编译原理

--运行时存储组织与管理

刘爽

中国人民大学信息学院

运行时环境

- 在生成目标代码之前，需要把静态的程序和实现这个程序的运行时的活动联系起来，主要是存储组织与管理
  - 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

目标程序运行时的活动

过程

- 过程：标识符所标记/指向的一个实体
  - 过程最简单的形式是一个标识符（过程名）和一段语句（过程体）组成
  - 函数是具有返回值的过程，也放入过程中进行讨论
  - 把完整的程序也看成过程
- 过程的调用：过程名出现在可执行语句里时
  - 过程名也可以出现在表达式里（有返回值的函数调用）
- 过程中定义的标识符
  - 形式参数
  - 局部变量

过程的活动

- 一个过程的活动指的是该过程的一次执行,过程的静态源程序和它的目标程序在运行时的活动之间的关系
- 过程P一个活动的生存期，指的是从执行该过程体第一步操作到最后一步操作之间的操作序列，也包括执行P时调用其他程序花费的时间
  - 每次控制从过程P进入过程Q后，如果没有错误，最后都要返回到过程P。
- 如果a和b都是过程的活动，那么它们的生存期或者是不重叠，或者是嵌套的。
  - 嵌套：控制在退出a之前进入b，那么必须在退出a之前退出b
- 如果一个过程是递归的（直接递归或间接递归），在某一时刻可能有几个活动记录活跃着

