(4) 被调用函数根据第1个参数到栈中取第2、第3个参数等等

5.2 全局栈式存储分配

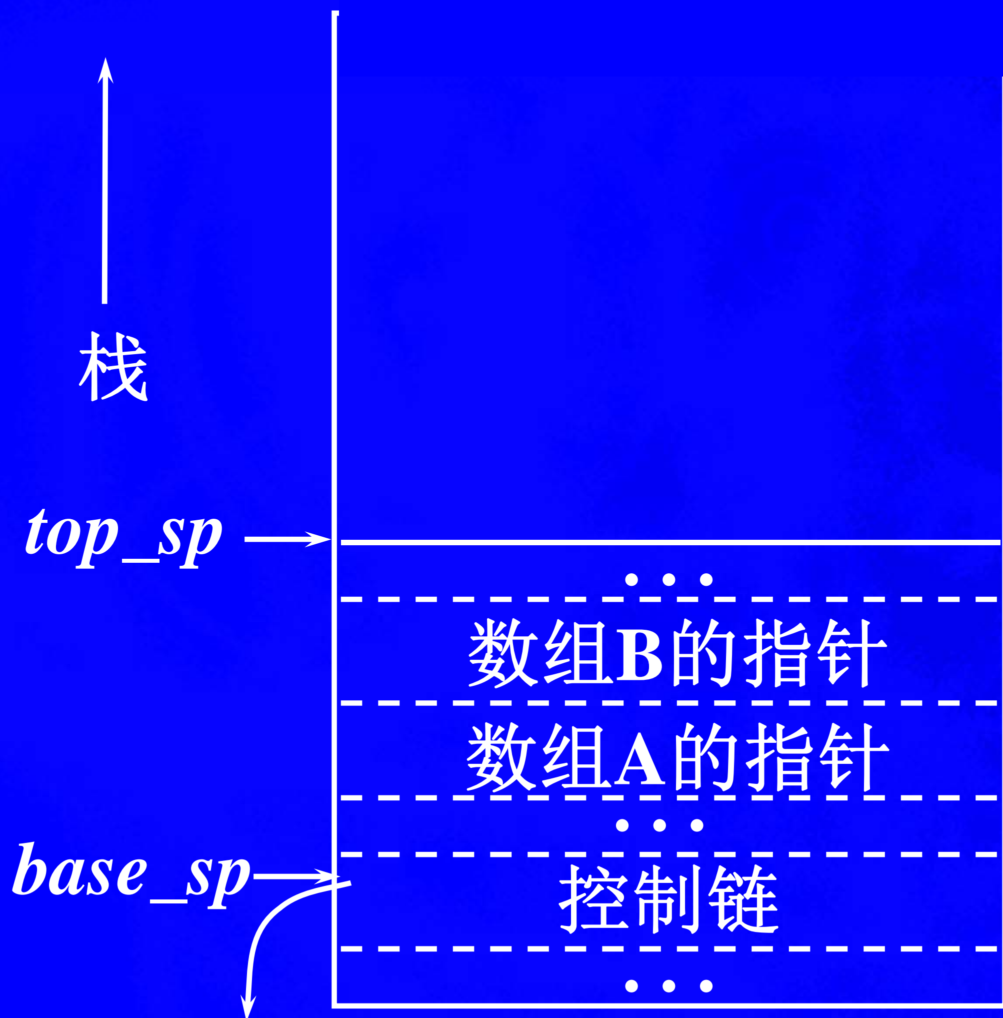
5.2.4 栈上可变长度数据

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

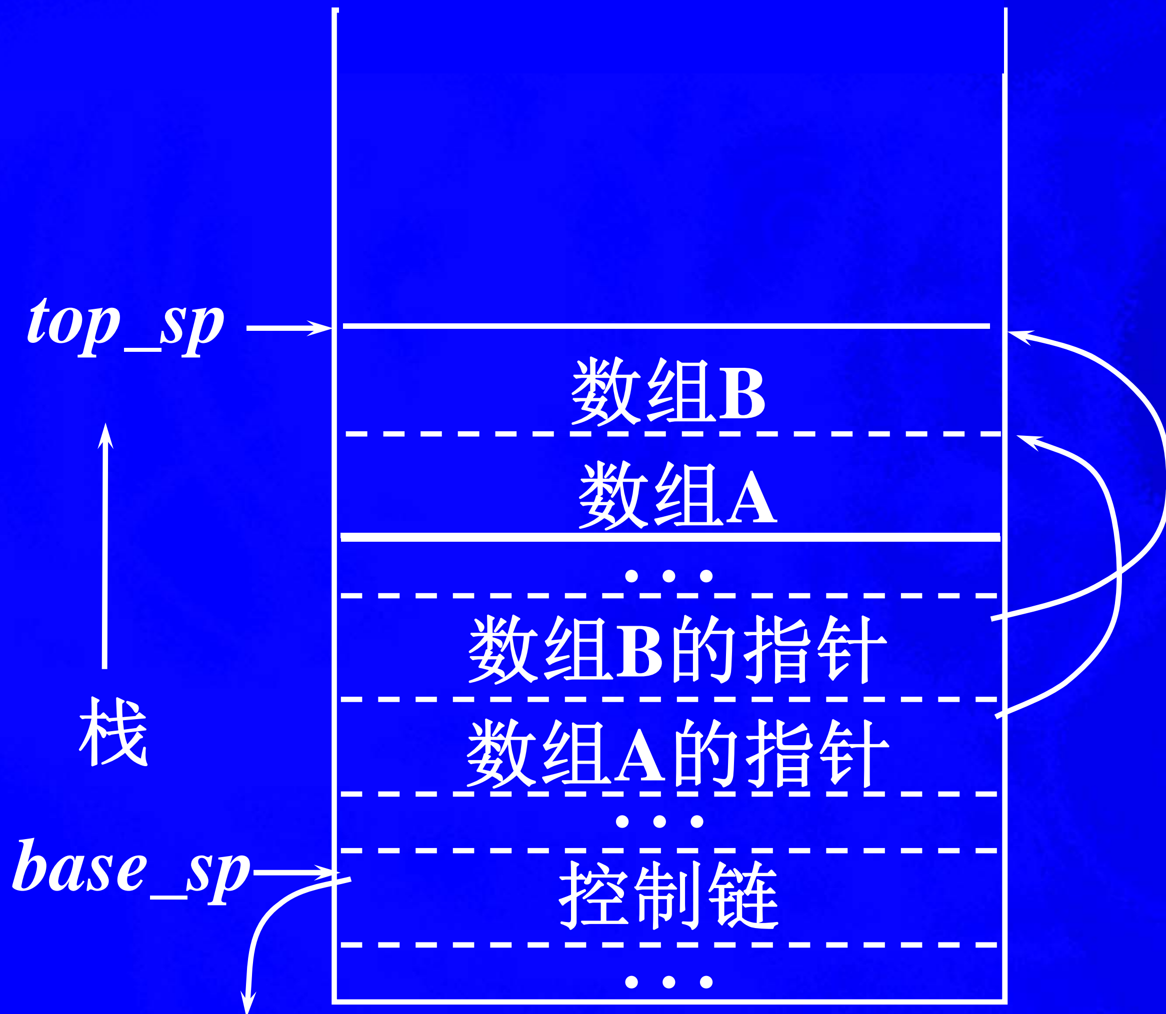
5.2 全局栈式存储分配



(1) 编译时，在活动记录中为这样的数组分别分配一个存放数组指针的单元

访问动态分配的数组

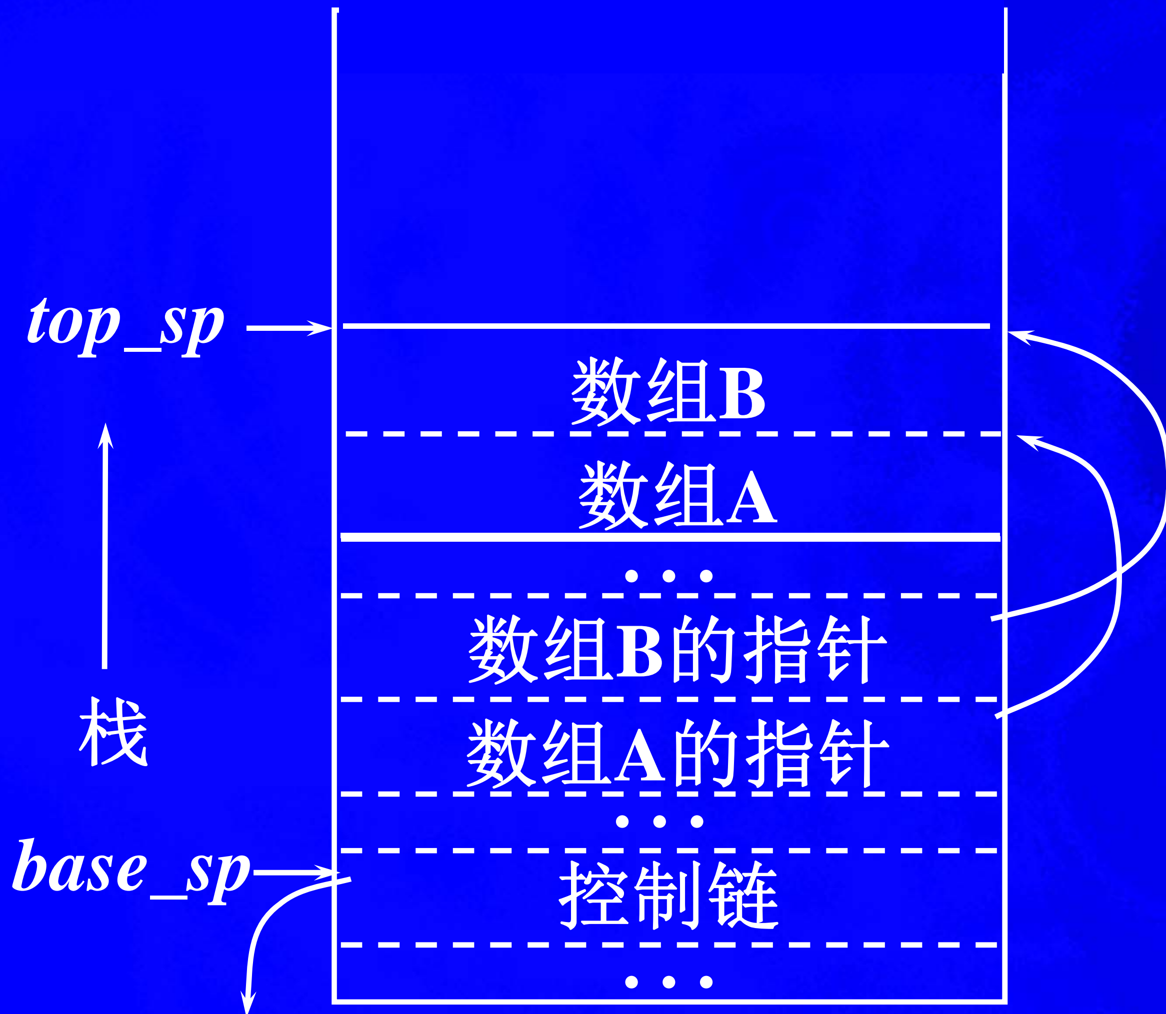
5.2 全局栈式存储分配



(2) 运行时，这些指针指向分配在栈顶的存储空间

访问动态分配的数组

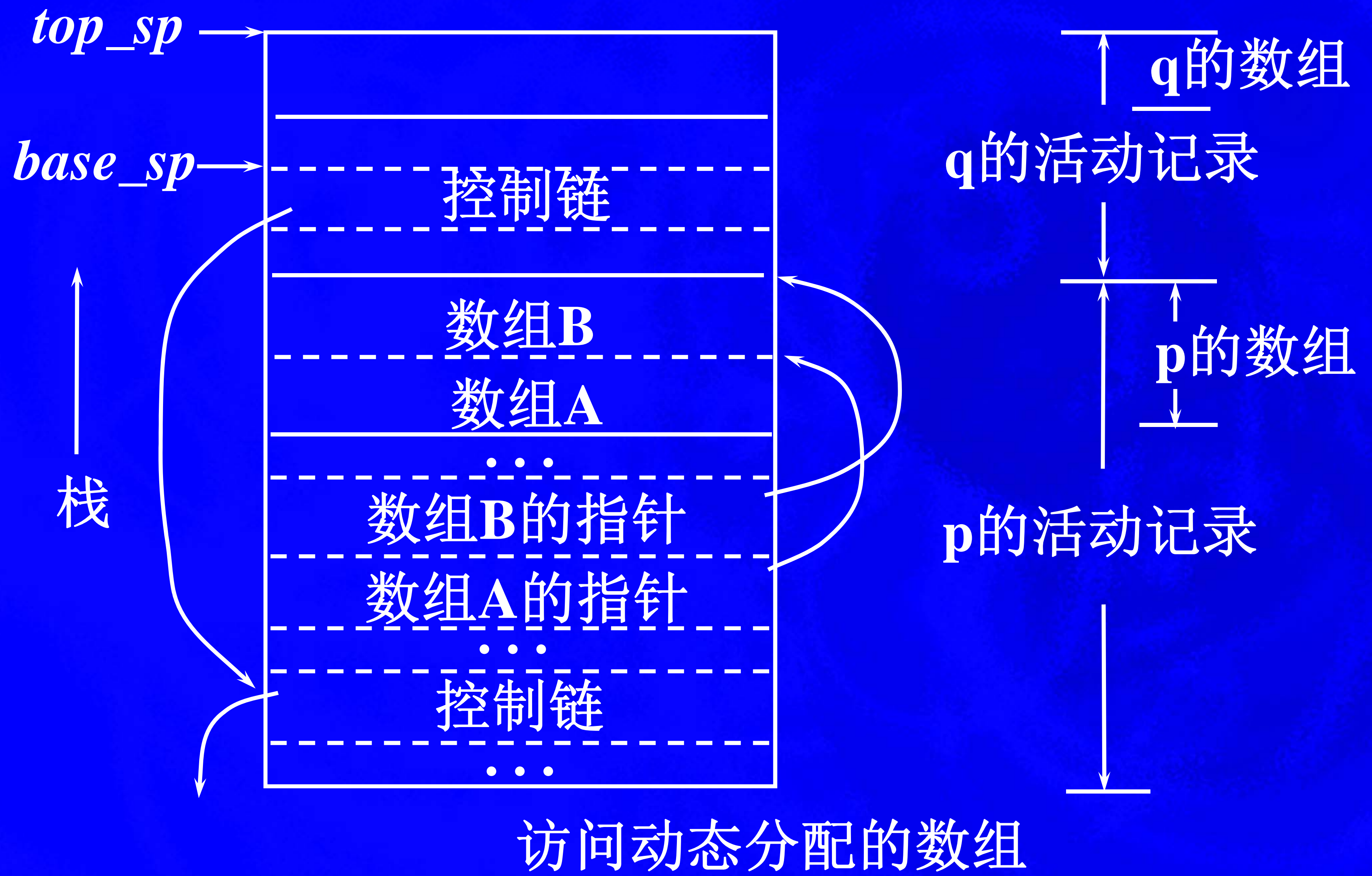
5.2 全局栈式存储分配



(3) 运行时，对数组A和B的访问都要通过活动记录中的相应数组指针来间接访问

访问动态分配的数组

5.2 全局栈式存储分配



访问动态分配的数组

5.2 全局栈式存储分配

5.2.5 悬空引用

悬空引用：引用某个已被回收的存储单元

5.2 全局栈式存储分配

5.2.5 悬空引用

悬空引用：引用某个已被回收的存储单元

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

5.3 非局部名字的访问

本节介绍

- 无过程嵌套的静态作用域（**C**语言）
- 有过程嵌套的静态作用域（**Pascal**语言）
- 动态作用域（**LISP**语言）

5.3 非局部名字的访问

5.3.1 无过程嵌套的静态作用域

- 过程体中的非局部引用可以直接使用静态确定的地址
- 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
