第九章　运行时存储空间组织

- In computer science, the **runtime system** is software that provides services for a running program but is itself not considered to be part of the operating system.
- Examples include:
  - the code that is generated by the compiler to manage the runtime stack.
  - library code for handling memory management (for example, malloc).
  - code that handles dynamic loading and linking.
  - debugger code that is generated at compile time or run time.
  - application-level thread management code.
- Byte-code interpreters and virtual machines can also be considered runtime systems.

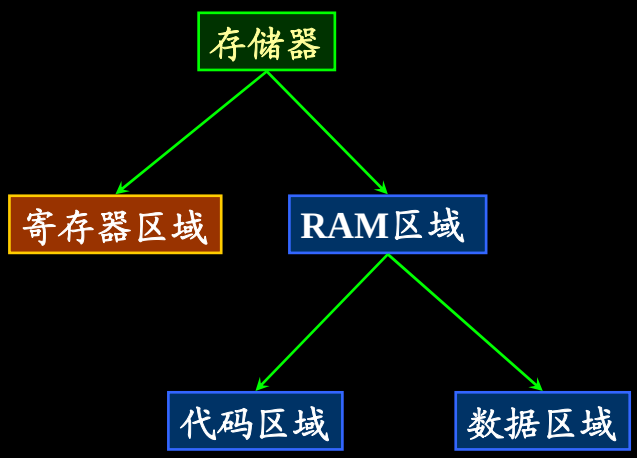
运行时存储空间组织（续）

- The structure of the target computer's registers and memory that serves to manage memory and maintain the information needed to guide the execution process.
- 有三种类型
  - 完全静态，FORTRAN77;
  - 基于栈的，C, C++, Pascal, and Ada
  - 完全动态的，LISP

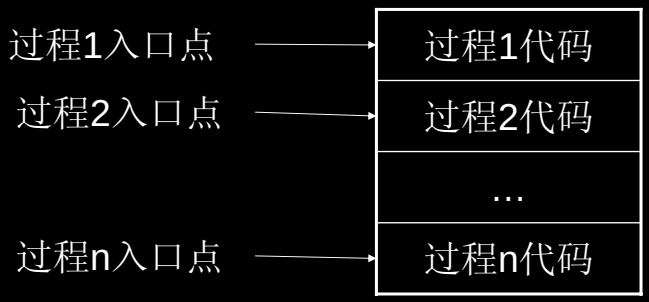
本章内容

- 运行时存储器的组织
- 目标程序运行时的活动
- 静态存储分配
- 简单的栈式存储分配
- 嵌套过程语言的栈式实现
- 堆式动态存储分配

9.1 运行时存储器的组织

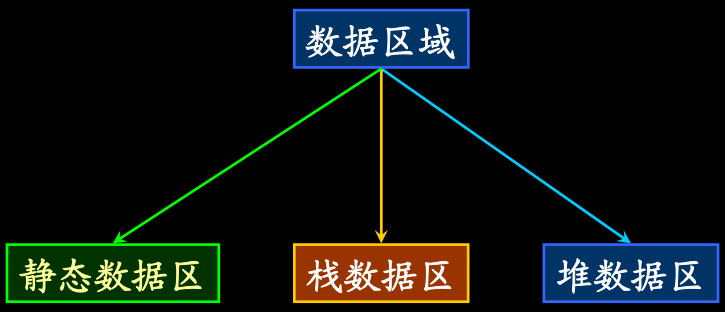


代码区域是静态的



- 在编译时代码及入口点已确定

数据区域



全局/静态数据区域

- FORTRAN77 所有数据
- Pascal全局变量
- C的外部变量和静态变量
- 此外，程序中的常量
  - C和Pascal用Const说明的
  - 文字值如字符串
  - 常数（小的常数可插入代码中）

静态数据区

- 编译时能确定

栈区与堆区

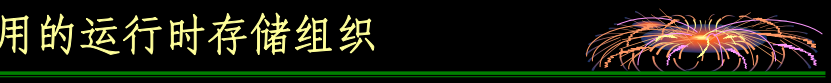
- 用于保存LIFO分配方式的数据
- 通常目标机提供处理器栈可利用
- 有时编译器显式在存储器中分配栈