运行时存储器的组织

程序运行时存储空间的布局（*layout*）

– 典型的程序布局

- 保留地址区

  目标机体系结构和操作系统专用

- 代码区 静态存放目标代码

- 静态数据区

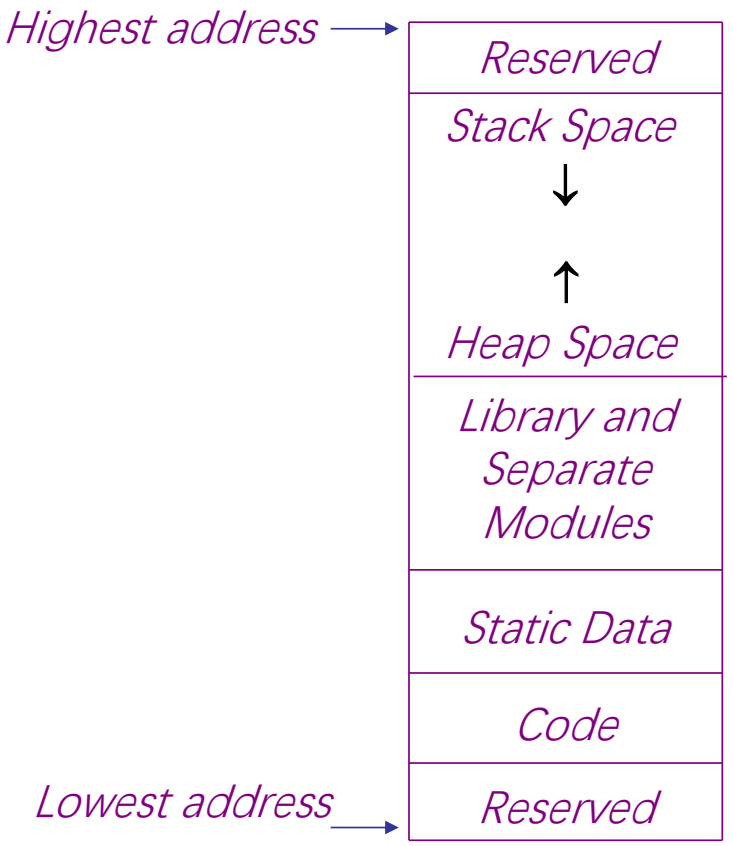
  静态存放全局数据

- 共享库和分别编译模块区

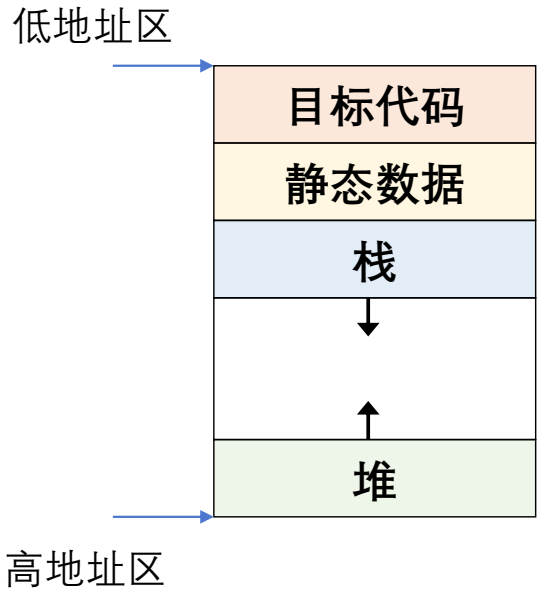
  静态存放这些模块的代码和全局数据

- 动态数据区

  运行时动态变化的堆区和栈区



存储组织

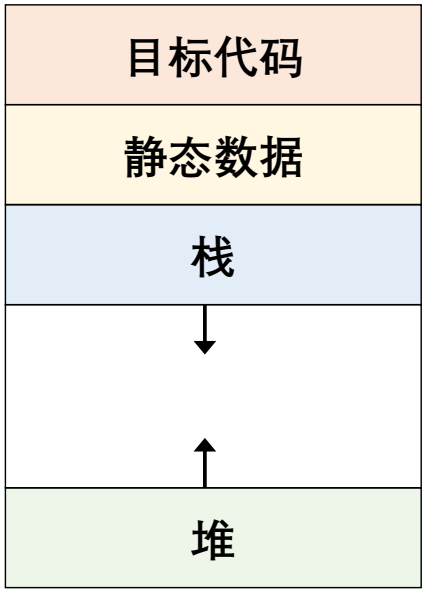


- **目标代码**
  - 长度在编译时可以确定
  - 放在静态区域，内存的低地址区
- **静态数据**
  - 某些数据的长度在编译时可知
  - 放在静态区域，其地址可以编译到目标代码中
- **拓广的控制栈**
  - 栈保存（时间上嵌套的）过程的活动信息
  - 堆用来保存那些生命周期不确定，或者将生存道被程序显示删除为止的数据

存储分配策略

存储分配策略

- 存储分配的三种策略
  - 静态分配策略
    - 在编译时为所有数据对象分配固定的存储单元，且在运行时始终保持不变
  - 栈式动态分配策略
    - 在运行时按栈方式管理运行时的存储空间
  - 堆式动态分配策略
    - 在运行时根据需要从堆数据区域分配和释放存储空间

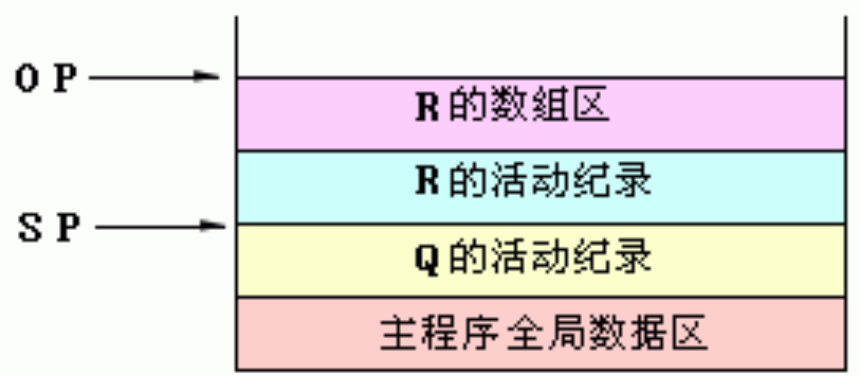


静态存储分配

- 在静态分配中，名字在程序编译时与存储单元绑定
  - 因为运行时不改变绑定，所以每次过程活动时，它的名字都绑定到同样的存储单元。
  - 这种性质允许局部名字的值在过程停止活动后仍然保持，即当控制再次进入过程时，局部名字的值同控制上一次离开时一样。
  - **适合**：编译时在目标代码中能填上所有操作的数据对象的地址
- 静态分配的局限性
  - 数据对象的长度和它在内存中的位置的约束在编译时必须知道
  - **不适合**：递归过程，因为递归过程一个过程的所有活动使用同样的局部名字绑定
  - 数据结构不能动态建立，因为没有运行时的存储分配机制。