- 过程可以作为参数来传递，也可以作为结果来返回

5.3 非局部名字的访问

5.3.2 有过程嵌套的静态作用域

```
sort
    readArray
    exchange
    quickSort
        partition
```

5.3 非局部名字的访问

5.3.2 有过程嵌套的静态作用域

- 过程嵌套深度

sort	1
readArray	2
exchange	2
quickSort	2
partition	3

- 变量的嵌套深度：以它的声明所在过程的嵌套深度作为该名字的嵌套深度

5.3 非局部名字的访问

- 访问链
  - 用来寻找非局部名字的存储单元

5.3 非局部名字的访问

- 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，$n_a \le n_p$。如何访问a的存储单元？

sort	1
readArray	2
exchange	2
quickSort	2
partition	3

5.3 非局部名字的访问

- 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，$n_a \le n_p$。如何访问a的存储单元？

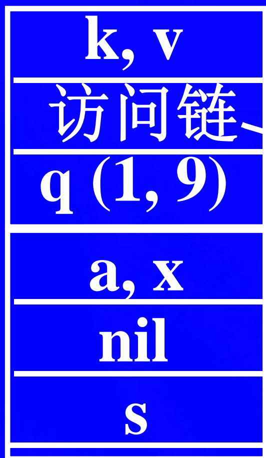
  – 从栈顶的活动记录开始，追踪访问链$n_p - n_a$次
  – 到达a的声明所在过程的活动记录
  – 访问链的追踪用间接操作就可完成

```
sort                  1
   readArray          2
   exchange           2
   quickSort                2
       partition      3
```

