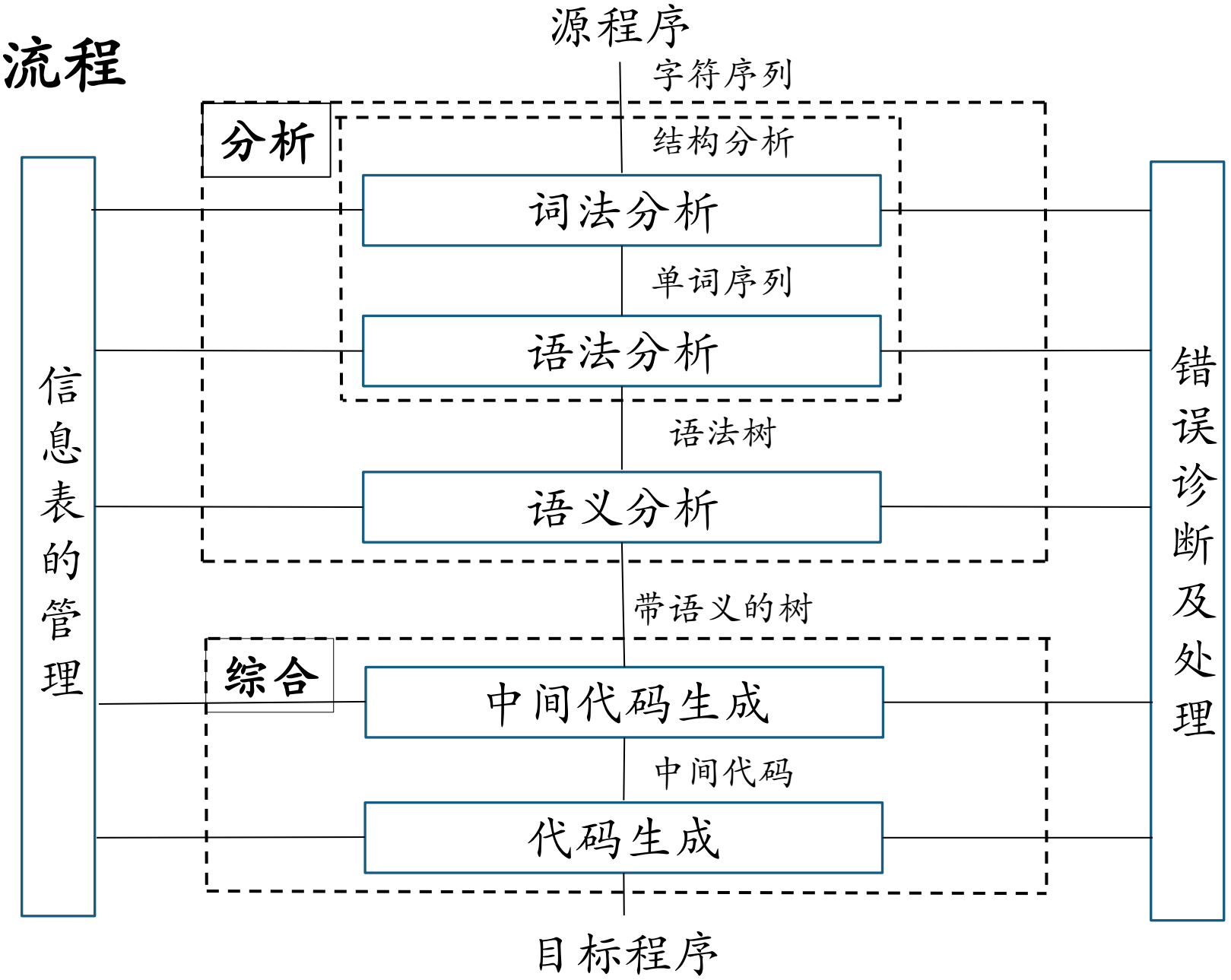
第七章 运行时存储空间组织与管理

信息与软件工程学院
邓伏虎

编译流程


源程序
字符序列
分析
结构分析
词法分析
单词序列
语法分析
语法树
语义分析
带语义的树
综合
中间代码生成
中间代码
代码生成
目标程序
信息表的管理
错误诊断及处理

本章内容

1、活动记录

2、变量的存储分配

3、存储分配模式

4、非局部环境的引用

5、参数传递

运行时环境

程序运行的流程

操作系统 OS

OS为程序分配存储空间

OS将代码复制到所分配的存储空间

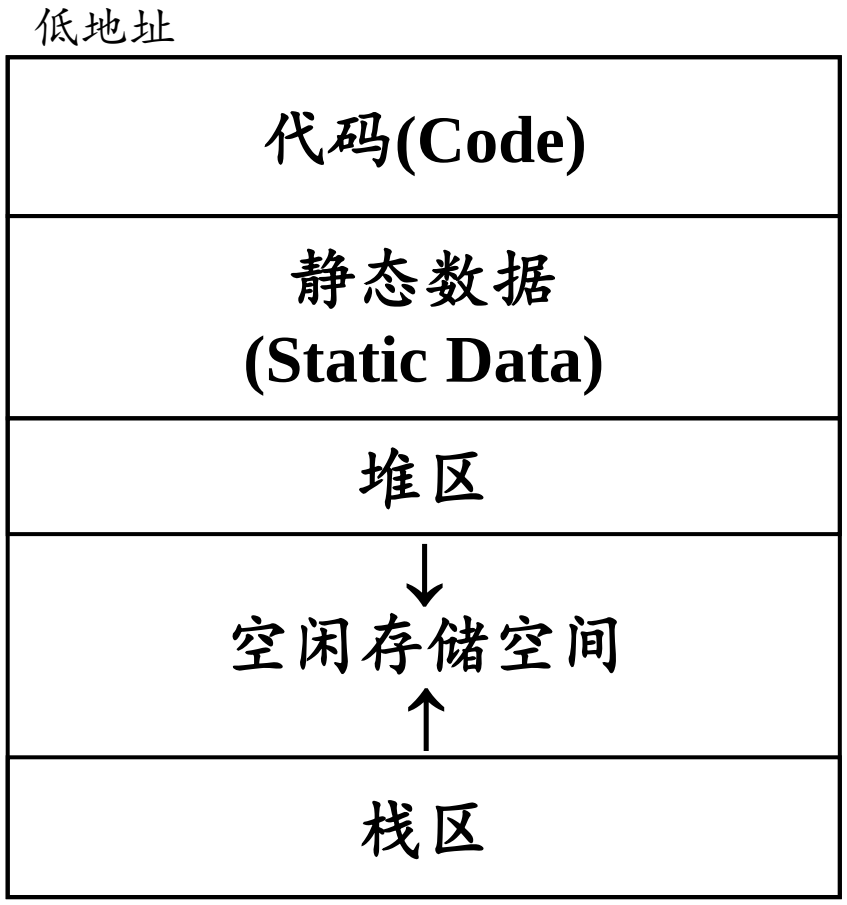
OS跳转到程序的入口地址（main）

。。。

存储的组织与管理

* 名字的地址分配在编译或运行时完成。
* 讨论目标程序运行时的活动与运行环境，主要讨论存储组织与管理，包括：
  - 活动记录的建立与管理。
  - 存储器的组织与存储分配策略。
  - 非局部名称的访问。
  - 源语言的特性等。

运行时存储空间的划分



代码区 存放目标代码；

静态数据区 编译时可确定的占用存储空间大小的数据；

堆区 存放动态数据；

栈区 存放活动记录。

运行时环境

函数的活动

函数的一次执行称为函数的一次活动

函数的活动需要：

可执行代码

存放所需信息的数据结构

通过活动记录进行管理

```
int  g() { return 10;  }
int  f(int x) {
  if (x == 0)
    return g();
  else
    return f(x-1);
}

int main()
{
   f(2);
   return 0;
}
```

函数被多次调用

对应多个活动记录

程序运行对应一个活动记录树

输入不同，活动记录树可能有所不同

活动记录之间“相互嵌套”

可用“栈”来跟踪各活动记录

当函数返回后相应的记录不再被引用

活动记录

讨论一次活动中的数据安排

讨论程序执行过程中所有活动记录的组织方式

活动记录

把过程的一个活动所需要的信息组织成一块连续的存储单元。

过程一次执行所需的信息用一块连续的存储区来管理，称为活动记录。

过程的活动需要可执行代码和存放所需信息的存储空间，后者称为活动记录

一个活动所需要的信息的每个数据项有相同的生存期，因此，组织成一个活动记录是很自然的。

源语言不同，实现方法不同，组成活动记录的域不同。

活动记录的内容

返回地址	返回调用单元的位置
动态链接	指向调用单元最新的活动记录
静态链接	指向非局部变量所在的活动记录
现场保护	存储调用单元当前的状态信息
参数个数	调用单元传递给被调用单元的参数个数
形式单元	存储被调用单元的形式参数
局部变量	
临时变量	

运行栈中的活动记录

活动记录

1. 返回地址(值)
2. 控制链(动态链接)
3. 访问链(静态链接)
4. 现场保护
5. 局部数据
6. 临时变量(中间变量)
7. 实参
8. 形参

运行栈

变量存储位置的确定

在活动记录中，除变量存储区外，其余部分的长度在编译时可以确定，则 元素i的地址可以用下式计算：

$$D + offset(i)$$

其中：

D是活动记录的首地址，offset(i)是i在活动记录中的位移。

变量的类型

根据活动记录首地址以及i的位移大小是在编译时还是在运行时确定，可以将变量划分为四种不同的类型：

静态变量：D和offset(i)在编译时都能够确定

半静态变量：offset(i) 在编译时能够确定下来，D在运行时才能确定下来

半动态变量：D和offset(i)在编译时都不确定，但是在运行时都能确定

动态变量：D和offset(i)在编译和运行时都不能完全确定下来

静态变量

1)D 和 offset(x) 在编译时都能确定下来

因此在编译时，可以为静态变量分配固定的存储空间。

- 每个过程的活动记录的大小及位置
- 活动记录中每一个名字所占用存储空间的大小及位置
- 数据对象的地址可以生成在目标代码中。

程序运行时：

- 过程每次被激活，同一名字都使用相同的存储空间。
- 允许局部名字的值在活动结束后被保留下来。
- 当控制再次进入时，局部名字的值即上次离开时的值。

静态变量

如果一个语言仅支持静态变量，则该语言：

不支持递归调用：因此静态变量在该过程的所有活动中都会分配到同一个地址中。

不支持动态数据结构：因为在运行时不能进行存储分配。

半静态变量

2）offset(x) 在编译时能确定下来，D 在运行时能确定下来

如果一个语言支持半静态变量，则

1）该语言中过程可以同时被多次激活，因为活动记录的长度能够在编译时确定。

2）该语言支持递归调用。

这一类变量的活动记录可以用栈来进行实现，变量支持“栈式存储分配”策略

半动态变量

3）D和offset(x) 在编译时都不确定，D但在运行时都能确定

如动态数组，在编译时，不能确定数组的长度，也不能确定过程被调用的次数。但是在运行时，这些都可以被确定。

半动态变量可以

动态分配在活动记录的“尾部”

即运行栈的“栈顶”

半动态变量的生存期与该活动记录相同

随着函数的返回而消失

```
int f()
{
    int n;
    scanf(“%d”, &n);
    int array[n];
    printf(“%d\n”, sizeof(array));
    return 0;
}
```