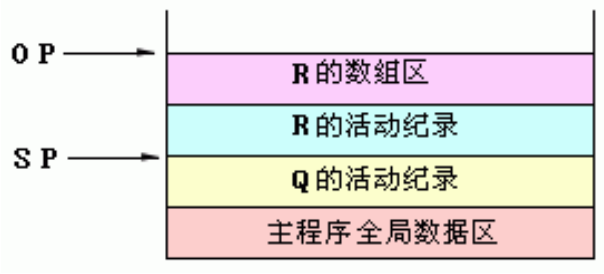
栈式存储分配

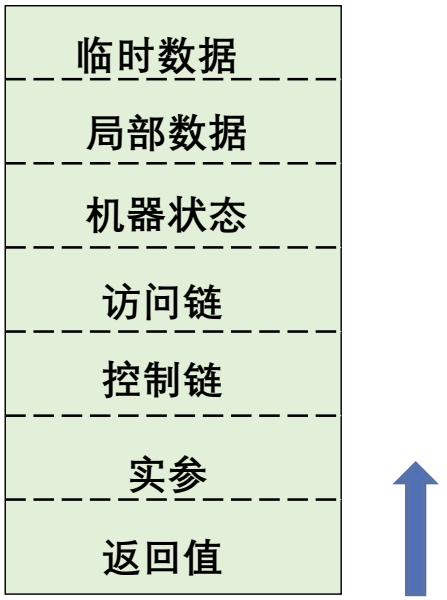
- 栈式存储分配的思想（基于控制栈）
  - 把存储空间组织为栈，而且随着过程活动的开始和结束将活动记录进栈和出栈
  - 过程每次调用时，局部量的存储空间包含在该次调用的活动记录中
  - 每次调用都引起新的活动记录进栈，每次活动时局部量都绑到新的存储单元
  - 活动记录弹出栈时局部量的存储空间将被释放，所以活动结束时局部量的值被删除
  - 用于有效实现可动态嵌套的程序结构
    - 递归过程、函数等



栈式存储分配

- 代码序列：实现过程调用的代码段
- 调用代码序列：
  - 为活动记录在栈中分配空间，并在此记录的字段中填写信息
  - 调用序列的代码常常分成两部分，分别处于调用过程和被调用过程中。
- 返回代码序列：
  - 恢复机器状态，使调用过程能继续执行。





栈式存储分配

- 在活动记录中，控制链、访问链和机器状态域出现在中间。
- 临时数据域的长度可以在编译时最终确定，但就前端而言，这个域的大小也可能是未知的。
- 临时数据放在局部数据域后面，它的长度的改变不会影响数据对象相对于中间那些域的位置。
- 返回地址和实参放在活动记录的最开始。方便调用者和被调用者之间的数据交换。

堆式存储分配

– 从堆空间为数据对象分配/释放存储

- 灵活 数据对象的存储分配和释放不限时间和次序

– 显式的分配或释放（*explicit allocation / deallocation*）

- 程序员负责应用程序的（堆）存储空间管理（借助于编译器与（或）运行时系统所提供的默认存储管理机制）
- 如：Pascal 中的 *new*，*deposit,*　　C++ 中的 *new*，*delete*
- C 语言没有堆空间管理机制，*malloc()* 和 *free()* 是标准库中的函数，可以由 *library vendor* 提供

– 隐式的分配或释放（*implicit allocation / deallocation*）

- （堆）存储空间的分配或释放不需要程序员负责，由编译器与（或）运行时系统自动完成
- Java采用垃圾回收（*garbage collection*）机制

堆式存储分配

- 显式释放堆空间的方法
  - 用户负责清空无用的数据空间（通过执行释放命令）
  - 堆管理程序只维护可供分配命令使用的空闲空间
  - 问题：可能导致灾难性的 *dangling pointer* 错误

例：Pascal 代码片断

```
var p,q: ^real;
…
new(p);
q:=p;
dispose(p);
q^:=1.0;
```

C++ 代码片断

```
 float  * p,*q;
…
 p=new  float;
q=p;
delete p;
*q:=1.0;
```

堆式存储分配

- 栈式存储分配策略在下列情况下不能使用:
  - 活动结束时必须保持局部名字的值
  - 被调用者的活动比调用者的活动的生存期长。
- 堆式存储器的策略: （堆管理器管理堆空间）
  - 把连续存储区域分成块，当活动记录或其他对象需要时分配。
  - 块的释放可以按任意次序进行，所以经过一段时间后，堆可能包含交错的正在使用的和已经释放的区域

堆管理方式

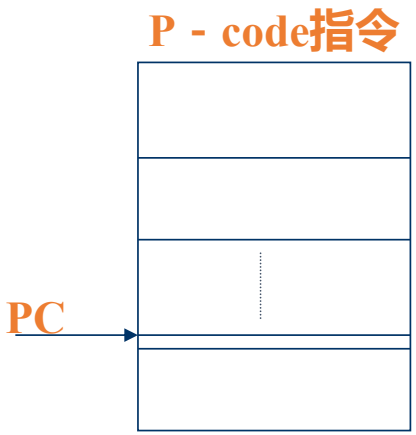
- 定长块管理：将堆存储空间分成长度相等的块，组织成一个链表
- 变长块管理
  - 保存一个空闲块链表
  - 分配方法：
    - 首次满足法/最先适应算法：在空闲块链表中搜到满足大小的块就分配
    - 最优满足法/最佳适应算法：为请求分配一个大小不小于请求的最小块。
    - 最差满足法：按块从大到小排序空闲块链表，每次从链表头分配
- 其具体管理方法可以参考操作系统中堆内存的管理方法。