5.3 非局部名字的访问

- 访问非局部名字的存储单元

sort
 readArray
 exchange
 quickSort
 partition



5.3 非局部名字的访问

- 过程p对变量a访问时，a的地址由下面的二元组表示：

  （$n_p - n_a$，a在活动记录中的偏移）

5.3 非局部名字的访问

- 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

sort 1

readArray 2

exchange 2

quickSort 2

partition 3

这时x肯定就声明在p中

5.3 非局部名字的访问

- 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

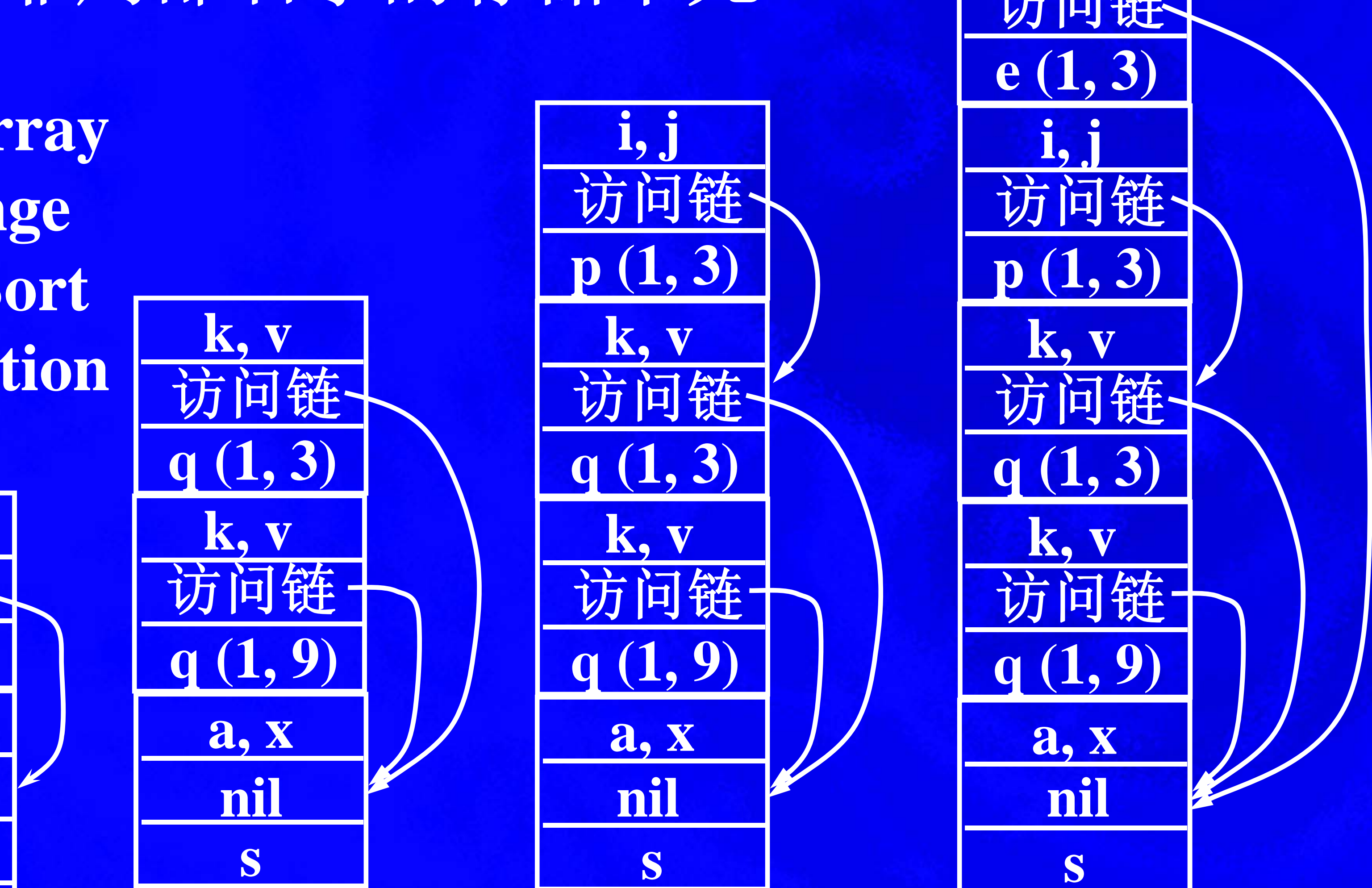
(1) $n_p < n_x$的情况

  - 被调用过程的访问链必须指向调用过程的活动记录的访问链

5.3 非局部名字的访问

- 访问非局部名字的存储单元

sort
 readArray
 exchange
 quickSort
  partition



5.3 非局部名字的访问

- 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

5.3 非局部名字的访问

- 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**

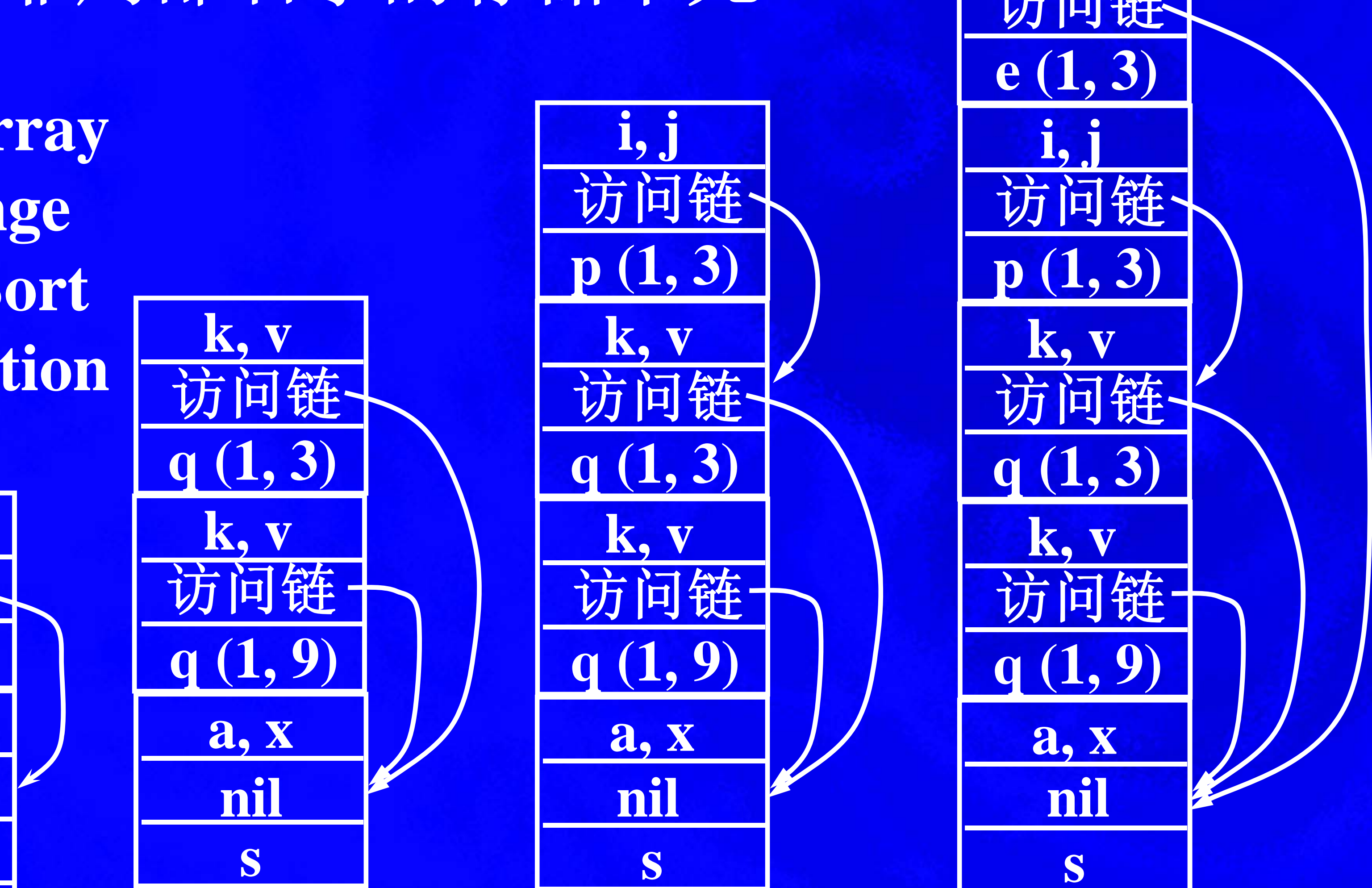
(2) $n_p \geq n_x$的情况

  - 追踪访问链$n_p - n_x + 1$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
  - 所到达的访问链就是**x**的活动记录中的访问链应该指向的那个访问链

5.3 非局部名字的访问

- 访问非局部名字的存储单元

sort
 readArray
 exchange
 quickSort
 partition



5.3 非局部名字的访问

```
program param(input, output);（过程作为参数）
  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.
```

过程作为参数传递时，怎样在该过程被激活时建立它的访问链?