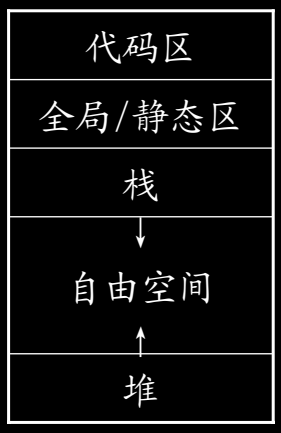
栈数据区

- 用于不具有LIFO方式的动态分配
  - 如C的指针

堆数据区

通用的运行时存储组织





- 栈按照地址增加方向生长
- 堆与栈共用自由空间
- 堆向地址减小方向扩张
- 堆的生长比栈复杂多

9.2目标程序运行时的活动

活动记录的一般形式

存放参数的空间
存放簿记信息的空间
存放局部数据的空间
局部临时空间

- 与目标机、语言甚至编译器作者的爱好有关
- 簿记区大小每个过程都一样
- 参数区和局部 数据区大小对单个过程始终不变
- 可分配活动记录的数据区域
  - 静态数据区
  - 栈数据区
  - 堆数据区

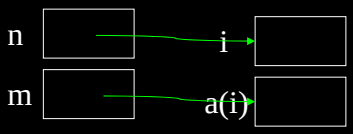
过程的活动

- 过程的活动：该过程的一次执行
- 过程活动的生存期：嵌套、并列
- 递归过程：直接递归、间接递归
- 说明的作用域：说明中的名字在作用域中都是局部的，其他名字是非局部的

```
program sort(input,output)
var a: array[0..10] of integer;
   x:integer;
   procedure readarray;
   var i:integer;
      begin…a….end;
   procedure exchange (i,j:integer);
      begin x:=a[i];a[i]:=a[j];a[j]=x end;
   procedure quicksort (m,n:integer);
      var k,v:integer;
      function partition (y,z:integer):integer;
      var i,j:integer;
         begin …a…v…exchange(i,j);…end;
   begin if(n>m)then begin
   i:=partition(m,n);
   qucksort(m,i-1); qucksort(i+1,n)…end
   end;
begin…readarray; quicksort(1,9) end.
```

参数传递

- 传地址(call-by-reference)
- 传值(call-by-value)
- 得结果(call-by-result)
- 传名(call-by-name)




```
swap(i,a(i))
```

```
procedure swap(n,m:real);
   var j:real;
   begin
      j:=n;
      n:=m;
      m:=j;
   end
```

Introduction

- Logically parameter-passing has following possible semantics:
  - IN: pass info from caller to callee
  - OUT: callee writes a value in the caller
  - IN/OUT: caller tells callee value of var, which may be updated by callee
- However, different mechanisms of implementing IN/OUT can have subtle semantics differences
- Some languages support many param passing modes (C++:by value, by reference, by const-reference; array-mode), some few (Java: by value)

Call by value

- IN mode:
  - can't have an effect on caller vars
  - unless value passed is a pointer(C) or referece
- Used by many languages, C/C++, Java,
- How it works:
  - Formal param is just like a local name: storage of formals in activation record of called proc.
  - Caller evaluates the actual params and sticks their R-values in the storage for the formals

L-value and R-value

- L-values are values that have addresses, meaning they are variables or dereferenced references to a certain memory location.
- R-value is either L-value or non-L-value — a term only used to distinguish from L-value.
- In C, the term L-value originally meant something that could be assigned to (coming from left-value, indicating it was on the left side of the = operator), but since 'const' was added to the language, this now is termed a 'modifiable L-value'.
- http://en.wikipedia.org/wiki/Value_(computer_science)

Call by reference

- IN/OUT mode
- General idea: 形实参指向同一内存单元
- Example languages: C++, Pascal
- How it works:
  - 如果形参是名字或具有L-值的表达式，将L-值（即，地址）传过去。
  - 如果不具有 L-值 (e.g.,i+3),则用临时单元对表达式求值，并传递这个单元的地址
  - 在被调过程中, 生成的对形参引用的代码能够逆向引用(dereference)传过去的指针
- 即， the compiler does for us what programmers are forced to do themselves in C to get IN/OUT by “pass by pointer”

Call by value-return

- IN/OUT mode
- General idea: 调用时将值考进来，返回时考回去
- 例子：某些Fortran实现用此，其他用call-by-reference
- How it works:
  - 调用者对实参求值，R-values are passed(as in pass by value). L-values are also computed.
  - 从被调者返回时，current R-values of formals are copied back into L-values of actuals.