对非局部名字的访问

- 语言的作用域规则确定了如何处理非局部名字的访问
  - 词法作用域规则（静态作用域规则）：仅仅根据程序正文即可以确定用于名字的声明。如最近嵌套规则
  - 动态作用域规则：在运行时根据当前的活动来决定用于名字的声明。

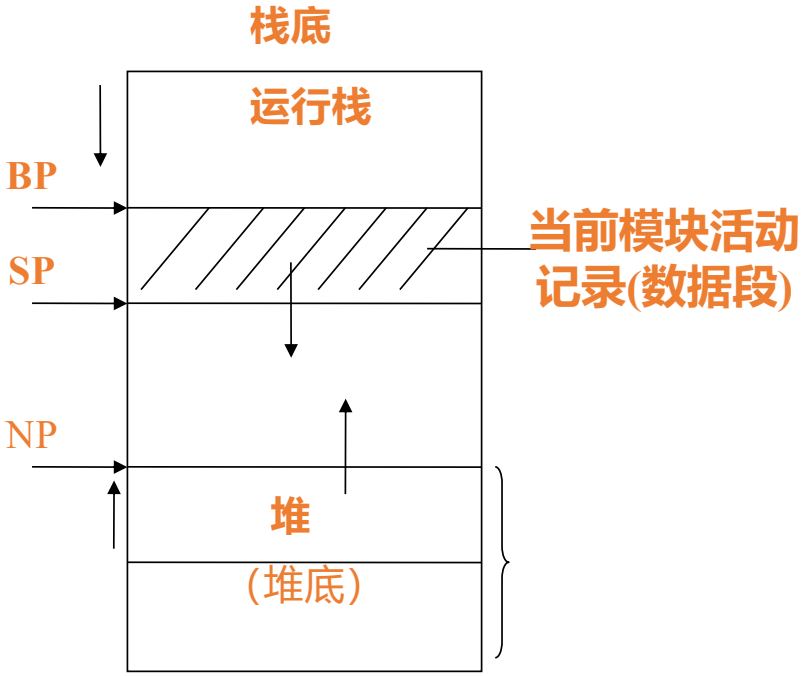
栈式抽象机：由三个存储器、一个指令寄存器和多个地址寄存器组成。

存储器：
- 数据存储器 （存放AR的运行栈）
- 操作存储器 （操作数栈）
- 指令存储器



程序指令存储器

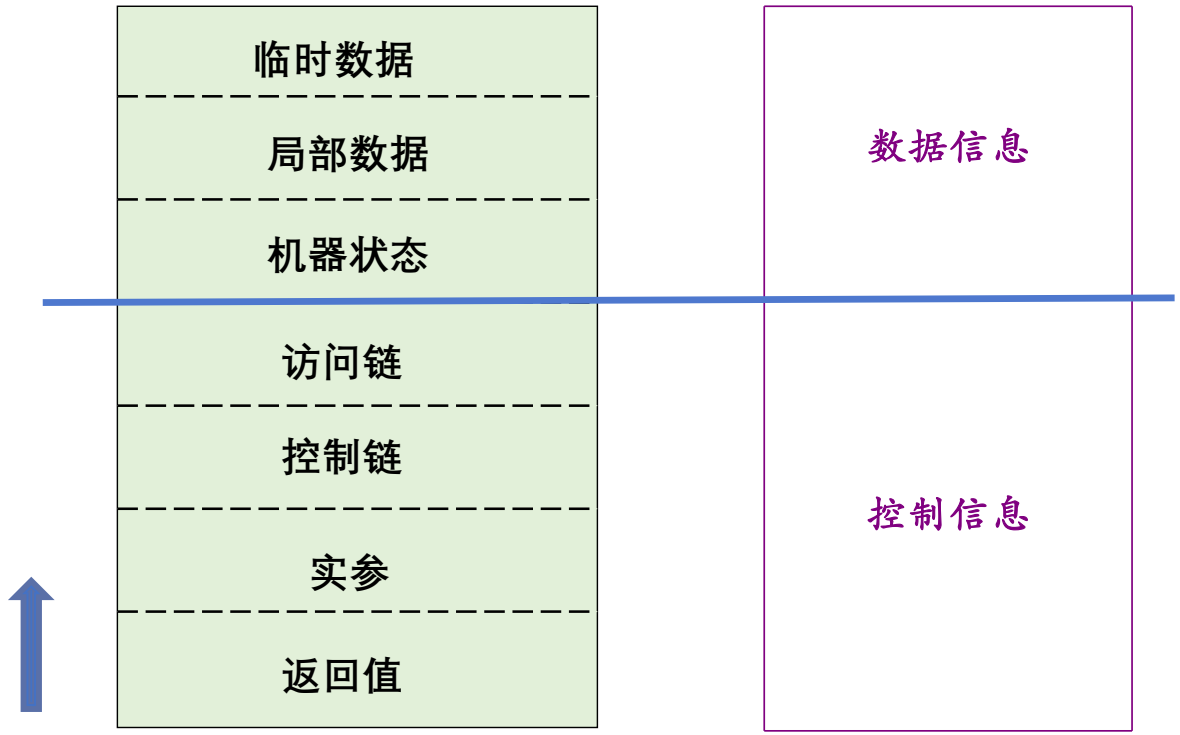
计算机的存储大致情况如下：



活动记录

活动记录

- 活动记录的定义
  - 过程一次执行所需要的信息用一块连续的存储区来管理，这块存储区叫做活动记录
  - 在过程调用时将活动记录压入栈，在控制返回调用者时把活动记录弹出栈



活动记录各部分信息