动态变量

4）D和offset(x) 在编译和运行时都不能完全确定

动态变量使用另外的数据结构“堆”来存储

需要堆的情况：

局部变量的值在单元活动后还需保留

被调用者的活动生存期比调用者更长

如，含有“文件”，“指针”等数据类型，或允许用户动态申请、释放存贮空间的源语言。

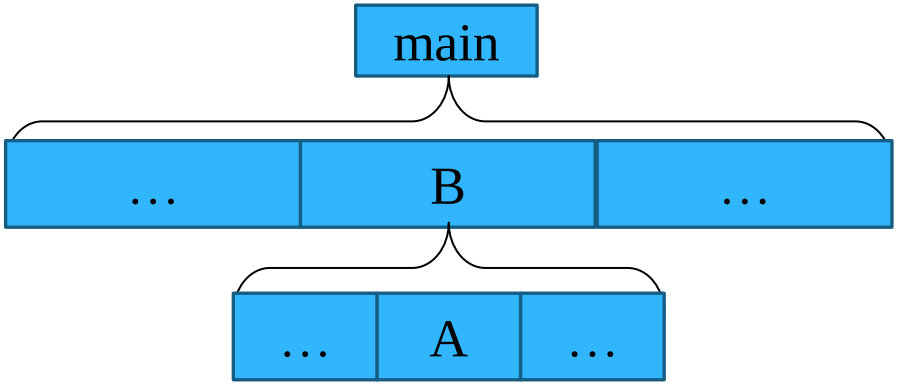
函数调用

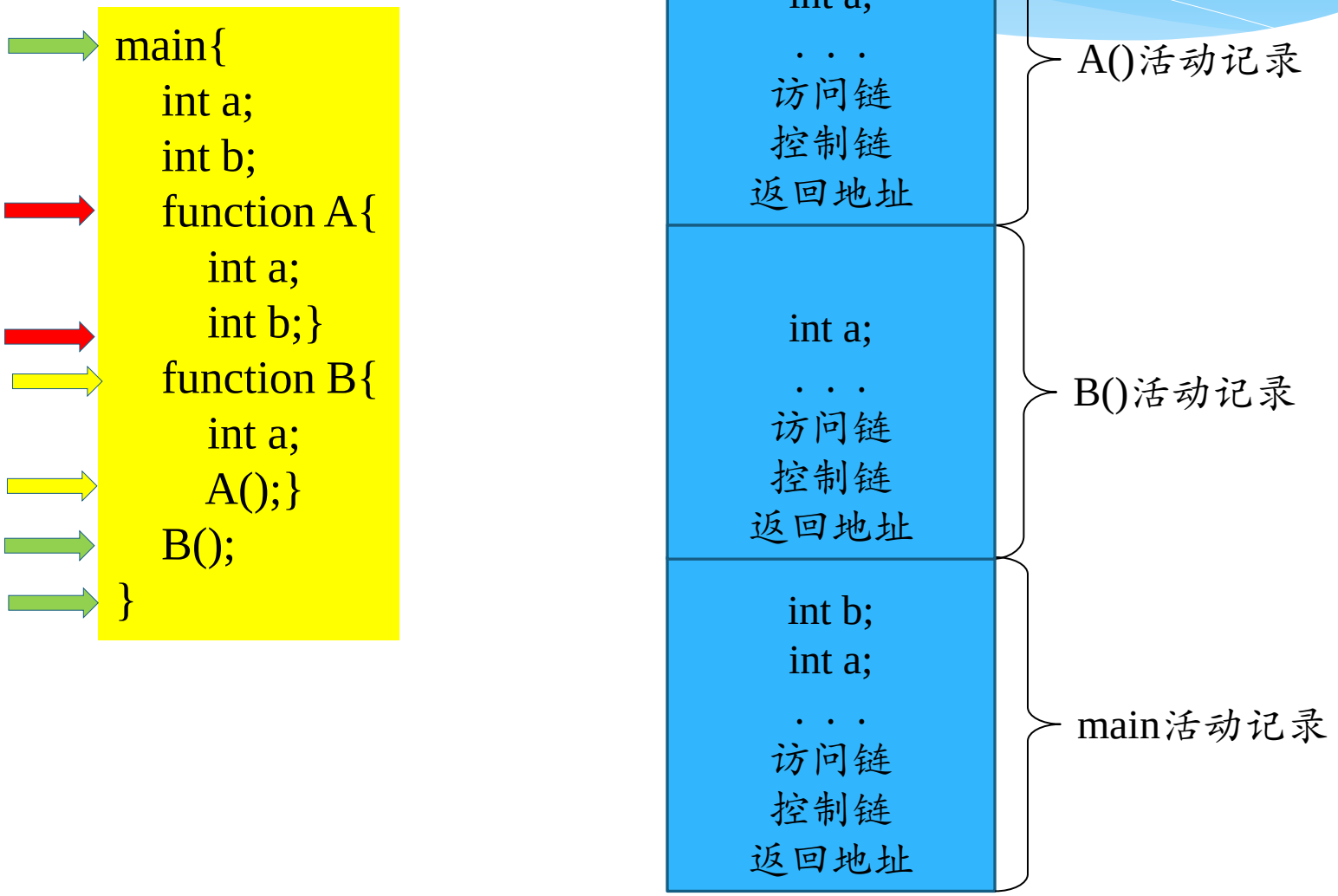
计算机程序通常使用函数来完成相关命令。很多程序设计语言都允许在一个函数的定义中出现对另一个函数的调用，这就出现了函数的嵌套调用。

如：

```
main{
    int a;
    int b;
    function A{
        int a;
        int b;}
    function B{
        int a;
        A();}
    B();}
```

在执行main函数中调用B函数的语句时，计算机转而执行B函数，在B函数中执行到调用A函数的语句时，计算机转而执行A函数，A函数执行完毕返回B函数的断点处继续执行剩余B函数语句，B函数执行完毕时返回main函数的断点处继续执行剩余的main函数语句。




main{
int a;
int b;
function A{
int a;
int b;}
function B{
int a;
A();}
B();
}
int b;
int a;
. . .
访问链
控制链
返回地址
A()活动记录
int a;
. . .
访问链
控制链
返回地址
B()活动记录
int b;
int a;
. . .
访问链
控制链
返回地址
main活动记录

过程调用时活动记录的组织

过程调用时，需要考虑的问题：

1、当调用某个过程时，如何保证能够返回调用单元？

2、调用过程中如何准确设置活动记录的指针？

3、在调用返回时，需要哪些调用单元的指针？

4、当嵌套调用时，如何准确定义变量的作用域？

5、．．．

核心关键：如何保证函数准确调用并准确返回？

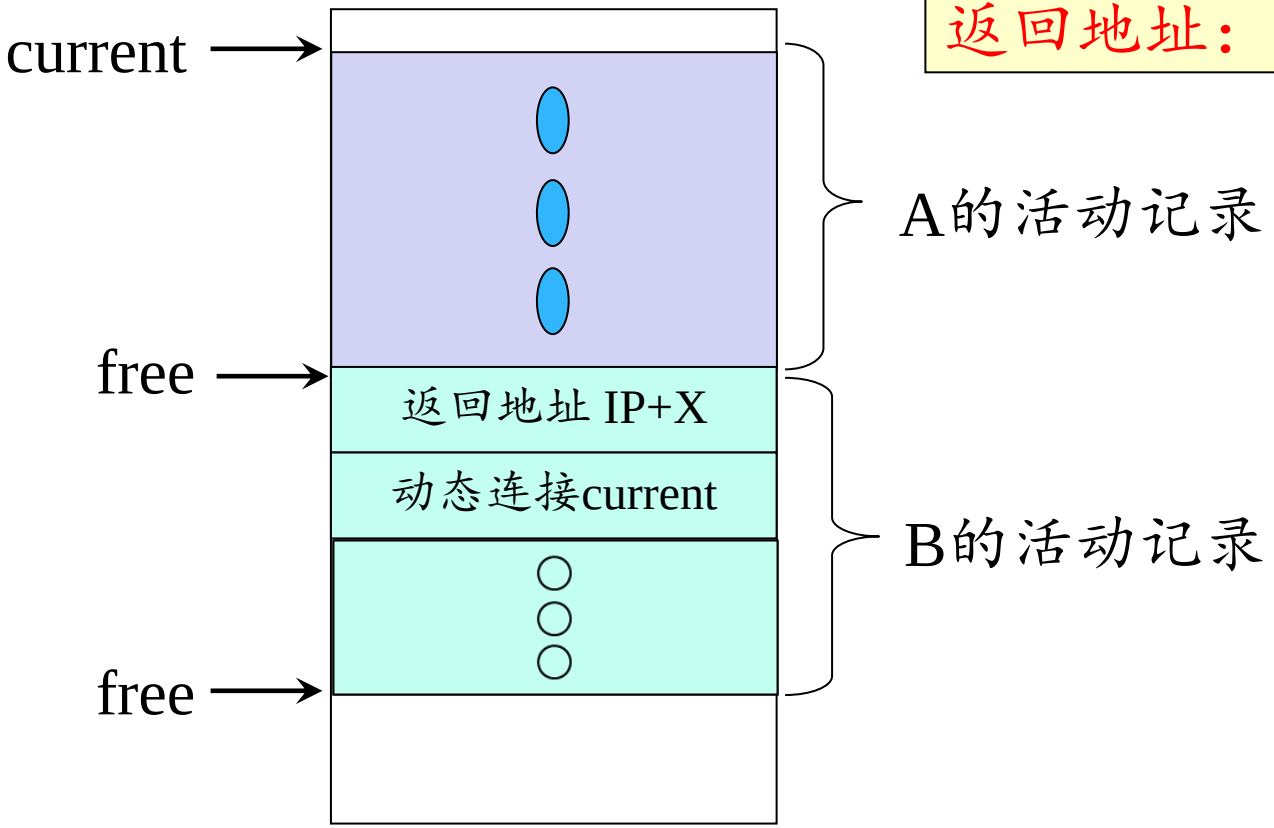
过程调用时活动记录的组织

例子：过程A调用过程B

哪些指针在返回时需要恢复？

返回地址：指令指针IP？

current,free



过程调用时活动记录的组织

处理步骤：（活动记录首地址D，长度为L）

(1) 设置当前栈指针current。表示当前活动记录的开始位置

(2) 指针free表示栈顶下一个可用单元，free = current + L

(3) 局部变量X在活动记录中的位移为 i，则变量的地址current+i，值为D[current+i]

(4) A调用B时，B单元被激活。

在A的栈帧（活动记录）之上建立B的当前实例的活动记录，将current和free绑定于B的活动记录，绑定之前保存current指针的当前值

(5) 从B返回时，释放其活动记录。

Call P 操作的翻译

100 D[free] := 105 （保存返回地址）

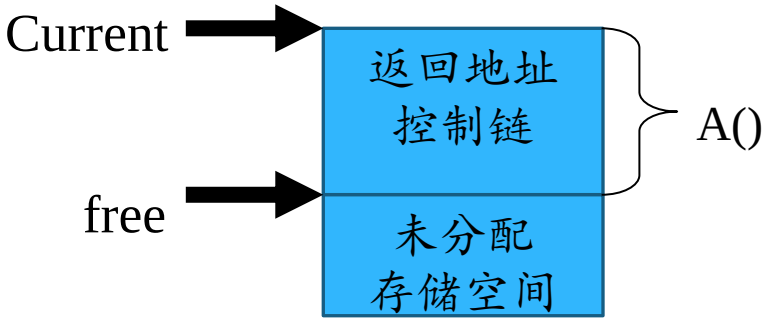
101 D[free + 1] := current （保存current）

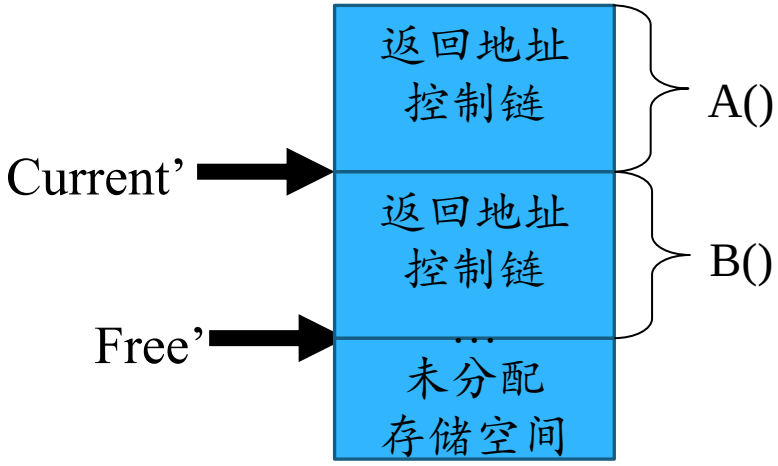
102 current := free （建立新的current）

103 free := free + L （调整free）

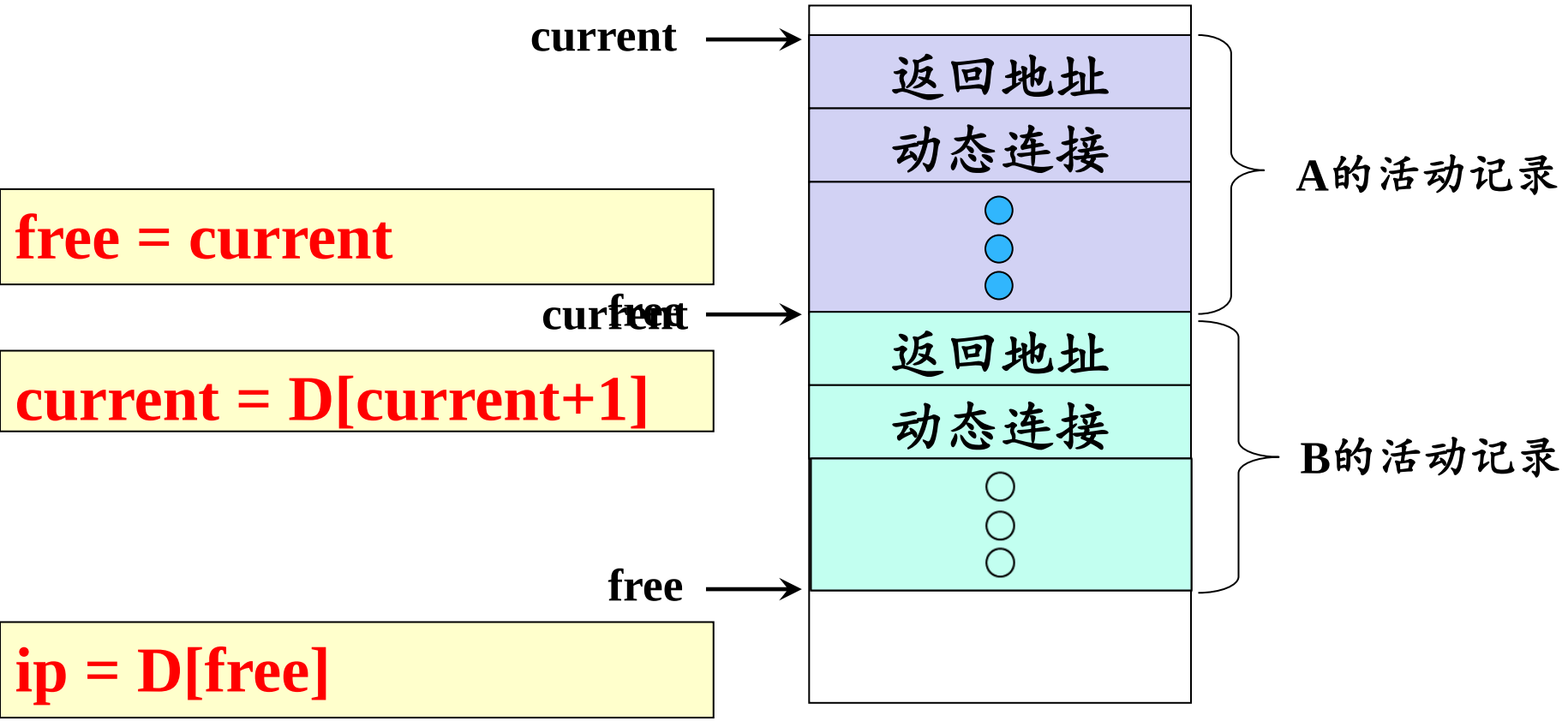
104 ip := P（转移到 P，即被调用过程的第一条代码）

105 … (函数返回后需执行的指令)