比较call-by-reference和value-return

- value-return在调用/返回时开销大；但变量引用代价小
- value-return可用在调用者和被调者位于不同地址空间（如RPC）
- 有时二者会得到不同结果

Call-by-name

- IN/OUT mode
- General idea: similar to macro-expansion
- Example laguages: Algol-60, some functional language (Haskell)
- How it works
  - the arguments to functions are not evaluated at all — rather, function arguments are substituted directly into the function body using *capture-avoiding substitution*. If the argument is not used in the evaluation of the function, it is never evaluated; if the argument is used several times, it is re-evaluated each time.

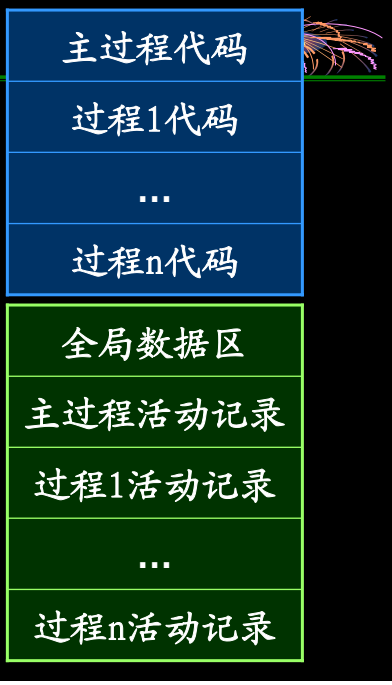
```
int i;
int A[3];
void Q(int B) {
  A[1] = 3;
  i = 2;
  write(B);
  B = 5;
}
int main() {
  i = 1;
  A[1] = 2;
  A[2] = 4;
  Q(A[i]);
}
```

9.3 静态存储分配

- 无指针
- 无动态存储分配
- 过程不能递归调用

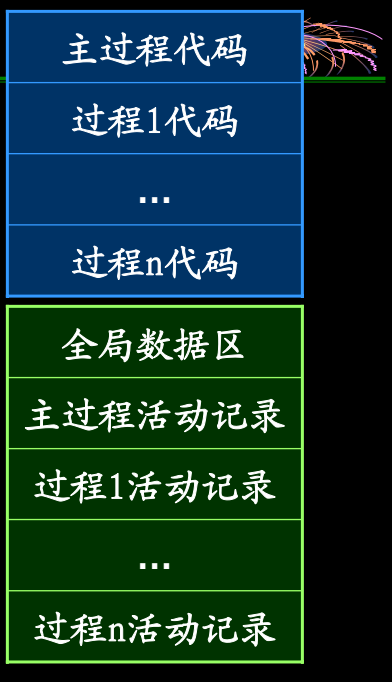
9.3 静态存储分配

- 所有变量都是静态分配
- 每个过程只有一个活动记录
- 活动记录在执行之前分配好
- 无论局部全局变量，可通过固定地址直接访问



FORTRAN77存储分配

- **Calling Sequence**
  - 计算实参
  - 实参值存放在被调者的对应参数单元
  - 保存返回地址
  - 跳转到被调者代码入口点
- **Return Sequence**
  - 跳转到返回地址