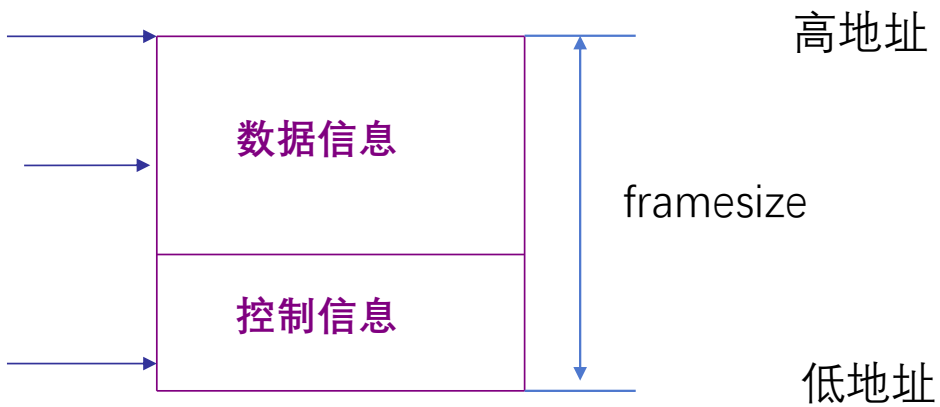
- 临时数据域：计算表达式时出现的中间结果
- 局部数据域：保存局部于过程执行的数据
- 机器状态域：保存过程调用前的机器状态信息，包括程序计数器的值和寄存器的值
- 访问链：引用存于其它活动记录中的非局部变量（静态链）
- 控制链：用来指向调用者的活动记录（动态链）
- 实参域：用于存放调用过程提供给被调用那个过程的参数
- 返回值域：用于存放被调用返回给调用过程的值

活动记录

栈指针 SP

某个数据对象的地址=
活动记录起始地址 FP
+ 偏移地址（*offset*）

活动记录起始地址 FP



活动记录

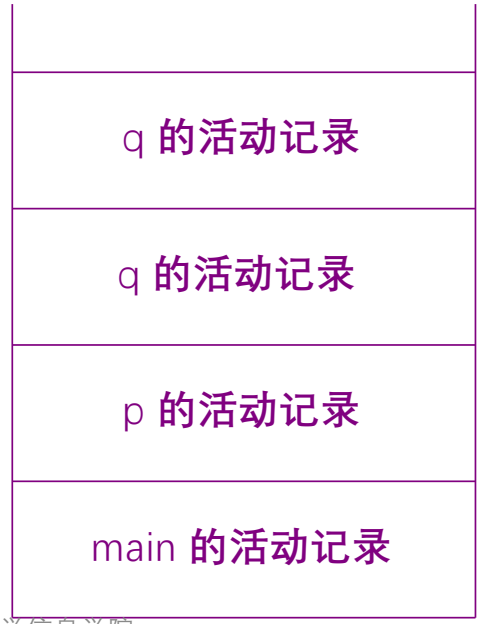
– 过程活动记录的栈式分配举例

```
void p( ) {
   …
   q( );
}

void q( ) {
   …
   q( );
}

int main  {
   p( );
}
```