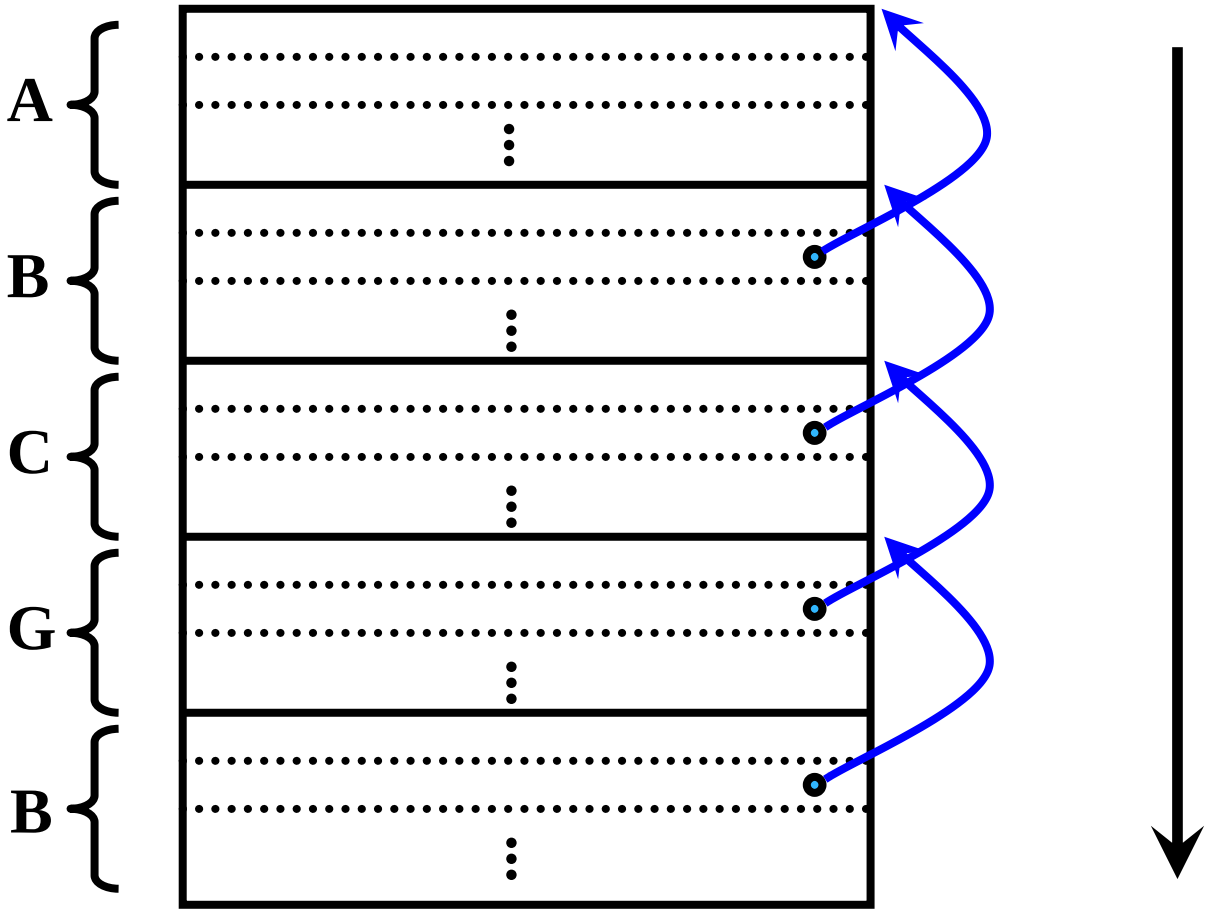




过程调用返回


current
返回地址
动态连接
A的活动记录
free = current
current
返回地址
动态连接
B的活动记录
current = D[current+1]
free
ip = D[free]

例： A call B; B call C; C call G; G call B;



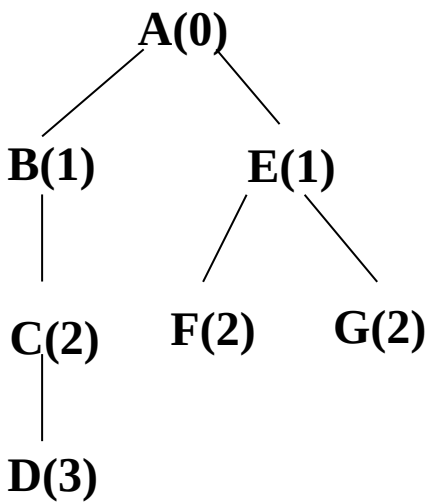
局部变量和非局部变量

局部变量：也称为内部变量。是指在一个函数内部或者复合语句内部定义的变量。局部变量的作用域是定义该变量的函数或复合语句。局部变量的生存期是从函数被电泳时刻算起直到函数返回调用处的时刻结束。对局部变量的引用可以在活动记录中进行查找。

非局部变量：相对于函数内部定义的局部变量而言，包含全局变量、名字空间域变量和静态类成员。对局部变量的引用比较复杂，需要格局作用域规则在合适的位置进行查找。

```
unit A;
  y: int;
  unit B;
    y: int;
    unit C;
      unit D;
        ……
      end D;
      ……
    end C;
  ……
  end B;
  unit E;
    x: int;
    unit F;
      z: int;
      ……
      z:=x+y;
    end F;
    unit G;
      x,y: int;
    end G;
  ……
  end E;
  ……
end A;
```

嵌套层次图



F或G中能否调用B?

F或G中能否调用C?

如何实现引用？

嵌套层次

通常定义主程序的层数为0，所以称主程序为第0层过程。

如果过程Q是在层数为n的的过程R内定义的，并且R是包围Q的最小的过程，则Q的层数被定义为n+1。因此R被称为过程Q的直接外层过程。Q被称为R的内层过程。

一个过程可以引用包围它的任一外层过程中所定义的变量或数组。过程的层数可以静态确定。

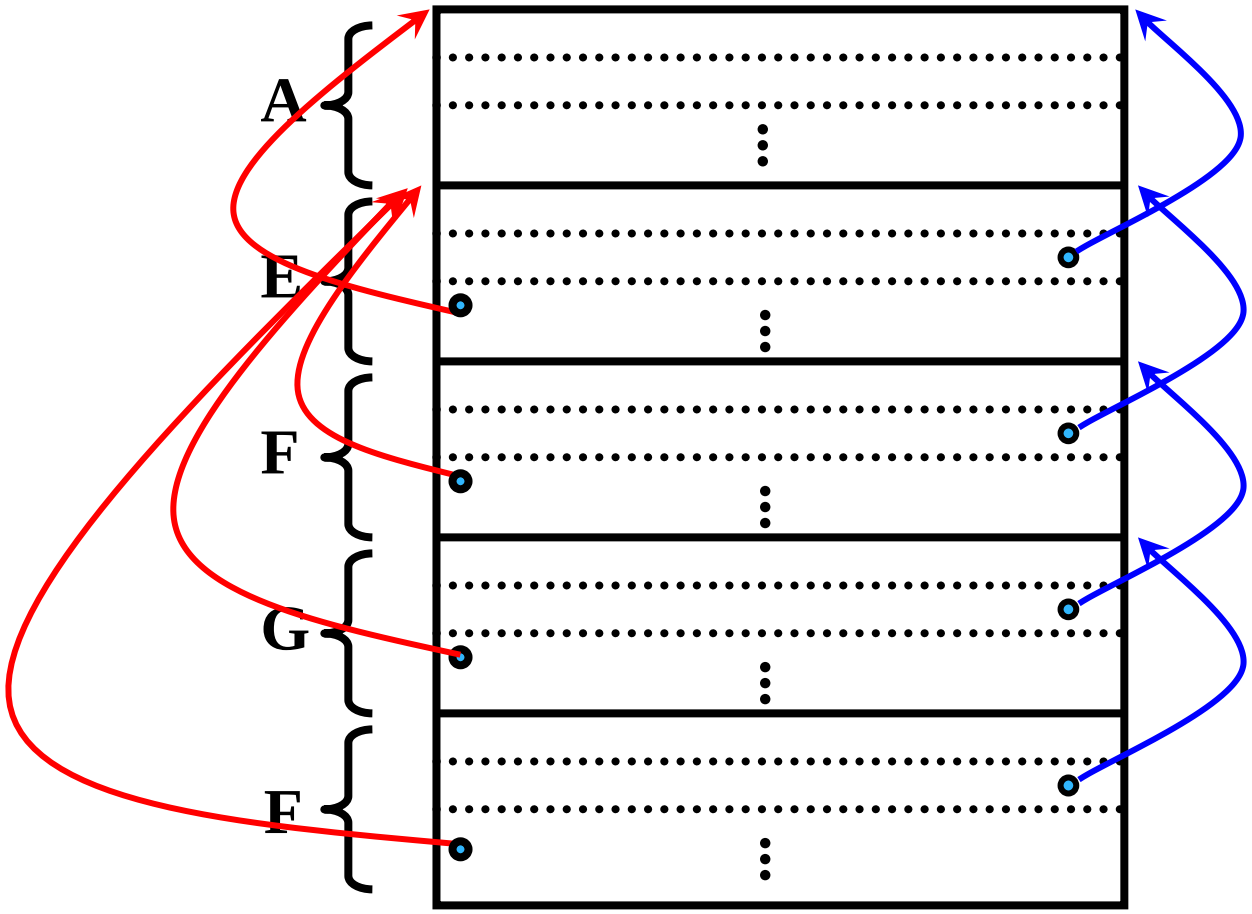
嵌套层次与活动记录

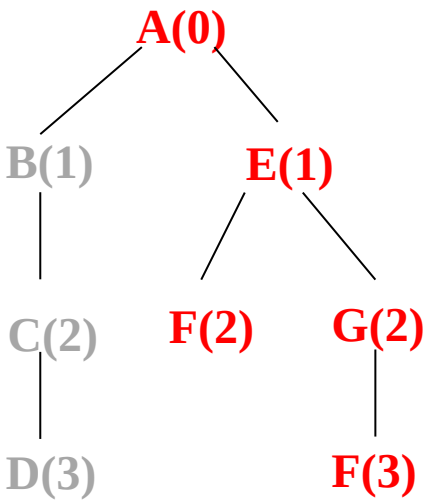
为了能够准确查找非局部变量的位置，在活动记录中采用访问链机制。

访问链：指向直接外层过程的最新活动记录的首地址。

保存位置：活动记录中的第三个存储单元。

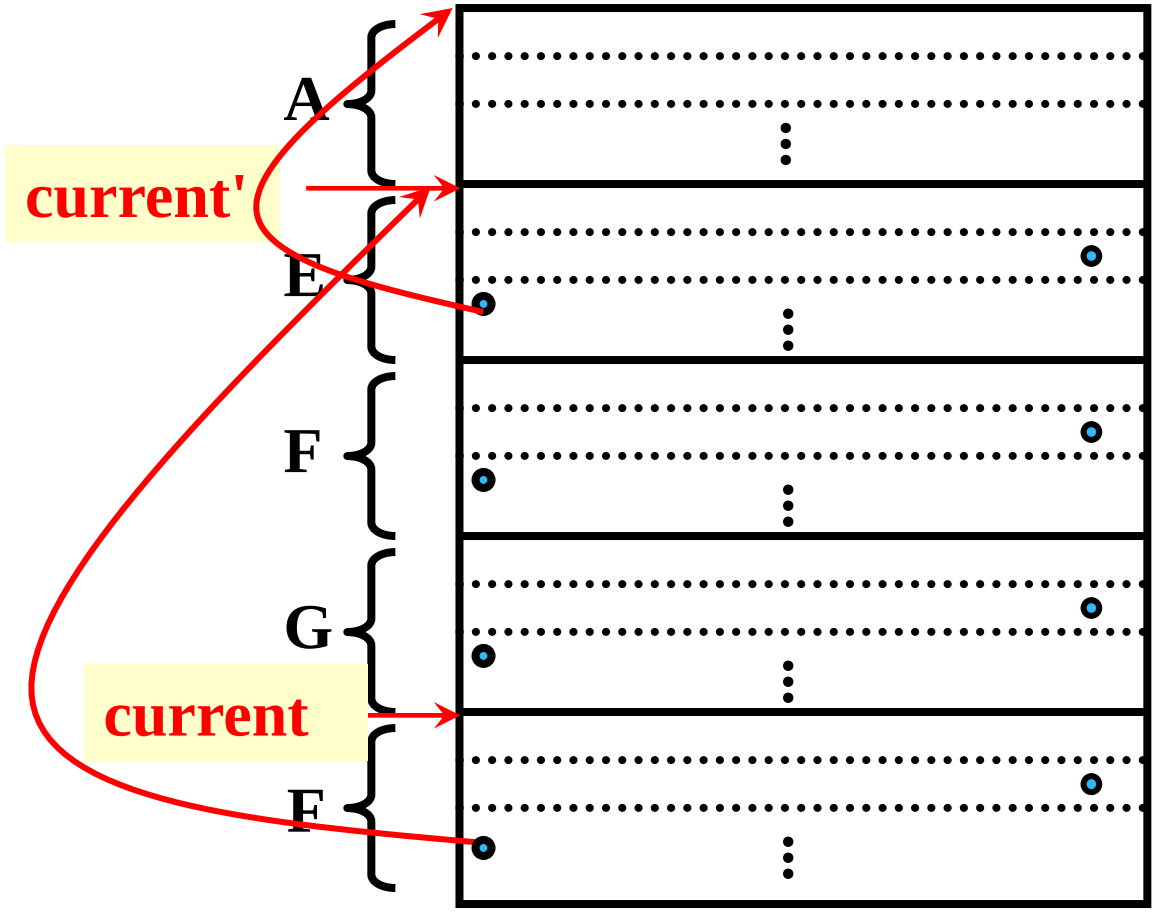
例： A call E; E call F; F call G; G call F;