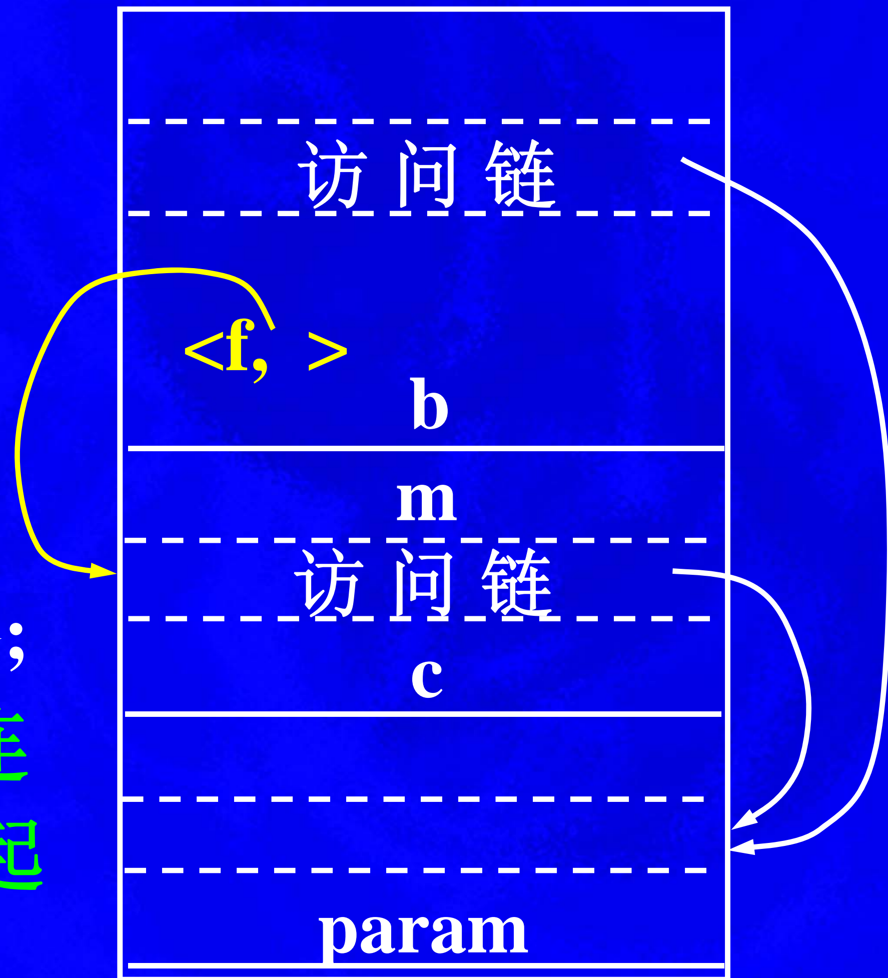
从b的访问链难以建立f 的访问链

5.3 非局部名字的访问

program param(input, output);（过程作为参数）

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

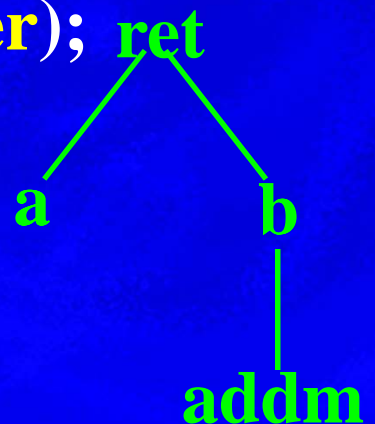
f 的起始地址连同它的访问链一起传递给b



5.3 非局部名字的访问

```
program ret (input, output);（过程作为返回值）
    var f: function (integer): integer;
    function a: function (integer): integer;
        var m: integer;
        function addm (n: integer): integer;
            begin return m+n end;
        begin m:= 0; return addm end;
    procedure b (g: function (integer): integer);
        begin writeln (g(2)) end;
    begin
        f := a; b(f)
    end.
```

执行addm时，a的活动记录已不存在，取不到m的值



5.3 非局部名字的访问

- C语言的函数声明不能嵌套，函数不论在什么情况下激活，要访问的数据分成两种情况
  - 非静态局部变量（包括形式参数），它们分配在活动记录栈顶的那个活动记录中
  - 外部变量（包括定义在其他源文件之中的外部变量）和静态的局部变量，它们都分配在静态数据区

5.4 参 数 传 递

5.4.1 值调用

- 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，并把其右值放入形参的存储单元中

5.4 参 数 传 递

5.4.2 引用调用

- 实参的左值传给被调用过程
- 引用调用可以如下实现：
  - 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，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，任何对形参的访问都是通过传给该过程的指针来间接访问实参

5.4 参 数 传 递

5.4.3 换名调用

从概念上说，每次调用时，用实参表达式对形参进行正文替换，然后再执行

```
procedure swap(var x, y: integer);
    var temp: integer;
        begin
            temp := x;
            x := y;
            y := temp
        end
```

5.4 参 数 传 递

5.4.3 换名调用

从概念上说，每次调用时，用实参表达式对形参进行正文替换，然后再执行

```
procedure swap(var x, y: integer);
  var temp: integer;
    begin
      temp := x;
      x := y;
      y := temp
    end
```

例如： 调用swap(i, a[i])

5.4 参 数 传 递

5.4.3 换名调用

从概念上说，每次调用时，用实参表达式对形参进行正文替换，然后再执行

```
procedure swap(var x, y: integer);
  var temp: integer;
    begin
      temp := x;
      x := y;
      y := temp
    end
```

例如：　　调用swap(i, a[i])

替换结果：

```
temp := i;
i := a[i];
a[i] := temp
```

5.4 参 数 传 递

5.4.3 换名调用

从概念上说，每次调用时，用实参表达式对形参进行正文替换，然后再执行

```
procedure swap(var x, y: integer);
  var temp: integer;
    begin
      temp := x;
      x := y;
      y := temp
    end
```

例如： 调用swap(i, a[i])

替换结果：

```
temp := i;
i := a[i];
a[i] := temp
```

交换两个数据的程序并非总是正确

5.5 堆 管 理

堆式分配

- 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，这些对象的生存期没有被约束在创建它们的过程活动的生存期之内
  - 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：free（C）或delete（C++）
  - 垃圾收集器自动收集（Java）。本课程不做介绍

5.5 堆 管 理

5.5.1 内存管理器

- 内存管理器把握的基本信息是堆中的空闲空间
  - 分配函数
  - 回收函数
- 内存管理器应具有下列性质
  - 空间有效性：极小化程序需要的堆空间总量
  - 程序有效性：较好地利用内存子系统，使得程序能运行得快一些
  - 低开销：分配和回收操作所花时间在整个程序执行时间中的比例尽量小

5.5 堆 管 理

5.5.2 计算机内存分层

5.5 堆 管 理

5.5.2 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
- 不同指令的执行时间的差别非常可观
- 差异源于硬件技术的基本局限：构造不出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
- 数据密集型程序可通过恰当利用内存子系统获益

5.5 堆 管 理

5.5.3 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

5.5 堆 管 理

5.5.3 程序局部性

- 从代码本身很难看出哪部分代码会被频繁使用，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

5.5 堆 管 理

例： 一个结构体大数组

```
struct student {
        int num;
        char name[20];
        …    …
}
struct  student st[10000];
```

分拆成若干个数组

```
int num[10000];
char name[10000][20];
…    …
```

- 若是顺序处理每个结构体的多个域，左边方式的数据局部性较好
- 若是先顺序处理每个结构的num域，再处理每个结构的name域，…，则右边方式的数据局部性较好
- 最好是按左边方式编程，由编译器决定是否需要将数据按右边方式布局

5.5 堆 管 理

5.5.4 手工回收请求

- 程序员在程序中显式释放堆块来达到回收堆块的目的
  - 内存泄漏：没有释放已经引用不到的堆块

    只要内存没有用尽，它就不影响程序的正确性

    自动无用单元收集通过回收所有无用单元来摆脱内存泄漏
  - 悬空引用：引用已经被释放的堆块

    因重复释放数据对象而引起

    悬空引用容易导致不会被捕获的错误

本 章 要 点

- 影响存储分配策略的语言特征
- 各种存储分配策略，主要了解静态分配和动态栈式分配
- 活动记录中各种数据域的作用和布局
- 非局部数据访问的实现方法
- 各种参数传递方式及其实现
- 堆管理

例　题　1

一个C语言程序及其在x86/Linux操作系统上的编译结果如下。根据所生成的汇编程序来解释程序中4 个变量的存储分配、作用域、生存期和置初值方式等方面的区别

```
static long aa = 10;
short bb = 20;
func( )
{
   static long cc = 30;
   short dd = 40;
}
```

例　题　1

```
.data
.align 4
   .type aa,@object
   .size aa,4
aa:
   .long 10
.globl bb
   .align 2
   .type bb,@object
   .size bb,2
bb:
   .value 20
```

```
   .align 4
   .type  cc.2,@object
   .size   cc.2,4
cc.2:
   .long 30
.text
   .align 4
.globl func
func:
   . . .
   movw $40,-2(%ebp)
   . . .
```

例　题　1

```
.data                         |       .align 4
.align 4                      |       .type  cc.2,@object
   .type aa,@object           |       .size   cc.2,4
   .size aa,4                 |   cc.2:
aa:                           |       .long 30
   .long 10                   |   .text
.globl bb                     |       .align 4
   .align 2                   |   .globl func
   .type bb,@object           |   func:
   .size bb,2                 |       . . .
bb:                           |       movw $40,-2(%ebp)
   .value 20                  |       . . .
```

例　题　1

```
.data
.align 4
   .type aa,@object
   .size aa,4
aa:
   .long 10
.globl bb
   .align 2
   .type bb,@object
   .size bb,2
bb:
   .value 20
```

```
      .align 4
      .type  cc.2,@object
      .size   cc.2,4
cc.2:
      .long 30
.text
      .align 4
.globl func
func:
      . . .
      movw $40,-2(%ebp)
      . . .
```

例 题 1

```
.data
.align 4
   .type aa,@object
   .size aa,4
aa:
   .long 10
.globl bb
   .align 2
   .type bb,@object
   .size bb,2
bb:
   .value 20
```