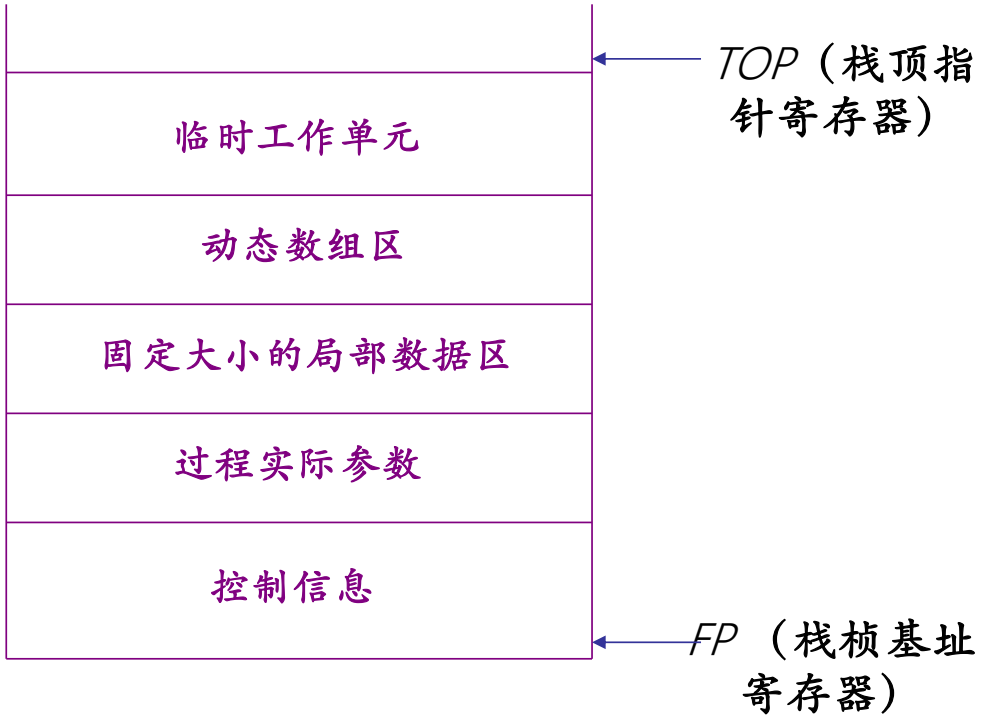
函数 q 被第二次激活时运行栈上活动记录分配情况



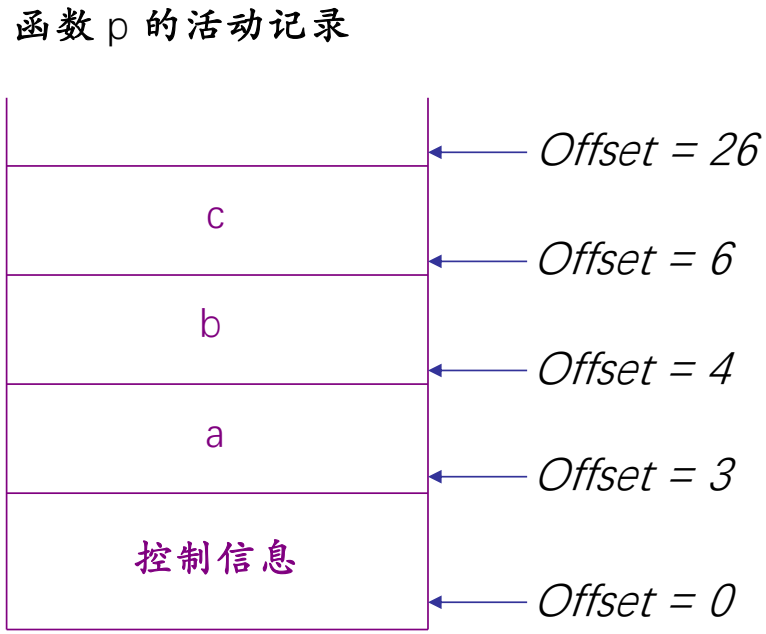
活动记录

```
void p( int a) {
  float b;
  float c[10];
  b=c[a]; }
```

– 典型的过程活动记录结构



– 过程活动记录举例



活动记录

– 过程活动记录举例

```
static int N;

void p( int a) {
   float b;
   float c[10];
   float d[N];
   float e;
   ...

}  /*d为动态数组*/
```