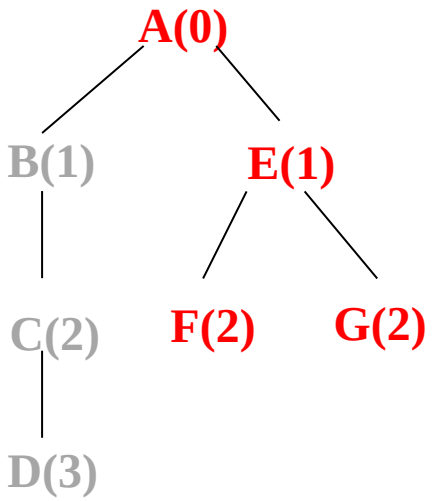




如何计算非局部x变量地址？

非局部变量x的地址的求法





F 引用 单元A中的变量x
其在A中的偏移为offset

嵌套层次：2
D[current+2]为静态连接
D[current' + 2]
D[current' + 2] + offset

非局部变量x的地址的求法

嵌套层次：

D[current+2]为静态链接

若A是B的直接外层，则：

DA ＝ D[current + 2]

若 A是B的第2个外层

DA = D[D[current + 2] + 2]

若A是B的第3个外层

DA = D[D[D[current + 2] ＋ 2] ＋ 2]

可定义函数实现 DA 的获取

非局部变量x的地址的求法

假设单元p中引用了单元t中的变量x

且p, t的深度分别为n_p, n_t

设$d = n_p - n_t$, 定义函数

```
f(d)  {
      if( d == 0 )  then  return  current ;
      else  return  D[ f(d-1) + 2 ] ;   }
```

f(0) = current; f(1) = D[current+2];

f(2) = D[f(1)+2] = D[D[current+2] +2];

假设单元p中引用了单元t中的变量x

且p, t的深度分别为n_p, n_t

设$d = n_p - n_t$, 定义函数

```
f(d) {
    if( d == 0 ) then return current ;
    else return D[ f(d-1) + 2 ] ; }
```

f(0) = current; f(1) = D[current+2];

f(2) = D[f(1)+2] = D[D[current+2] +2];

静态连接建立

1、什么时候建立?

保存位置：在活动记录的第3个存储单元

在返回地址、动态连接之后保存

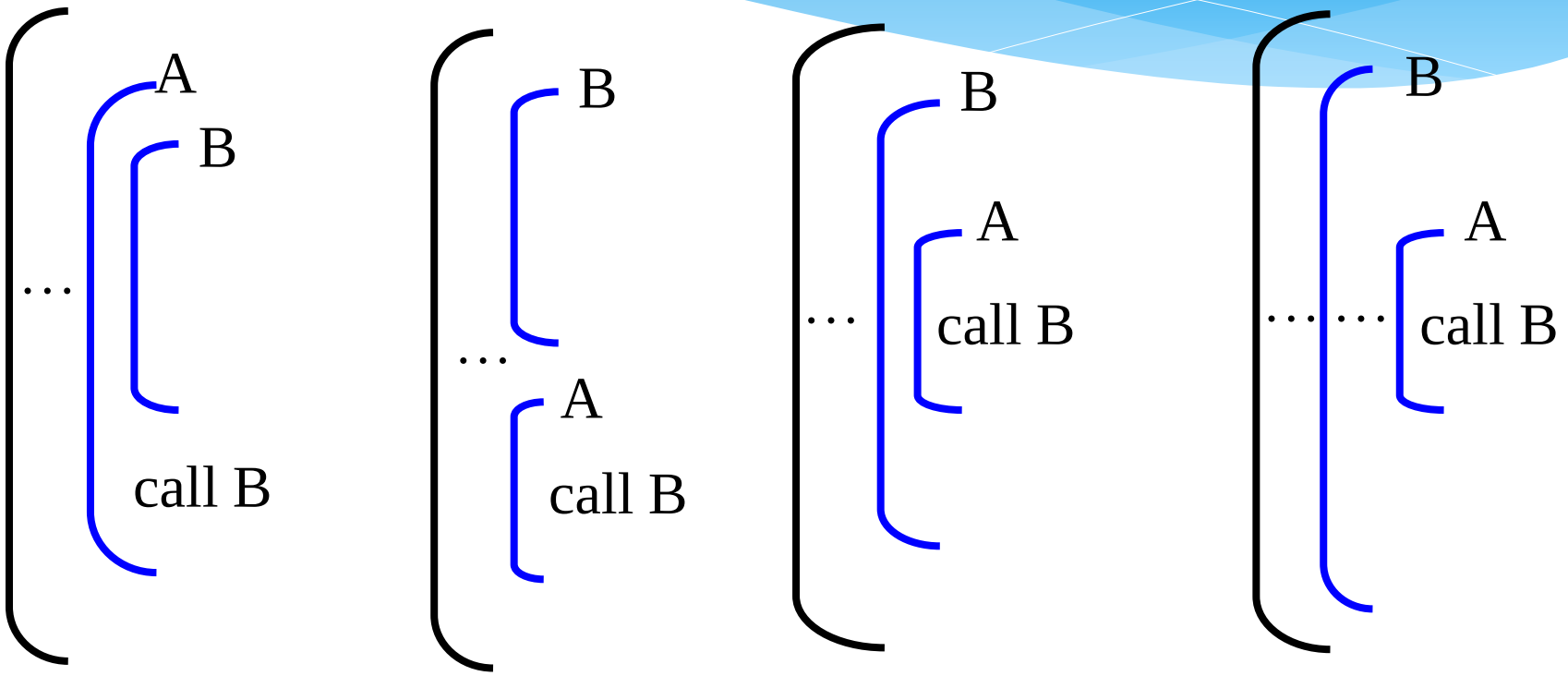
在处理程序单元调用时建立

2、怎么建立?

利用当前的栈帧

考查嵌套结构下单元调用关系

例子：嵌套结果下单元A调用单元B



(1) n_A-n_B = -1 current=f(0)

(2) n_A-n_B = 0 D[current + 2] =f(1)

(3) n_A-n_B = 1 f(2)

(4) n_A-n_B > 1 f(d+1)

静态连接 D[free+2] 的值为 f(d+1)

嵌套定义下 Call P 语句的翻译

1) D[free] := ip + 6 （保存返回地址）

2) D[free + 1] := current （保存current）

3) D[free + 2] := f(d+1) （保存静态连接）

4) current := free （建立新的current）

5) free := free + L （调整free）

6) ip := P（转移到P）

7) … (函数返回后需执行的指令)

静态存储分配

在编译时能够确定目标程序运行时所需的全部数据空间的大小，且在执行期间保持固定，则在编译时就可以组织后程序运行时的全部数据空间，并确定程序中定义的所有名字的存储位置，这种分配策略称为静态存储分配。

特点：绑定是1 vs 1的映射。名字在程序编译时与存储空间结合，每次过程活动时，它的名字映射到同一存储单元。程序运行时不再对存储空间的分配。

静态存储分配适用于没有指针或动态分配、过程不可递归调用的语言。

静态存储分配的性质

(1) 数据对象的大小和它在内存中的位置在编译时就已经确定。

(2) 一个过程的所有活动都使用同一个局部名字绑定，因此，不允许过程被递归调用。

(3) 没有运行时的存储分配机制，因此数据结构不能动态建立。

Fortran语言是一个典型的完全采用静态存储分配策略的程序设计语言。

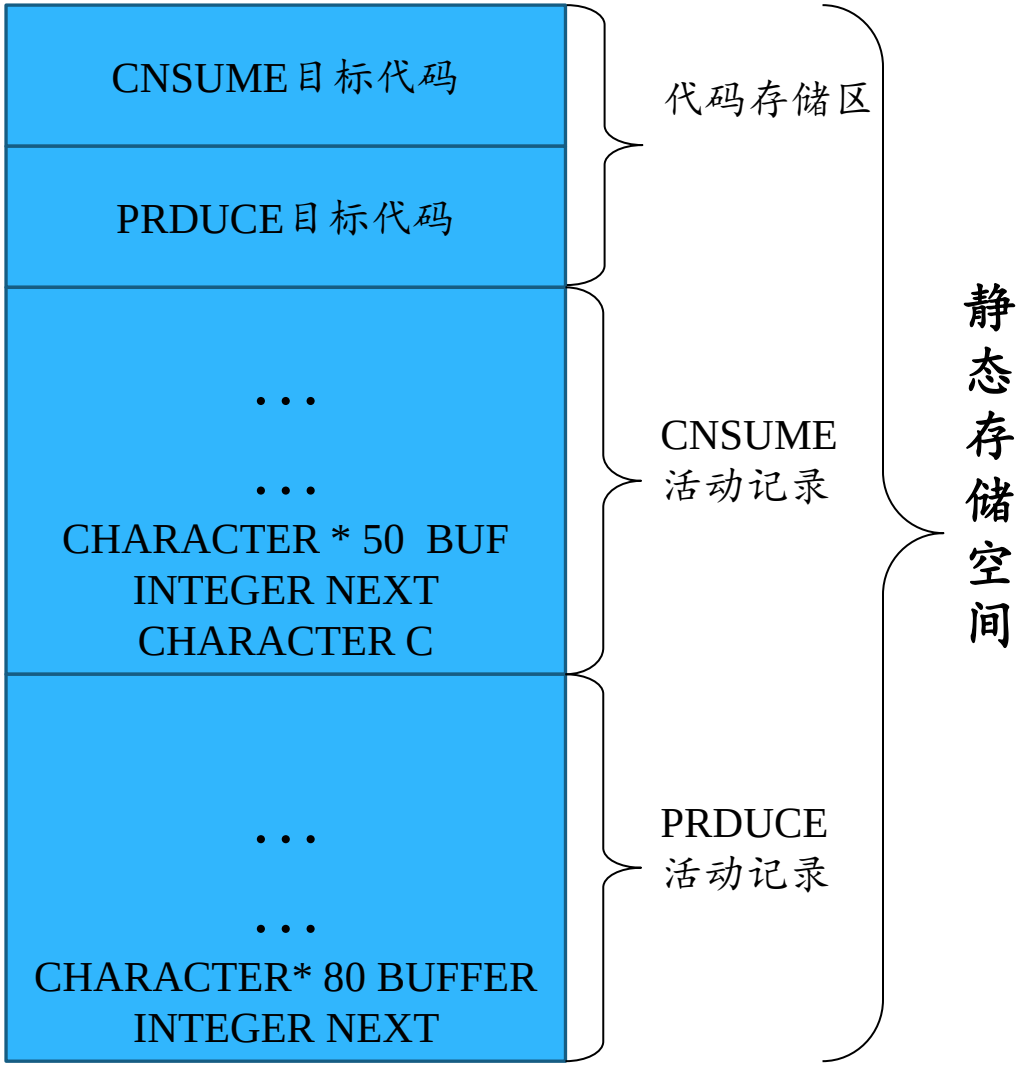
静态存储分配的实现

- 编译程序处理声明语句时，每遇到一个变量名就创建一个符号表条目，填入相应的属性，包括名字、类型、存储地址等。
- 每个变量所需存储空间的大小由其类型确定，并且在编译时是已知的。
- 根据名字出现的先后顺序，连续分配空间。

静态存储分配的实现

```
   PROGRAM  CNSUME
     CHARACTER * 50 BUF
     INTEGER    NEXT
     CHARACTER    C , PRDUCE
     DATA    NEXT / 1 /,  BUF /‘ ’/
6              C = PRDUCE( )
               BUF( NEXT , NEXT ) = C
               NEXT = NEXT + 1
               IF ( C .NE. ‘ ’ ) GOTO 6
               WRITE( * , ‘( A )’ ) BUF
               END

   CHARACTER FUNCTION PRDUCE( )
     CHARACTER * 80 BUFFER
     INTEGER                          NEXT
     SAVE     BUFFER , NEXT
     DATA    NEXT / 81 /
               IF  ( NEXT .GT. 80 ) THEN
                  READ(* , ‘( A )’ ) BUFFER
                  NEXT = 1
               END IF
               PRDUCE = BUFFER( NEXT , NEXT )
               NEXT = NEXT + 1
               END
```