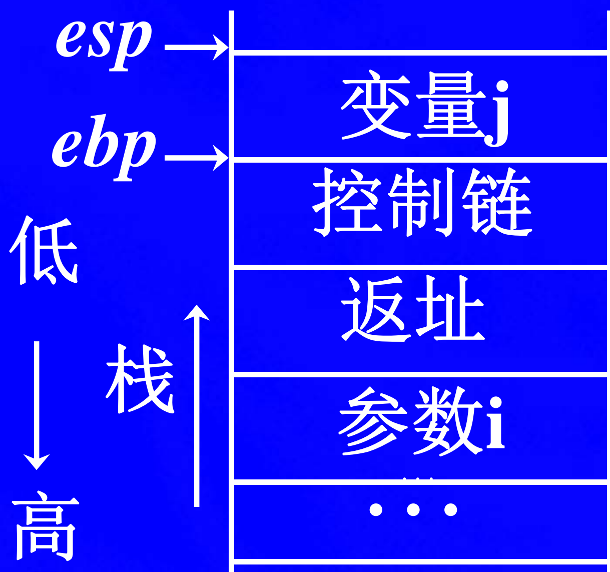
```
     .align 4
     .type  cc.2,@object
     .size   cc.2,4
cc.2:
     .long 30
.text
     .align 4
.globl func
func:
     . . .
     movw $40,-2(%ebp)
     . . .
```

例　题　2

```
func(i)
long i;
{
    long j;
    j= i -1;
    func(j);
}
```

例　题　2

```
func(i)
long i;
{
   long j;
   j= i -1;
   func(j);
}
```

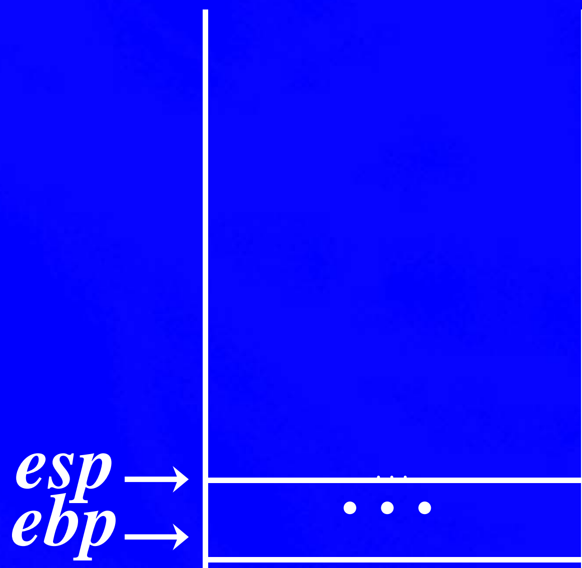



```
func:
   pushl %ebp             老的基地址指针压栈
   movl %esp,%ebp         修改基地址指针
   subl $4,%esp           为j分配空间
   movl 8(%ebp),%edx      取i到寄存器
   decl %edx              i – 1
   movl %edx,-4(%ebp)     i – 1 ⇒ j
   movl -4(%ebp),%eax
   pushl %eax             把实参j的值压栈
   call func              函数调用
   addl $4,%esp           恢复栈顶指针
L1:
   leave  即 movl %ebp, %esp; popl %ebp
   ret    即 popl %eip(下条指令地址）
```

例　题　2

```
func(i)
long i;
{
    long j;
    j= i -1;
    func(j);
}
```

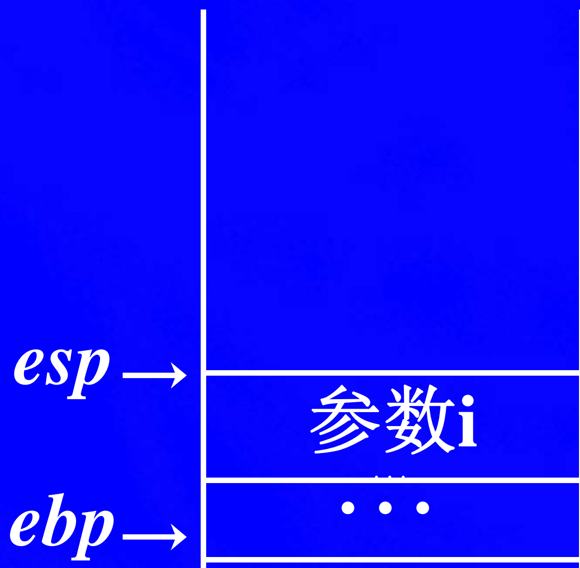
```
func:
    pushl %ebp              老的基地址指针压栈
    movl %esp,%ebp          修改基地址指针
    subl $4,%esp            为j分配空间
    movl 8(%ebp),%edx       取i到寄存器
    decl %edx               i – 1
    movl %edx,-4(%ebp)      i – 1 ⇒ j
    movl -4(%ebp),%eax
    pushl %eax              把实参j的值压栈
    call func               函数调用
    addl $4,%esp            恢复栈顶指针
L1:
    leave   即 movl %ebp, %esp; popl %ebp
    ret     即 popl %eip(下条指令地址）
```



例　题　2

```
func(i)
long i;
{
    long j;
    j= i -1;
    func(j);
}
```

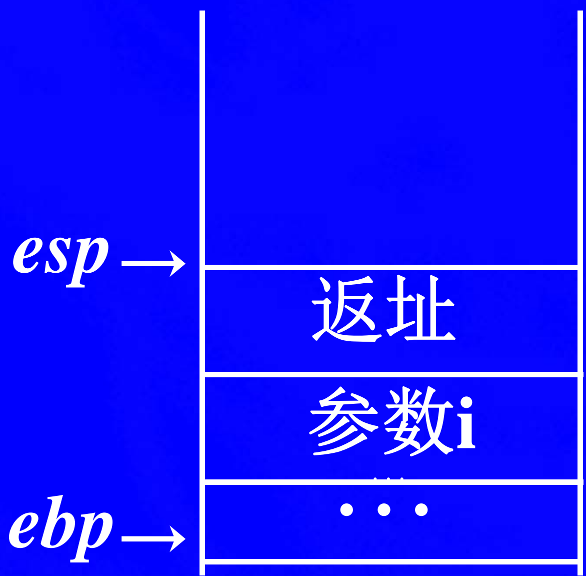
```
func:
    pushl %ebp          老的基地址指针压栈
    movl %esp,%ebp      修改基地址指针
    subl $4,%esp        为j分配空间
    movl 8(%ebp),%edx   取i到寄存器
    decl %edx           i – 1
    movl %edx,-4(%ebp)  i – 1 ⇒ j
    movl -4(%ebp),%eax
    pushl %eax          把实参j的值压栈
    call func           函数调用
    addl $4,%esp        恢复栈顶指针
L1:
    leave   即 movl %ebp, %esp; popl %ebp
    ret     即 popl %eip(下条指令地址）
```



例 题 2

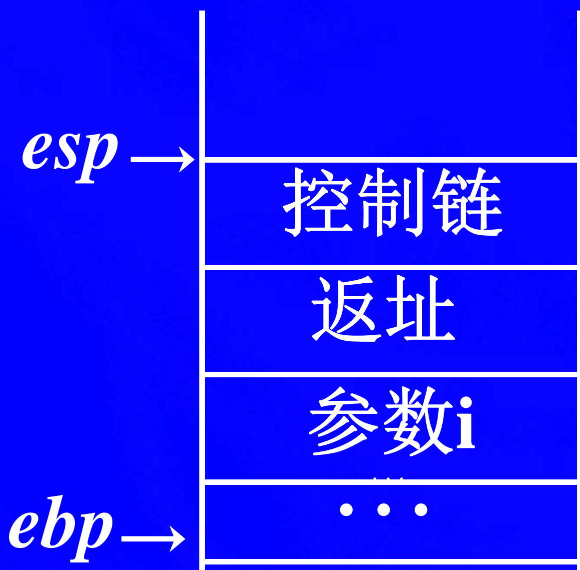
```
func(i)
long i;
{
   long j;
   j= i -1;
   func(j);
}
```

```
func:
   pushl %ebp              老的基地址指针压栈
   movl %esp,%ebp          修改基地址指针
   subl $4,%esp            为j分配空间
   movl 8(%ebp),%edx       取i到寄存器
   decl %edx               i – 1
   movl %edx,-4(%ebp)      i – 1 ⇒ j
   movl -4(%ebp),%eax
   pushl %eax              把实参j的值压栈
   call func               函数调用
   addl $4,%esp            恢复栈顶指针
L1:
   leave  即 movl %ebp, %esp; popl %ebp
   ret    即 popl %eip(下条指令地址）
```



例 题 2

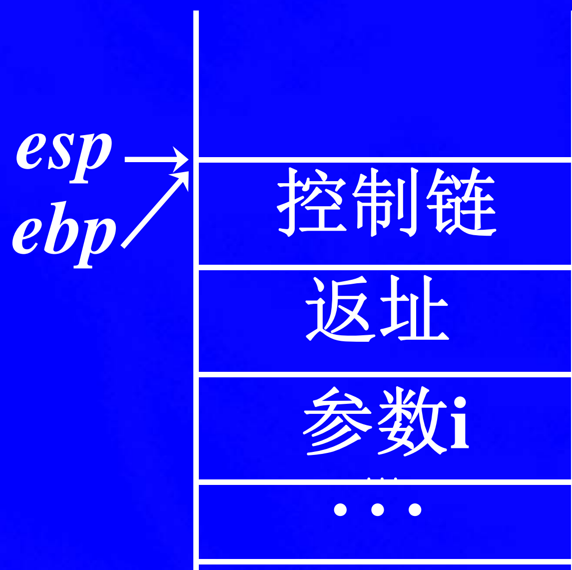
```
func(i)
long i;
{
    long j;
    j= i -1;
    func(j);
}
```




```
func:
    pushl %ebp              老的基地址指针压栈
    movl %esp,%ebp          修改基地址指针
    subl $4,%esp            为j分配空间
    movl 8(%ebp),%edx       取i到寄存器
    decl %edx               i – 1
    movl %edx,-4(%ebp)      i – 1 ⇒ j
    movl -4(%ebp),%eax
    pushl %eax              把实参j的值压栈
    call func               函数调用
    addl $4,%esp            恢复栈顶指针
L1:
     leave   即 movl %ebp, %esp; popl %ebp
     ret     即 popl %eip(下条指令地址）
```