活动记录

– 含嵌套过程说明语言的栈式分配

- 主要问题: 解决对非局部量的引用（存取）
- 解决方案

➢ 采用 Display 表

➢ 为活动记录增加静态链域

活动记录

– 嵌套过程语言的栈式分配

- 采用 Display 表 （或称全局 Display 表）

  Display 表记录各嵌套层当前过程的活动记录在运行栈上的起始位置（基地址）

  当前激活过程的层次为K（主程序的层次设为*0*），则对应的 Display 表含有 K+1 个单元，依次存放着现行层，直接外层…直至最外层的每一过程的最新活动记录的基地址

  嵌套作用域规则确保每一时刻Display 表内容的唯一性

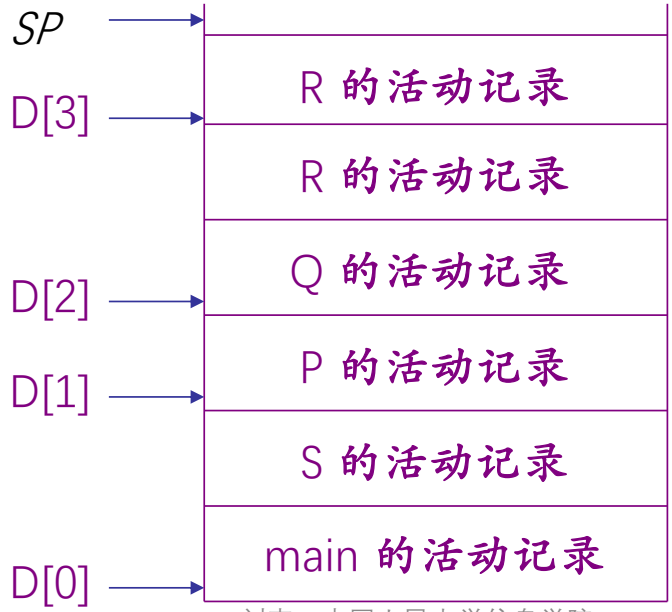
  Display 表的大小（即最多嵌套的层数）取决于实现

活动记录

– 嵌套过程语言的栈式分配

- Display 表方案举例

过程 R 被第二次激活后运行栈和 Display 寄存器 D[i] 的情况




```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
      begin
        … R; …
      end;  /*R*/
    begin
      … R; …
    end;  /*Q*/
  begin
    … Q; …
  end;  /*P*/
procedure S;
  begin
    … P; …
  end;  /*S*/
begin
  …  S; …
end.  /*main*/
```

活动记录

– 嵌套过程语言的栈式分配

- Display 表的维护（过程被调用和返回时的保存和恢复）

方法一 极端的方法是把整个 Display 表存入活动记录
若过程为第 *n* 层，则需要保存 D[0] ~D[*n*] ）

一个过程（处于第 *n* 层）被调用时，从调用过程的 Display 表中自下向上抄录 *n* 个 TOP 值，再加上本层的 TOP 值

方法二 只在活动记录保存一个的 Display 表项，在静态存储区或专用寄存器中维护全局 Display 表

活动记录

– 嵌套过程语言的栈式分配

- 采用静态链（*static link*）

  Display 表的方法要用到多个存储单元或多个寄存器，有时并不情愿这样做，一种可选的方法是采用静态链

  所有活动记录都增加一个静态链（如在offset 为 0 处）的域，指向定义该过程的直接外过程（或主程序）运行时最新的活动记录

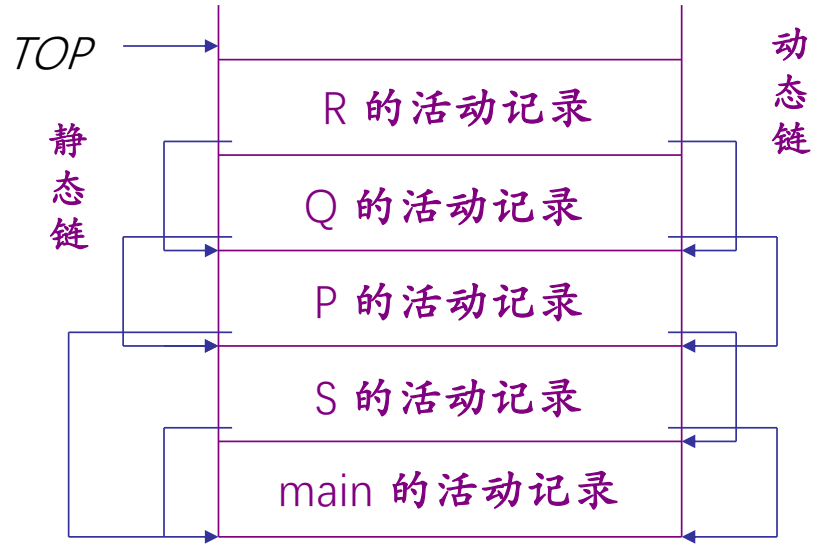
  在过程返回时当前 AR 要被撤销，为回卷（*unwind*）到调用过程的AR（恢复FP）需要用到动态链域