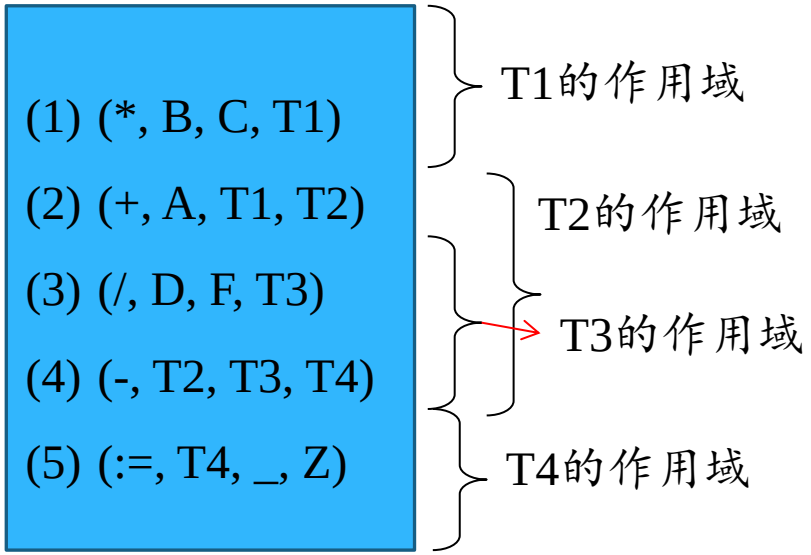


临时变量的地址分配

* 临时变量的作用域
  - 该临时变量第一次被赋值到最后一次被引用之间的全部四元式。

最简单的方式是为每一个临时变量都分配存储空间，但会造成极大的浪费，因此，需要通过临时变量的作用域来确定是否可以共用存储空间。

例如 Z:= A + B * C – D/F的中间代码为：



只需要2个存储单元，而不是4个

动态存储分配

* **动态存储分配方案：**

在编译时不能确定目标程序运行时所需的全部数据空间的大小，而是在目标程序运行时动态确定。

适用于除静态变量以外的其他变量。

主要分为：**栈式存储分配**和**堆式存储分配**

栈式存储分配

* 栈式存储分配方案：

在内存中独立分配一个栈区，按照栈的特点(后入先出)进行变量的存储空间分配。程序运行时，当调用一个过程时，该过程所需的存储空间动态地分配与栈顶，调用结束时，释放所占用的存储空间。

适用于含有可变数组或递归调用过程的程序设计语言。

简单的栈式存储分配和嵌套过程语言的栈式存储分配

简单的栈式存储分配

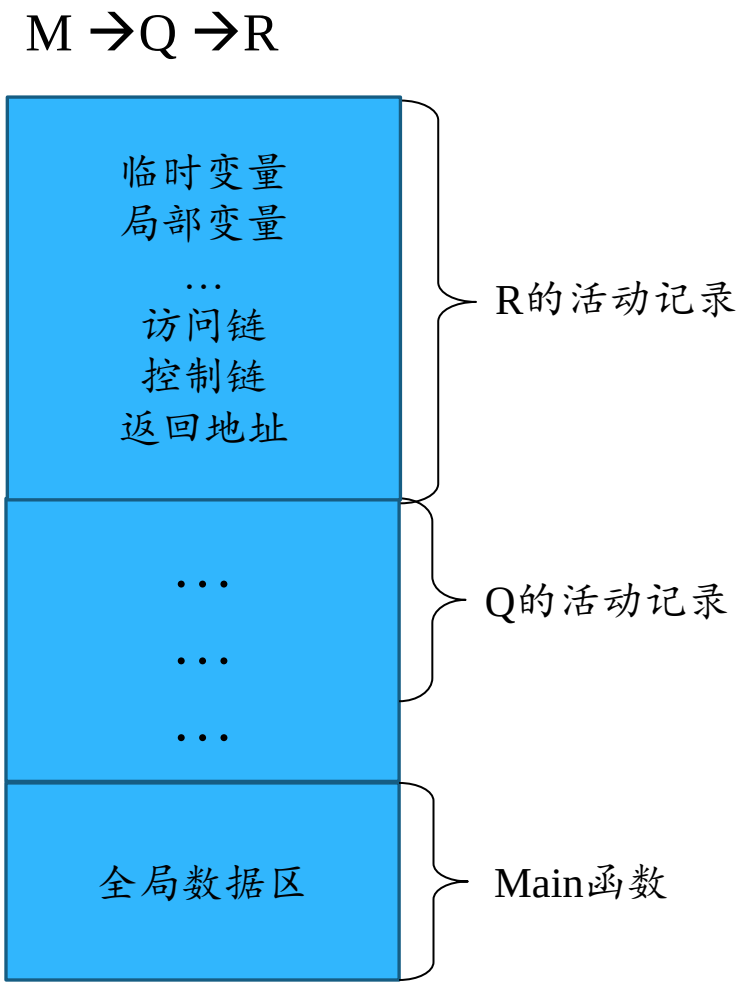
* 适用于没有分程序结构，过程定义不嵌套，允许过程递归调用的语言。
* 例如：

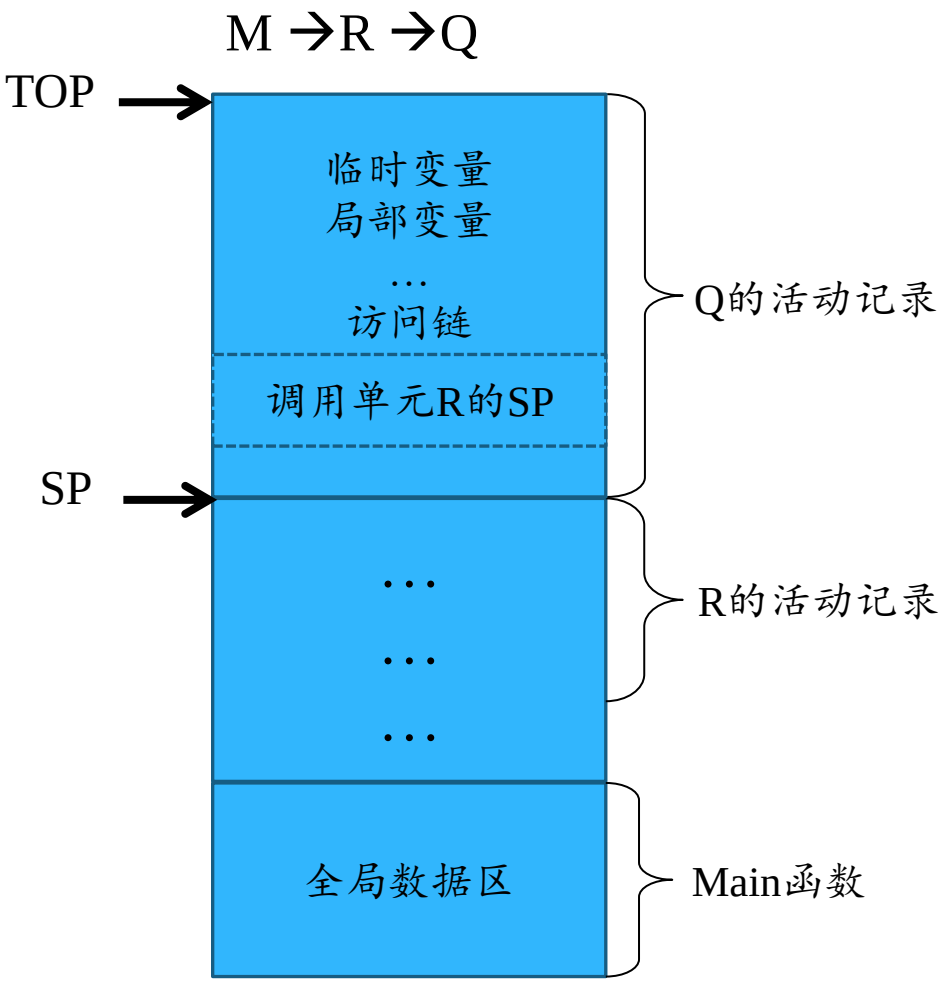
在程序运行时，主程序执行语句中可能存在如下两种情况:

```
Main{
    全局变量的说明
        proc    R
        ……
        end  R;
        proc   Q
        ……
        end  Q;
        主程序执行语句
end  main}
```

M →Q →R 和 M →R→Q

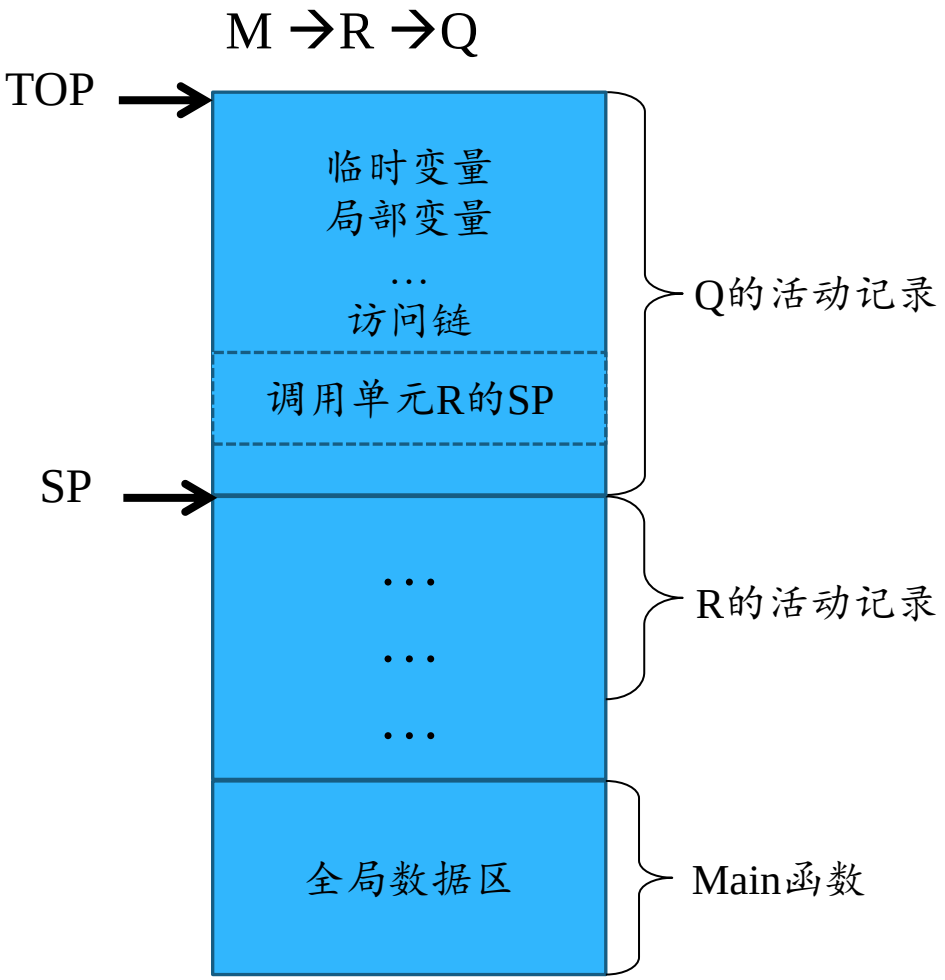
简单的栈式存储分配


M →Q →R
临时变量
局部变量
…
访问链
控制链
返回地址
R的活动记录
…
…
…
Q的活动记录
全局数据区
Main函数


M →R →Q
TOP
临时变量
局部变量
…
访问链
调用单元R的SP
SP
Q的活动记录
…
…
…
R的活动记录
全局数据区
Main函数

简单的栈式存储分配

Q中局部变量的地址
= SP + 相对地址



嵌套过程语言的栈式存储分配

简单栈式存储分配适用于过程定义不嵌套的程序设计语言。如果程序设计语言允许过程定义嵌套，则需要一种复杂的栈式存储分配方案。

过程定义嵌套的特点：一个过程可以引用包围它的任何一个外层过程所定义的标识符(如变量、数组或者过程等)。在运行过程中主要体现在一个过程可以引用它的任一外层过程中的最新活动记录中的某些数据(如某个变量)。