例　题　2

```
func(i)
long i;
{
   long j;
   j= i -1;
   func(j);
}
```



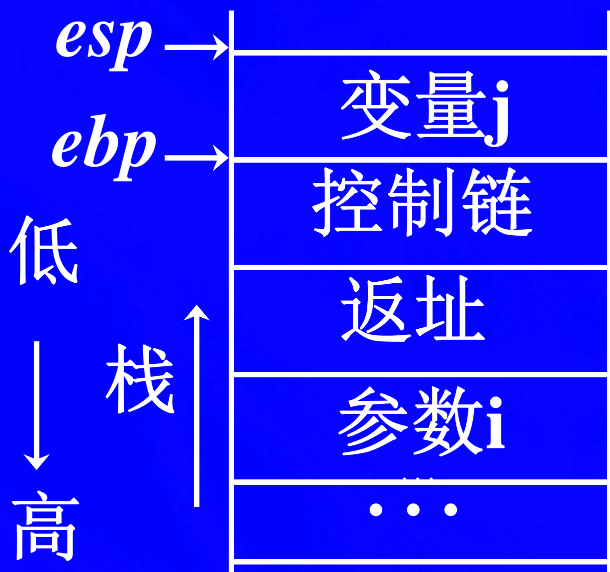
func:
- pushl %ebp　老的基地址指针压栈
- movl %esp,%ebp 修改基地址指针
- subl $4,%esp　为j分配空间
- movl 8(%ebp),%edx　取i到寄存器
- decl %edx　i – 1
- movl %edx,-4(%ebp)　i – 1 ⇒ j
- movl -4(%ebp),%eax
- pushl %eax　把实参j的值压栈
- call func　函数调用
- addl $4,%esp　恢复栈顶指针

L1:
- leave　即 movl %ebp, %esp; popl %ebp
- ret　即 popl %eip(下条指令地址）

例　题　2

```
func(i)
long i;
{
    long j;
    j= i -1;
    func(j);
}
```

```
func:
   pushl %ebp              老的基地址指针压栈
   movl %esp,%ebp          修改基地址指针
   subl $4,%esp            为j分配空间
   movl 8(%ebp),%edx       取i到寄存器
   decl %edx               i – 1
   movl %edx,-4(%ebp)      i – 1 ⇒ j
   movl -4(%ebp),%eax
   pushl %eax              把实参j的值压栈
   call func               函数调用
   addl $4,%esp            恢复栈顶指针
L1:
   leave  即 movl %ebp, %esp; popl %ebp
   ret    即 popl %eip(下条指令地址）
```



例 题 2

调用序列之一
调用序列之二

```
func(i)
long i;
{
    long j;
    j= i -1;
    func(j);
}
```

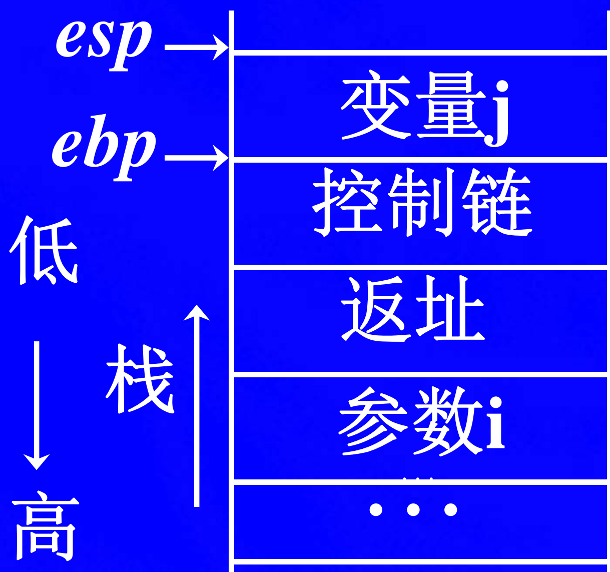
func:

pushl %ebp	老的基地址指针压栈
movl %esp,%ebp	修改基地址指针
subl $4,%esp	为j分配空间
movl 8(%ebp),%edx	取i到寄存器
decl %edx	i – 1
movl %edx,-4(%ebp)	i – 1 ⇒ j
movl -4(%ebp),%eax	
pushl %eax	把实参j的值压栈
call func	函数调用
addl $4,%esp	恢复栈顶指针

L1:

leave 即 movl %ebp, %esp; popl %ebp

ret 即 popl %eip(下条指令地址）

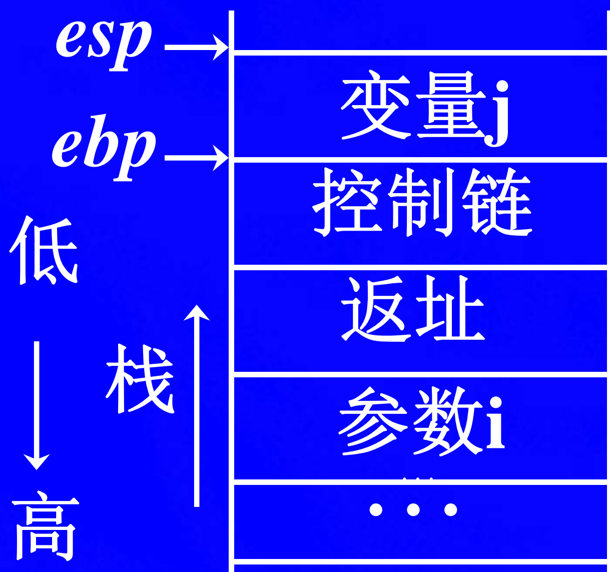


例　题　2

返回序列之一
返回序列之二

```
func(i)
long i;
{
   long j;
   j= i -1;
   func(j);
}
```

```
func:
   pushl %ebp            老的基地址指针压栈
   movl %esp,%ebp        修改基地址指针
   subl $4,%esp          为j分配空间
   movl 8(%ebp),%edx     取i到寄存器
   decl %edx             i – 1
   movl %edx,-4(%ebp)    i – 1 ⇒ j
   movl -4(%ebp),%eax
   pushl %eax            把实参j的值压栈
   call func             函数调用
   addl $4,%esp          恢复栈顶指针
L1:
   leave   即 movl %ebp, %esp; popl %ebp
   ret     即 popl %eip(下条指令地址）
```



例　题　3

```
main( )
{
        char *cp1, *cp2;

        cp1 = "12345";
        cp2 = "abcdefghij";
        strcpy(cp1,cp2);
        printf("cp1 = %s\ncp2 = %s\n", cp1, cp2);
}
```

在某些系统上的运行结果是:

```
        cp1 = abcdefghij
        cp2 = ghij
```

为什么cp2所指的串被修改了？

例　题　3

因为常量串“12345”和“abcdefghij”连续分配在常数区

执行前：

```
1 2 3 4 5 \0 a b c d e f g h i j \0
↑            ↑
cp1          cp2
```

例 题 3

因为常量串“12345”和“abcdefghij”连续分配在常数区

执行前：

```
1 2 3 4 5 \0 a b c d e f g h i j \0
↑            ↑
cp1          cp2
```

执行后：

```
a b c d e f g h i j \0 f g h i j \0
↑           ↑
cp1         cp2
```

例　题　3

因为常量串“12345”和“abcdefghij”连续分配在常数区

执行前：

```
1 2 3 4 5 \0 a b c d e f g h i j \0
↑            ↑
cp1          cp2
```

执行后：

```
a b c d e f g h i j \0 f g h i j \0
↑           ↑
cp1         cp2
```

现在的编译器大都把程序中的串常量单独存放在只读数据段中，因此运行时会报错

例　题　4

下面的程序运行时输出3个整数。试从运行环境和printf 的实现来分析，为什么此程序会有3个整数输出？

```
main( )
{
        printf(“%d, %d, %d\n”);
}
```

例　题　5

```
func(i, j, f, e)
short i, j; float f, e;{
    short i1, j1; float f1, e1;
    printf("Addresses of i, j, f, e =%o, %o, %o, %o\n",
        &i, &j, &f, &e);
    printf("Addresses of i1, j1, f1, e1 =%o, %o, %o,
        %o\n", &i1, &j1, &f1, &e1);
}
main( ) {
    short i, j; float f, e;
    func(i, j, f, e);
}
```

Address of i, j, f, e = ...36, ...42, ...44, ...54（八进制数）

Address of i1, j1, f1, e1 = ...26, ...24, ...20, ...14

例　题　5

Sizes of short, int, long, float, double = 2, 4, 4, 4, 8（在SPARC/SUN工作站上）

```
func(i, j, f, e)
short i,j; float f,e; {
    short i1,j1; float f1,e1;
    printf("Addresses of i, j, f, e =%o, %o, %o, %o\n",
        &i, &j, &f, &e);
    printf("Addresses of i1, j1, f1, e1 =%o, %o, %o,
        %o\n",  &i1, &j1, &f1, &e1);
}
main( ) {
    short i, j; float f, e;
    func(i, j, f, e);
}
```

Address of i, j, f, e = ...36, ...42, ...44, ...54（八进制数）

Address of i1, j1, f1, e1 = ...26, ...24, ...20, ...14

例　题　5

```
func(i, j, f, e)
short i,j; float f,e; {
    short i1,j1; float f1,e1;
    printf(“Addresses of i, j, f, e =%o, %o, %o, %o\n”,
        &i, &j, &f, &e);
    printf(“Addresses of i1, j1, f1, e1 =%o, %o, %o,
        %o\n”,  &i1, &j1, &f1, &e1);
}
main( ) {
    short i, j; float f, e;
    func(i, j, f, e);
}
```

Sizes of short, int, long, float, double = 2, 4, 4, 4, 8（在SPARC/SUN工作站上）

为什么4个形式参数i,j,f,e的地址间隔和它们类型的大小不一致?