活动记录

– 嵌套过程语言的栈式分配

- 采用静态链的方法举例

过程 R 被第一次激活后运行栈的情况




```
program Main( I,O);
procedure P;
  procedure Q;
    procedure R;
      begin
        … R; …
      end;  /*R*/
    begin
      … R; …
    end;  /*Q*/
  begin
    … Q; …
  end;  /*P*/
procedure S;
  begin
    … P; …
  end;  /*S*/
begin
  …  S; …
end.  /*main*/
```

过程调用与参数传递

参数传递

- 说明的作用域
  - 如果一个说明的作用域是在一个过程里，那么这个过程里出现的该说明中的名字都是局部于本过程的；
  - 除上述之外的名称是非局部的。
- 参数传递方式：过程的形式参数和实在参数的对应方式。
- 形式参数和实在参数的“左值”和“右值”之间的对应关系划分参数传递方式：
  - 传值调用
  - 引用调用（传地址调用）
  - 复制-恢复调用
  - 传名调用

–表达式的左值代表存储该表达式值的地址
–表达式的右值代表该表达式的值

参数传递——传值调用

```
swap(x,y)
int x,y
{   int temp;
    temp =x;
    x=y;
    y=temp;
}
main()
{   int a=1,b=2;
    swap(a,b);
    printf("a is %d, b is %d", a,b);
}
```

- 传值调用：计算实参，并把它的右值传给被调用过程
  - 把形参当作局部名字看待，形参的存储单元在被调用过程的活动记录中
  - 调用者计算实参，并把其右值放入形参的存储单元中
  - 传值调用的显著特征是对形参的运算不影响调用者活动记录中的值
- 打印结果
  - a is 1, b is 2

参数传递——引用调用

```
swap(x,y)
int x,y
{   int temp;
    temp =x;
    x=y;
    y=temp;
}
main()
{   int a=1,b=2;
    swap(a,b);
    printf("a is %d, b is %d", a,b);
}
```

- 引用调用：传递时，调用过程把实参存储单元的地址传递给被调用过程
  - 如果实参是有左值的名字或表达式，则传递这个左值本身；
  - 如果实参是表达式，没有左值，则计算该表达式的值并存入新的存储单元，然后传递这个单元的地址
- 打印结果
  - a is 2, b is 1

参数传递——复制-恢复

```
swap(x,y)
int x,y
{   int temp;
    temp =x;
    x=y;
    y=temp;
}
main()
{   int a=1,b=2;
    swap(a,b);
    printf("a is %d, b is %d", a,b);
}
```

- 传值调用和引用调用的混合
  - 在控制流进入被调用过程之前计算实参，实参的右值像传值调用那样传递给被调用过程，此外如果实参有左值的话，在调用之前确定它的左值
  - 当控制返回时，将形参的当前右值复制回实参的左值，该左值是上述调用前计算的左值。
- 打印结果
  - a is 2, b is 1

参数传递——传名调用

```
swap(x,y)
int x,y
{   int temp;
    temp =x;
    x=y;
    y=temp;
}
main()
{   int a=1,b=2;
    swap(a,b);
    printf("a is %d, b is %d", a,b);
}
```

- 传名调用的用法类似于宏
  - 过程被看做宏，也就是说，在调用过程中将调用替换为被调用过程的过程体，但要把任何一个出现的形式参数都文字的替换为相应的实参
  - 被调用过程中局部名字要保持与调用过程中的名字不同
- 打印结果
  - a is 2, b is 1

过程调用与参数传递

– 过程/函数参数

- 不含嵌套过程/函数声明

  如 C 语言，任何过程/函数内部访问的非局部量只有全局量
  可以将所有全局量分配在静态区

  这种情况下，无论采取什么方式激活一个过程/函数，活动记录
  没有什么差异，局部数据在活动记录中访问，而非局部数据只
  有全局量，均在静态区访问

- 包含嵌套过程/函数声明

  介绍龙书中的一种解决方案：使用带静态链的函数实参（*call-by-closure*）