关键问题：如何对非局部变量进行引用？

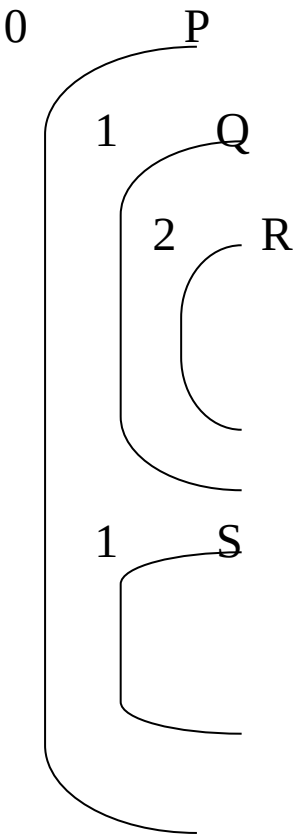
关键技术：用静态链跟踪每个外层过程的最新活动记录

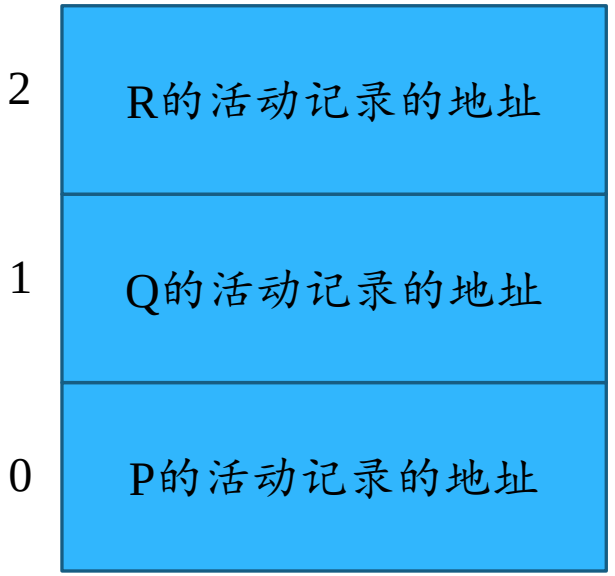
用display表来存放所有嵌套的外层过程的现行活动记录的基地址

Display表

Display表，也叫嵌套层次显示表。具有栈的结构，用于存放所有嵌套的外层过程的现行活动记录的基地址。程序运行时，每进入一个过程，就会在建立它的活动记录区的同时建立一张display表。如果过程的嵌套层数为i，则该表将包含i+1个单元。

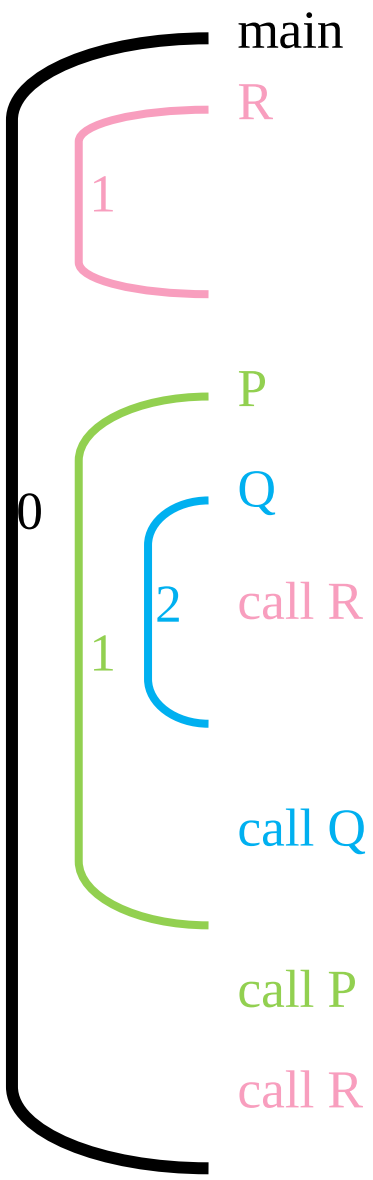
例如：



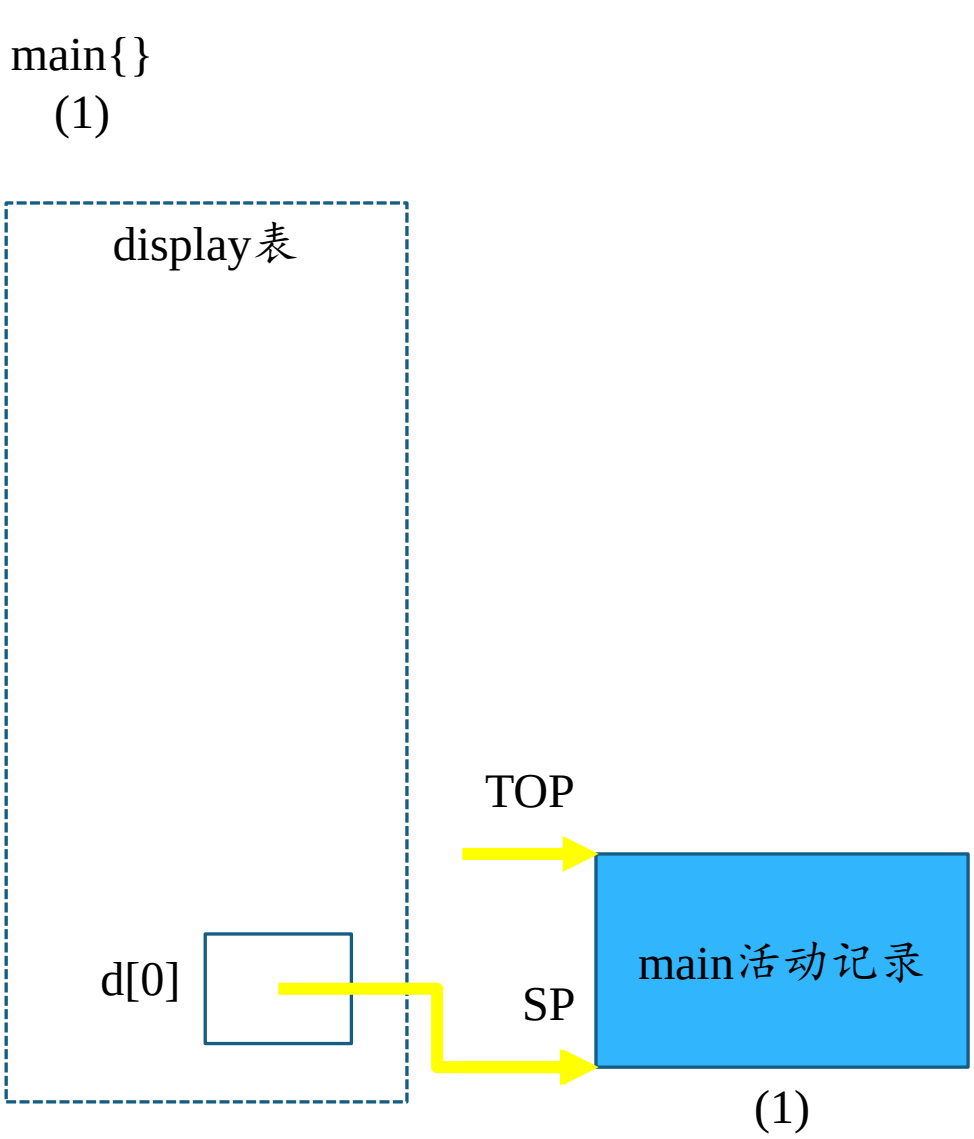


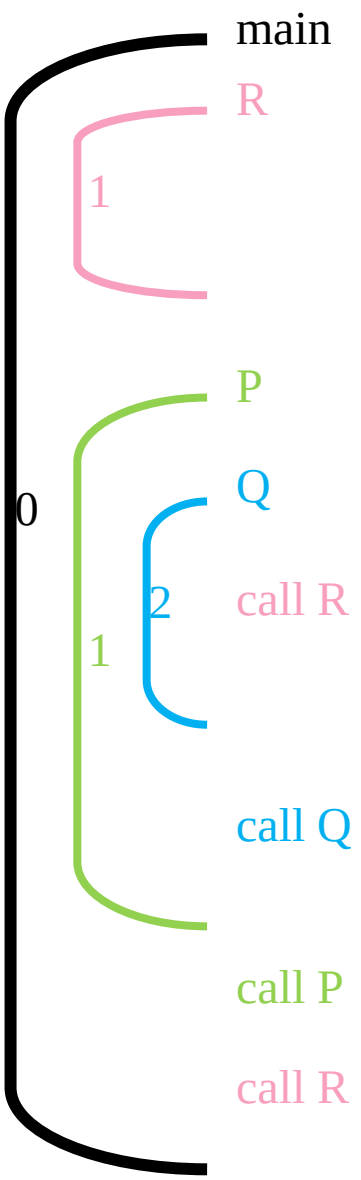
```
program main(i,0) ;
    ……
    proc R(c,d) ;
    ……
    end /*R*/
    proc P (a);
        ……
        proc Q (b);
            ……
            R(x,y);
        end /* Q*/
        ……
        Q(z);
    end /*P*/
        ……
    P(W);
        ……
    R(U,V);
        ……
end /* main*/
```