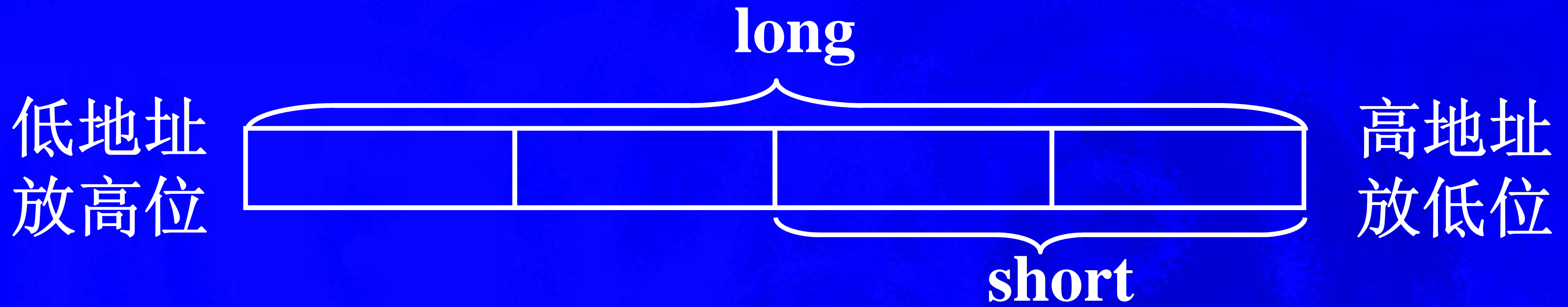
Address of i, j, f, e = ...36, ...42, ...44, ...54（八进制数）

Address of i1, j1, f1, e1 = ...26, ...24, ...20, ...14

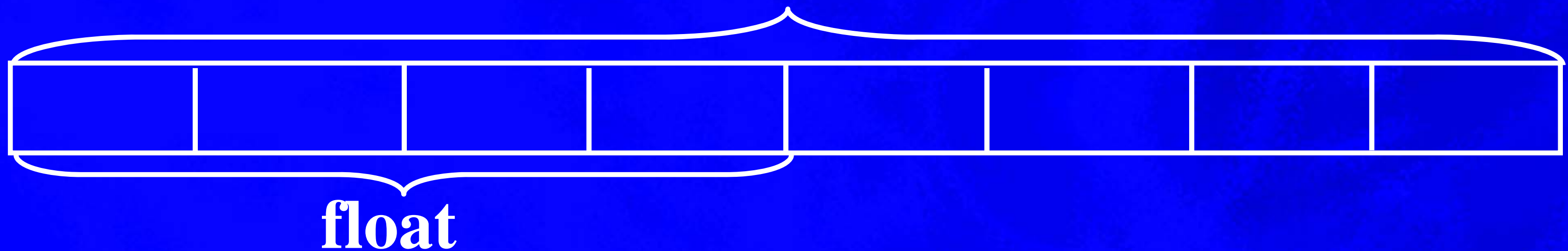
例　题　5

- 当用传统的参数声明方式时，C语言编译器是不做实在参数和形式参数的个数和类型是否一致的检查的，由程序员自己保证它们的一致性
- 但是对于形式参数和实在参数是不同的整型，或者是不同的实型，编译器则试图保证目标代码运行时能得到正确的结果，条件是，当需要把高级别类型数据向低级别类型数据转换时，不出现溢出
- 当整型或实型数据作为实在参数时，将它们分别提升到**long**和**double**类型的数据再传递到被调用函数
- 被调用函数根据形式参数所声明的类型，决定是否要将实在参数向低级别类型转换

例　题　5

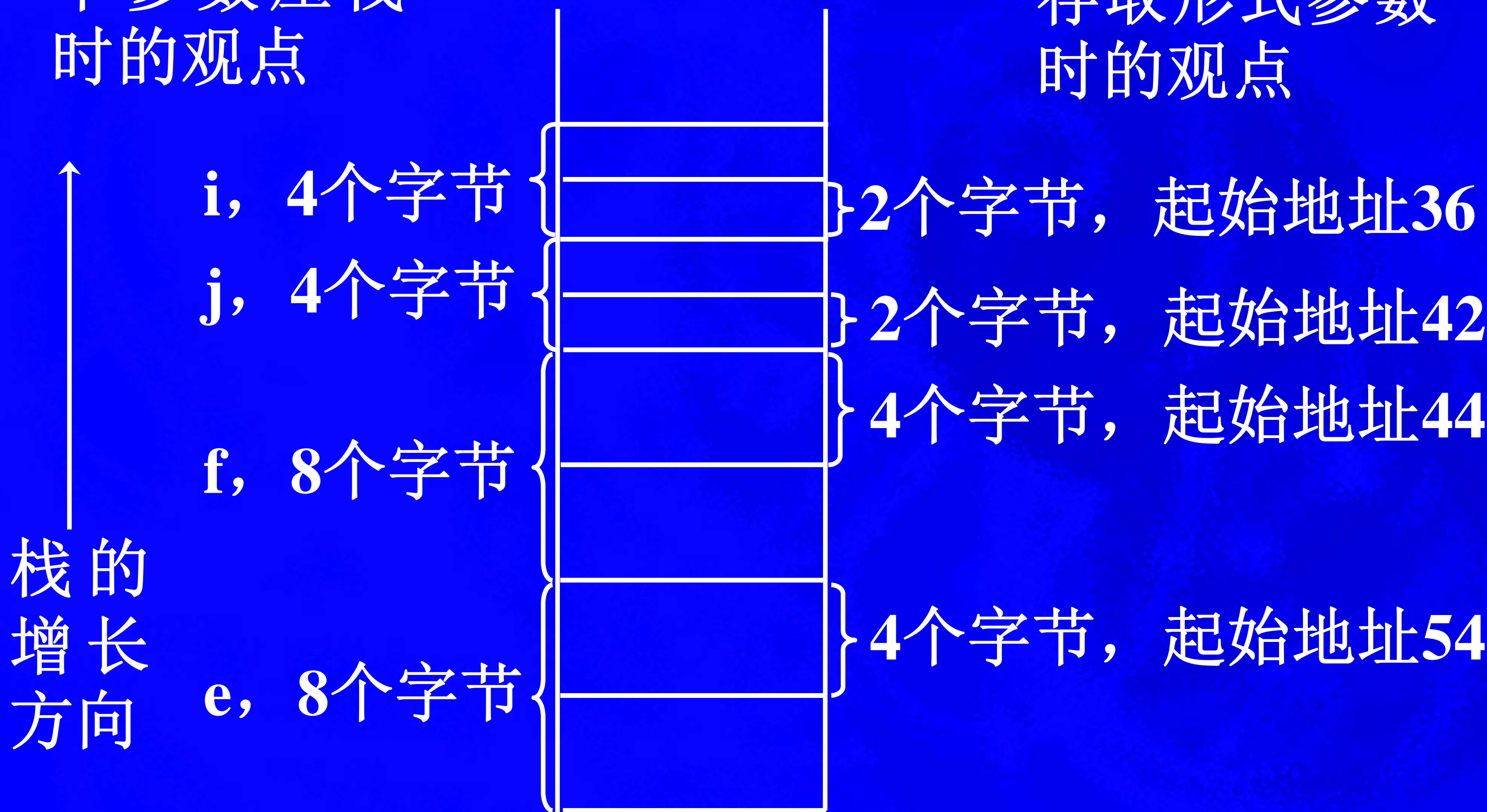


长整型和短整型



双精度型和浮点型

例 题 5



例　题　6

下面程序为什么死循环（在SPARC/SUN工作站上）？

```
main( ) { addr( );  loop( ); }
long *p;
loop( )
{
   long i, j;
   j=0;
   for(i=0; i<10; i++) { (*p)--; j++; }
}
addr( ) { long k; k=0; p=&k;}
```