程序结构图

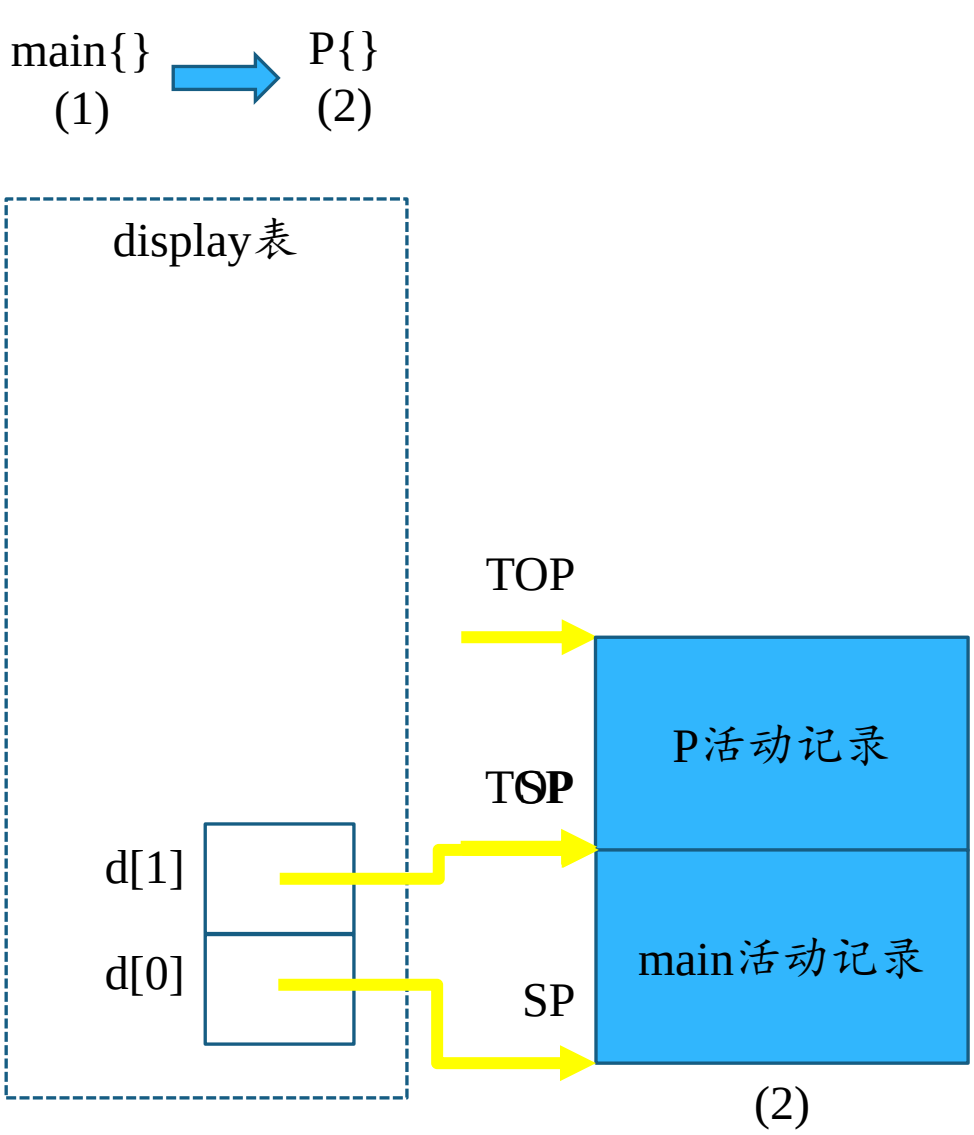


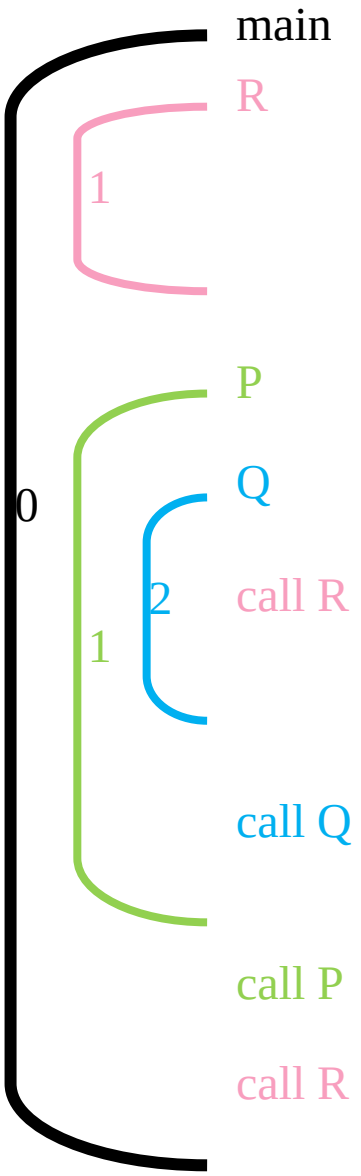
Display表与活动记录



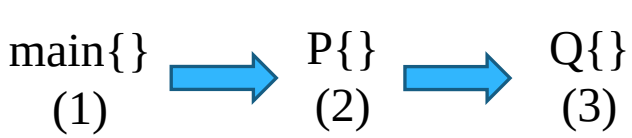


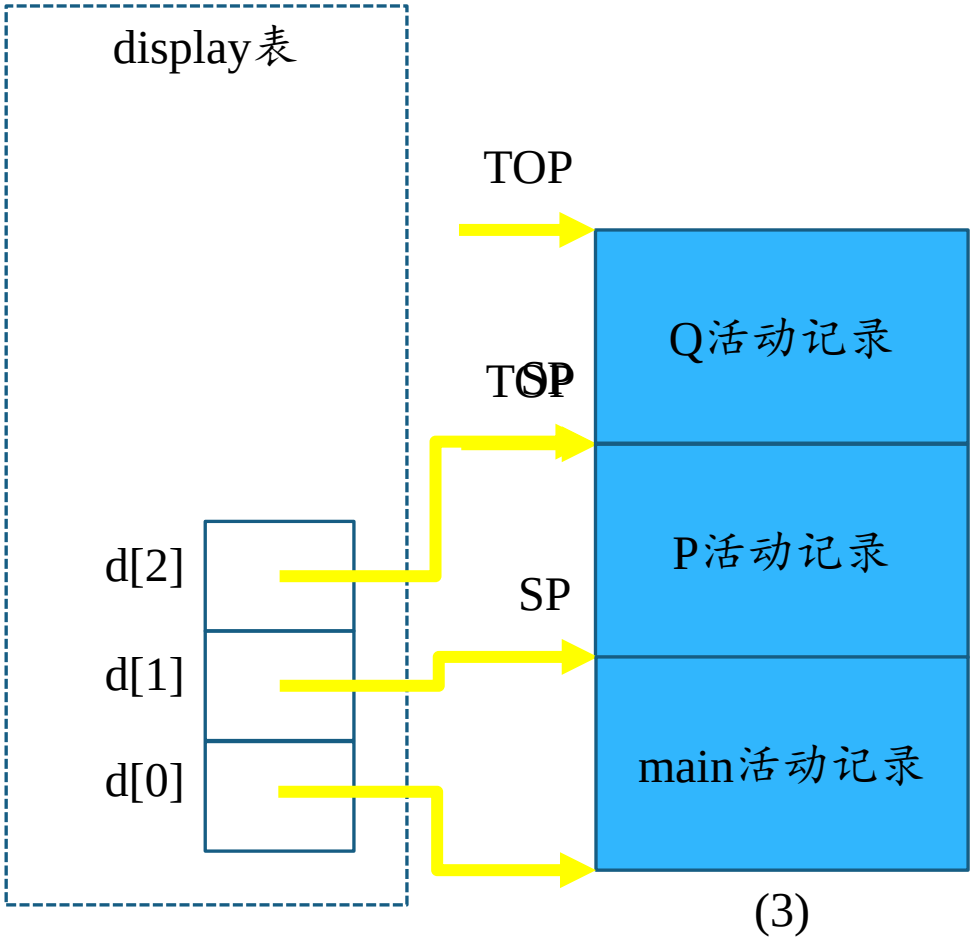
Display表与活动记录

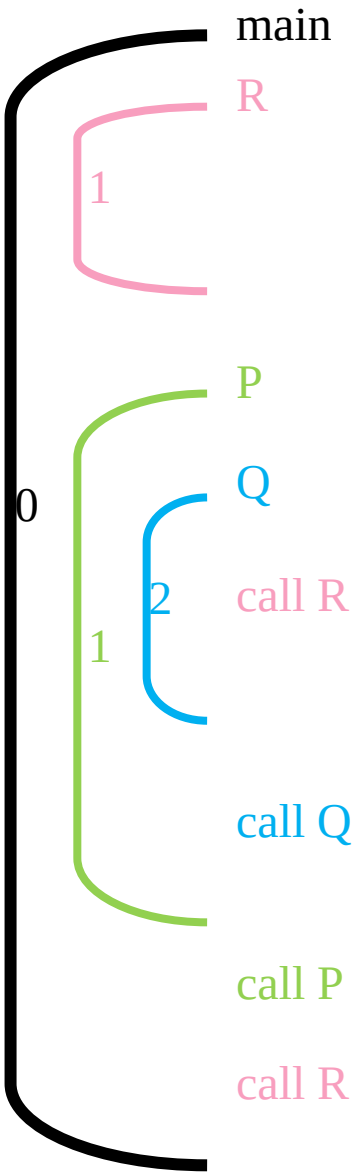




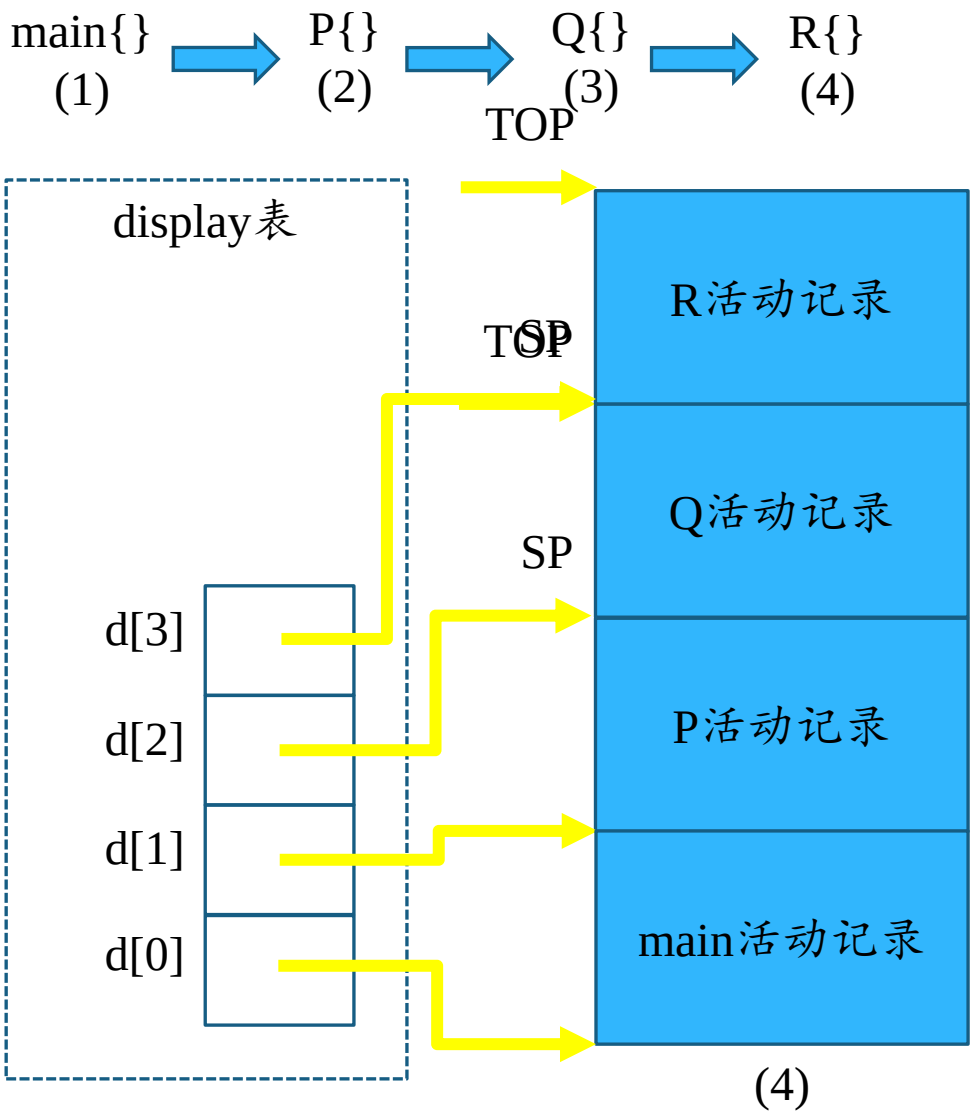
Display表与活动记录

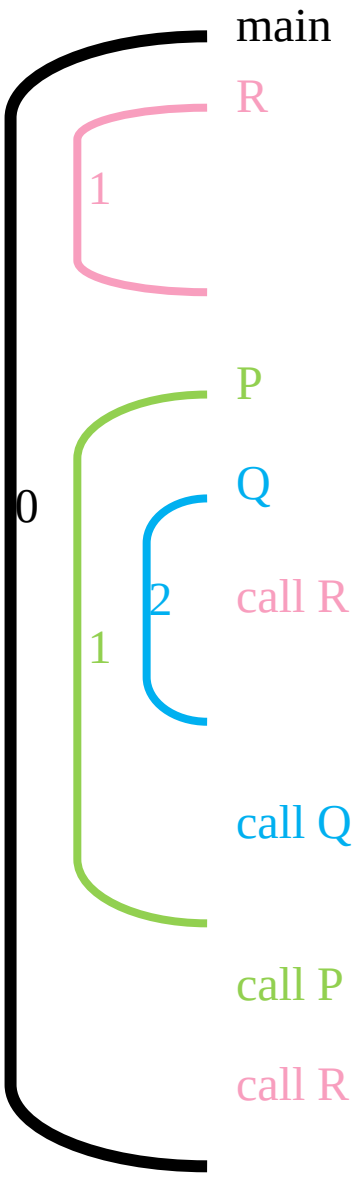






Display表与活动记录





Display表的应用

从上面的例子可以看出，通过Display表的一个域即可确定任意外层活动记录的指针，沿着该指针可找到处于外层活动记录中的非局部变量。

而静态链中存储的是直接外层函数的首地址，如果需要查找非直接外层过程中的变量，则需要一层一层地返回查找。

堆式存储分配

在编译时，如果能够确定一个程序在运行时所需的存储空间的大小，且在执行期间保持固定，则可采用静态存储分配方案。

如果某些存储需求在编译时不能确定，但是在程序执行期间，在程序的入口处可以确定时，可采用栈式存储分配方案。

对于一个允许用户自由地申请和释放数据空间的程序设计语言来说，空间的使用不一定遵循先申请后释放或者后申请先释放的栈式原则，上述两种分配方案则不再适用，需要一种更加灵活的存储分配方案。

这就是堆式存储分配方案。

堆

所谓堆，是指一片连续的、足够大的专用全局存储空间，其存储空间的分配与释放不再遵循后进先出的规则，在用户需要时就分配一块存储区，当某个存储区不再使用时就释放给堆存储空间。

比如C语言中的malloc函数以及free函数被用来申请和释放空间。

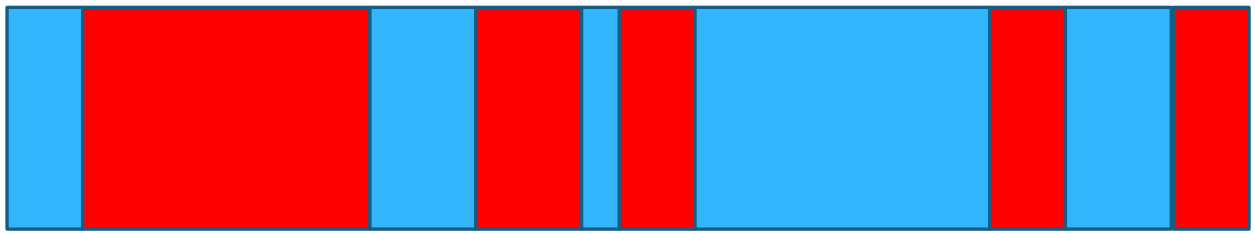
由于堆中存储区的申请和释放时间并不一致，虽然程序刚开始时这一块空间是连续的，但程序运行一段时间后，则会被分割成许多大小不一的存储空间，有些正在使用，有些则已经释放。

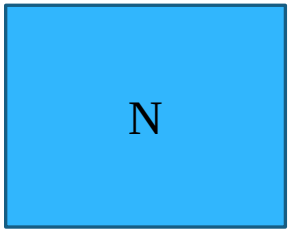
堆式存储分配

在对堆进行管理时，需要两个操作：分配操作和释放操作

当程序需要对一块长度为N的空间时，应该如何在堆中进行分配?