例　题　6

将long *p改成short *p，long k 改成short k后，循环体执行一次便停止，为什么？

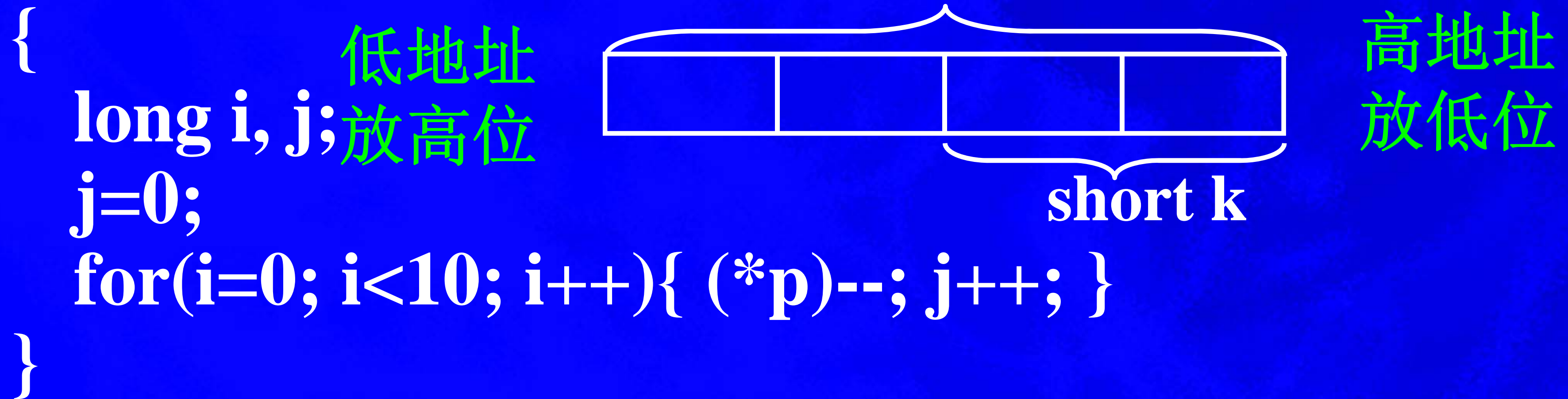
```
main( ) { addr( ); loop( ); }
short *p;
loop( )
{
   long i, j;
   j=0;
   for(i=0; i<10; i++){ (*p)--; j++; }
}
addr( ) { short k; k=0; p=&k;}
```

例　题　6

将long *p改成short *p，long k 改成short k后，循环体执行一次便停止，为什么？

```
main( ) { addr( );  loop( ); }
short *p;
loop( )
{
   long i, j;
   j=0;
   for(i=0; i<10; i++){ (*p)--; j++; }
}
addr( ) { short k; k=0; p=&k;}
```

活动记录栈是从高向低方向增长



例　题　7

```
main( )
{  func( );  printf("Return from func\n"); }

func( )
{  char s[4];
   strcpy(s,"12345678"); printf("%s\n",s); }
```

在x86/Linux操作系统上的运行结果如下：

12345678

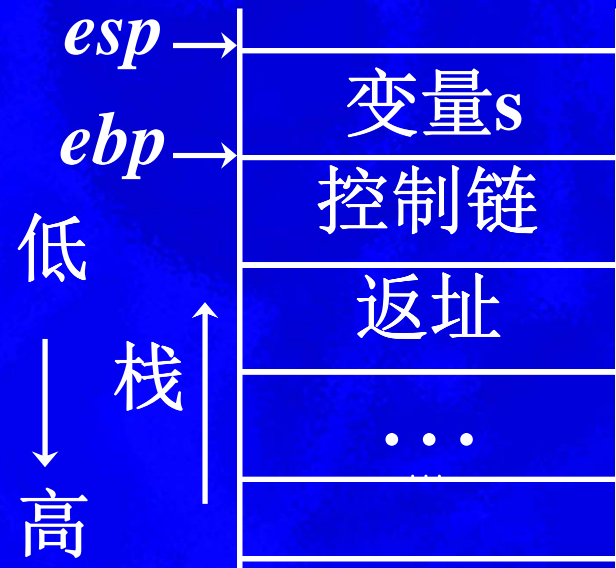
Return from func

Segmentation fault (core dumped)

例　题　7

```
main( )
{  func( );  printf("Return from func\n"); }

func( )
{  char s[4];
   strcpy(s, "12345678");
   printf("%s\n", s);
}
```

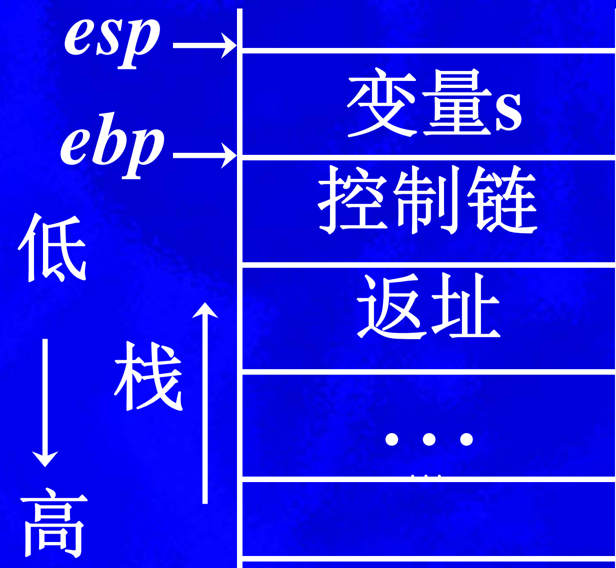


例 题 7

```
main( )
{  func( );  printf("Return from func\n"); }

func( )
{  char s[4];
   strcpy(s, "123456789");
   printf("%s\n", s);
}
```

123456789
Segmentation fault (core dumped)



例　题　8

```
int fact(i)                  | main()
int i;                       | {
{                            |    printf("%d\n", fact(5));
   if(i==0)                  |    printf("%d\n", fact(5,10,15));
        return 1;            |    printf("%d\n", fact(5.0));
   else                      |    printf("%d\n", fact());
        return i*fact(i-1);  |  }
}
```

该程序在x86/Linux机器上的运行结果如下：

```
120
120
1
Segmentation fault (core dumped)
```

例　题　8

请解释下面问题：

– 第2个fact调用：结果为什么没有受参数过多的影响？
– 第3个fact调用：为什么用浮点数5.0作为参数时结果变成1?
– 第4个fact调用：为什么没有提供参数时会出现Segmentation fault？

例　题　8

请解释下面问题：

- 第2个fact调用：结果为什么没有受参数过多的影响？

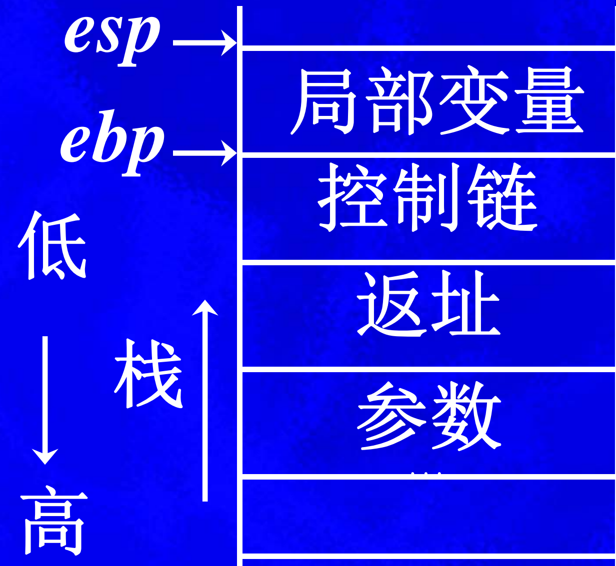
解答：参数表达式逆序计算并进栈，fact能够取到第1个参数

例　题　8

请解释下面问题：

– 第3个fact调用：为什么用浮点数5.0作为参数时结果变成1?

解答：参数5.0转换成双精度数进栈，占8个字节其低地址的4个字节看成整数时正好是0

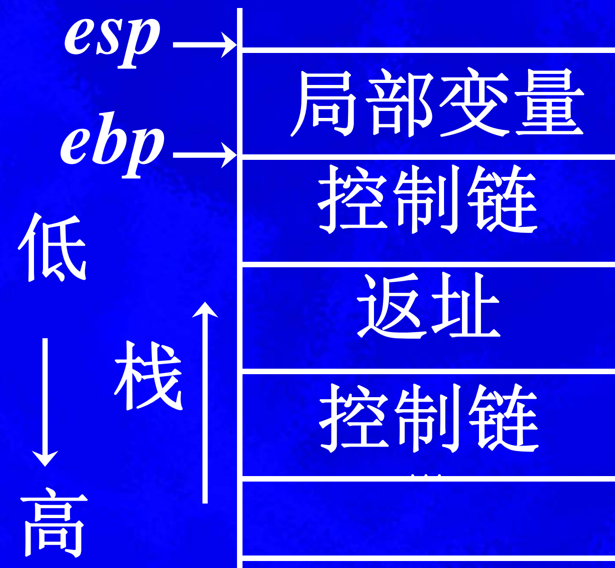


例　题　8

请解释下面问题：

– 第4个fact调用：为什么没有提供参数时会出现Segmentation fault？

解答：由于没有提供参数，而main函数又无局部变量，fact把老ebp（控制链）（main的活动记录中保存的ebp）当成参数，它一定是一个很大的整数，使得活动记录栈溢出



习　　题

- 第一次：**5.4, 5.5**
- 第二次：**5.6, 5.9, 5.12**
- 第三次：**5.15, 5.17, 5.22**