- 堆中存在比N大的空闲存储区
- 堆中没有比N大的空闲存储区，但总的空闲存储区之和大于N
- 堆中没有比N大的空闲存储区，且总的空闲存储区之和小于N





堆式存储分配技术

* 定长块方式:

将堆空间分成长度相同的若干块，每块包含一个链域，按照相邻的顺序将所有的块链成一个链表，分配时总是优先分配链表头部的空闲块。释放空间时，将释放的空闲块插入到链表的头部。

- 优点：剩余空闲空间相对连续。
- 缺点：程序所需的空间大小不一致，可能存在存储空间浪费。

堆式存储分配技术

* 变长块方式:

根据程序需要分配不同长度的存储块。分配时分配所需大小的存储块。释放空间时，如果释放的空闲块能够与现有空闲块合并，则合并为一个空闲块，如果不能合并，则将空闲块链成一个链表。再次分配时，从空闲链表中找出满足需要的空闲块。

- 优点：按需取用，浪费较少
- 缺点：链表管理操作复杂。

堆式存储分配技术

* **存储回收：**

程序运行时，有可能出现程序对存储块的申请得不到满足的情形。比如在程序运行时经常出现的“存储空间不足”。为了程序能够运行下去，有时候可以暂时挂起程序，使系统对存储空间进行回收，然后再恢复程序的运行。

过程调用

过程：是函数、方法、过程的统称。

当过程名出现在可执行语句里的时候，称该过程在这一点被调用。过程调用将导致过程体被执行。

形参：出现在过程定义中的标识符

实参：出现在过程调用中的标识符、常数及表达式。

参数传递

调用过程与被调用过程之间的信息传递方式有两种：一种是通过全局变量传递，另一种是通过参数传递。

参数传递机制：传值调用、得结果调用、传地址调用、传名调用

传值调用

传值调用是一种最简单的参数传递方式。

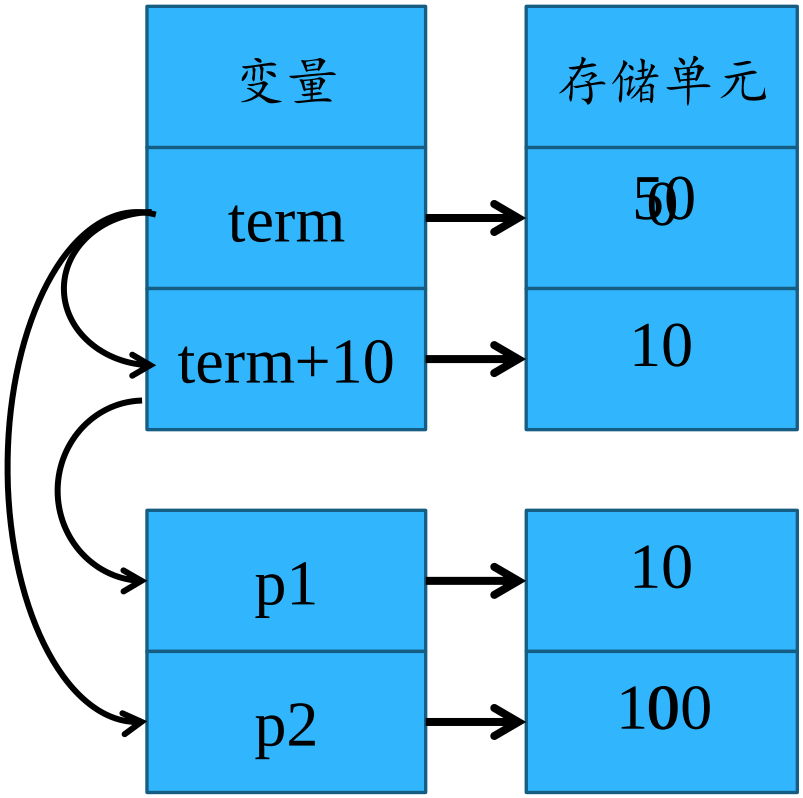
调用过程把实际参数的值计算出来并存放在一个被调用过程能够获取的存储空间中。被调用过程开始工作时，首先把这些值复制进自己的形式单元中，然后就像使用局部变量一样使用这些形式单元。如果实际参数不是指针， 则被调用过程无法改变实参的值。

这种传递机制时函数式语言中唯一的参数传递机制

- C++、Pascal语言的内置机制
- C语言、Java语言唯一的参数传递机制

在被调用过程中，参数和局部变量一样可以被赋值，但计算结果不影响过程体之外的变量的值

传值调用-样例




```
1. function outer_func(){
2.     var tem = 0;
3.     function inner_func(p1,
p2){
4.         console.log(p1);
5.         tem = tem + 50;
6.         p2 = p2 + 100;
7.         console.log(p1);
8.         console.log(p2);
9.     }

10.    inner_func(tem + 10,
tem);
11.    console.log(tem);
12. }
```

p1 = 10

p1 = 10

p2 = 100

tem = 50

传值调用-参数是指针类型时

传值调用并不意味着参数的使用一定不会影响过程体外变量的值。

如果参数的类型为指针，参数的值就是一个地址。

- 通过它可以改变过程体外部的内存值。如，有C语言函数

```
void init_ptr(int *p) { *p=3; }
```

- 对参数p的直接赋值不会改变过程体外的实参的值。如：

```
void init_ptr(int *p) { p=(int *) malloc(sizeof(int)); }
```

在一些程序设计语言中，某些值是隐式指针

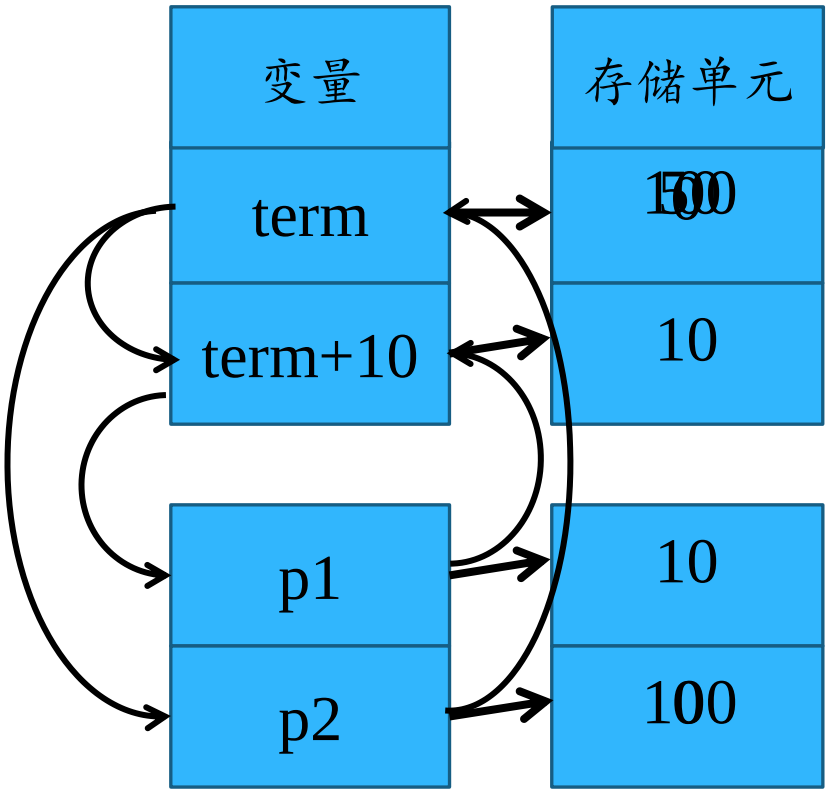
- 如C语言中的数组是隐式指针（指向数组空间的起始位置）
- 可以使用数组参数来改变存储在数组中的值。如：

```
void init_array_0(int p[]) { p[0]=0;}
```

得结果

所谓得结果参数传递机制，是指每个形式参数对应两个单元，第1个单元存放实参的地址，第2个单元存放实参的值。在被调用单元中，对形参的任何引用或赋值都看成是对它的第2个单元的直接访问，但在过程工作完成返回前必须把第2个单元的内容存放到第1个单元所值的那个实参单元之中。

传值得结果-样例




```
1. function outer_func(){
2.     var tem = 0;
3.     function inner_func(p1,
p2){
4.         console.log(p1);
5.         tem = tem + 50;
6.         p2 = p2 + 100;
7.         console.log(p1);
8.         console.log(p2);
9.     }

10.    inner_func(tem + 10,
tem);
11.    console.log(tem);
12. }
```

p1 = 10

p1 = 10

p2 = 100

term = 100

得结果与传值的区别

得结果传递机制与传值传递机制在前半部分的操作是一致的，区别在于从被调用单元返回时，得结果传递机制需要把形参当前的值传递给实参。这种方式能够改变调用单元中实参的值。

在被调用过程中复制和使用实参的值，然后当从被调用过程退出时，将形参的最终值复制回实参的地址。因此，这个方法有时也被称为复制进，复制出，或复制存储，复制恢复。

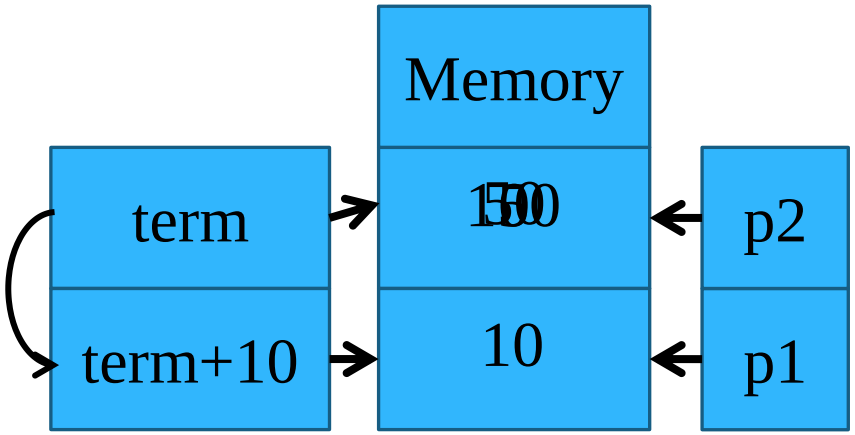
按引用传递

按引用传递，也称为传地址。是指把实际参数的地址传递给相应的形式参数的一种参数传递方式。

当调用一个过程时，调用单元必须预先把实际参数的地址传递到一个被调用单元可以访问的地方。如果实际参数是一个变量，则直接传递它的地址；如果实际参数是一个常数或者是一个表达式，则先计算出它的值并存放在一个临时单元中，再传递该临时单元的地址。

被调用单元激活后，首先把实参地址复制到相应的形式单元中，过程体对形参的任何引用或者赋值都被处理程对形式单元你的间接访问。

按引用传递



p1 = 10

p1 = 10

p2 = 150

term = 150

```
1. function outer_func(){
2.     var tem = 0;
3.     function inner_func(p1,
p2){
4.         console.log(p1);
5.         tem = tem + 50;
6.         p2 = p2 + 100;
7.         console.log(p1);
8.         console.log(p2);
9.     }

10.    inner_func(tem + 10,
tem);
11.    console.log(tem);
12. }
```

按引用传递

原则上要求：实参必须是已经分配了存储空间的变量。

调用过程把实参存储单元的地址（即一个指向实参存储单元的指针）传递给被调用过程的相应形参。

被调用过程执行时，通过形参间接地引用实参。
可以把形参看成是实参的别名，对形参的任何引用都是对相应实参的引用。

Fortran语言唯一的参数传递机制。

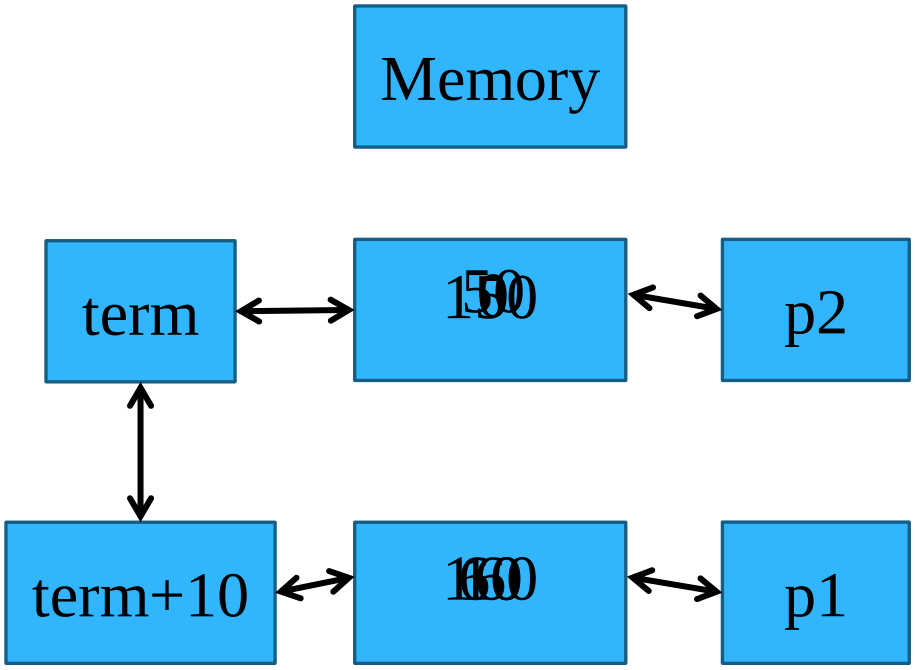
Pascal语言，通过在形参前加关键字var来指定采用按引用调用机制。

参数传递-按名传递

按名传递是一种特殊的形参实参相结合的方式，其本质是替换。

过程调用的作用相当于把被调用段的过程体抄写到调用出现的位置，再对其中任一出现的形式参数进行文字替换，替换为相应的实际参数。

参数传递-按名传递



p1 = 10

p1 = 160

p2 = 150

term = 150

```
1. function outer_func(){
2.     var tem = 0;
3.     function inner_func(p1,
p2){
4.         console.log(p1);
5.         tem = tem + 50;
6.         p2 = p2 + 100;
7.         console.log(p1);
8.         console.log(p2);
9.     }

10.    inner_func(tem + 10,
tem);
11.    console.log(tem);
12. }
```

参数传递-按名传递

是Algol 60语言所定义的一种特殊的参数传递方式。

被调用过程中的局部名字不能与调用过程中的名字重名。

可以考虑在做宏扩展之前，对被调用过程中的每一个名字都系统地重新命名，即给以一个不同的新名字。

为保持实参的完整性，可以用括号把实参的名字括起来。

本章小结

1、活动记录

2、变量的存储分配

3、存储分配模式

4、非局部环境的引用

5、参数传递