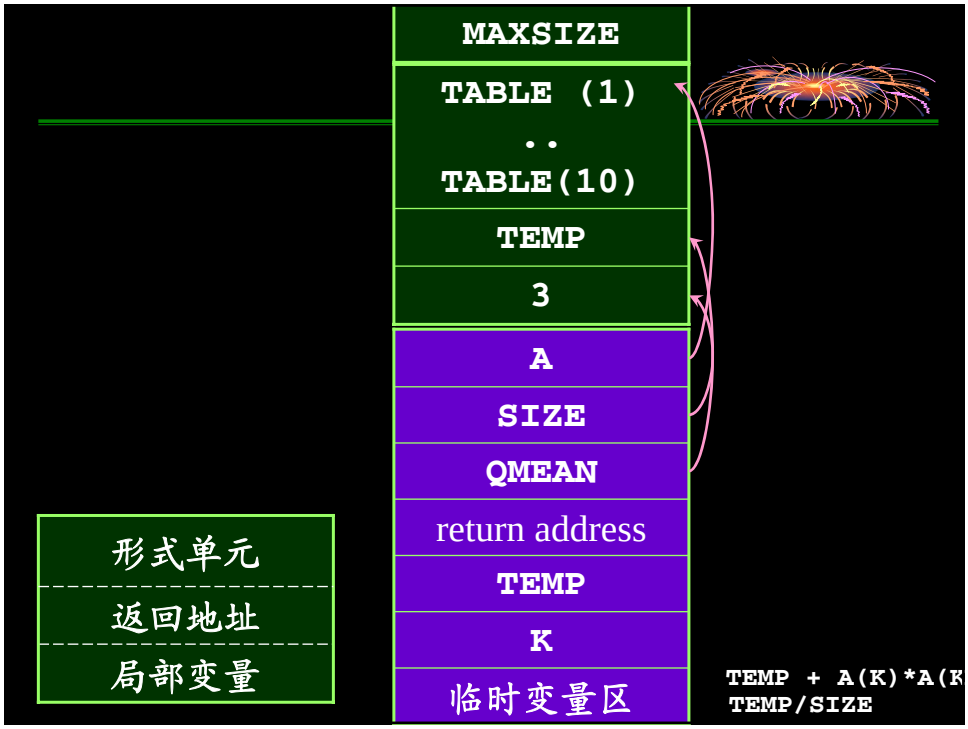


例. FORTRAN77存储分配

```
      PROGRAM TEST
      COMMON MAXSIZE
      INTEGER MAXSIZE
      REAL TABLE(10),TEMP
      MAXSIZE = 10
      READ *, TABLE(1), TABLE(2),TABLE(3)
      CALL QUADMEAN(TABLE,3,TEMP)
      PRINT *, TEMP
      END
```

例.（续）

```
      SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
      COMMON MAXSIZE
      INTEGER MAXSIZE, SIZE
      REAL A(SIZE), QMEAN, TEMP
      INTEGER K
      TEMP = 0.0
      IF((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)GOTO 99
      DO 10 K = 1, SIZE
      TEMP = TEMP + A(K)*A(K)
10    CONTINUE
99    QMEAN = SQRT(TEMP/SIZE)
      RETURN
      END
```



9.3.2 临时变量的地址分配

- 产生四元式时不加限制引进临时变量;
- 是初等类型的简单变量;
- 只需要在出现的地方附带类型信息而没有必要登记入符号表
  - 临时变量的作用域不相交,则可以公用一个存储单元
  - 大部分临时变量名用来存放表达式的中间结果,这类变量名有一个特点,它们均只被定值一次,被引用一次.它们的作用域是层次嵌套的

分配时首先求出作用域，然后检查每个已分配单元，如果其作用域跟当前这个不相交则将这个单元也分配给当前临时变量，否则分配一个新的单元。

例子:对赋值语句 X:= A×B–C×D+E×F 产生的临时变量的处理

四元组				临时变量	地址
×	A	B	T1	T1	a
×	C	D	T2	T2	a+1
–	T1	T2	T3	T3	a
×	E	F	T4	T4	a+1
+	T3	T4	T5	T5	a
:=	T5		X		

“代真”后的四元式序列

四元组				k当前值a
×	A	B	$a	a+1
×	C	D	$a+1	a+2
–	$a	$a+1	$a	a+1
×	E	F	$a+1	a+2
+	$a	$a+1	$a	a+1
:=	$a		X	赋值时k++引用时k--

9.4 动态存储分配 --- 简单栈式

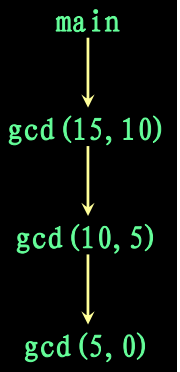
C语言特点:

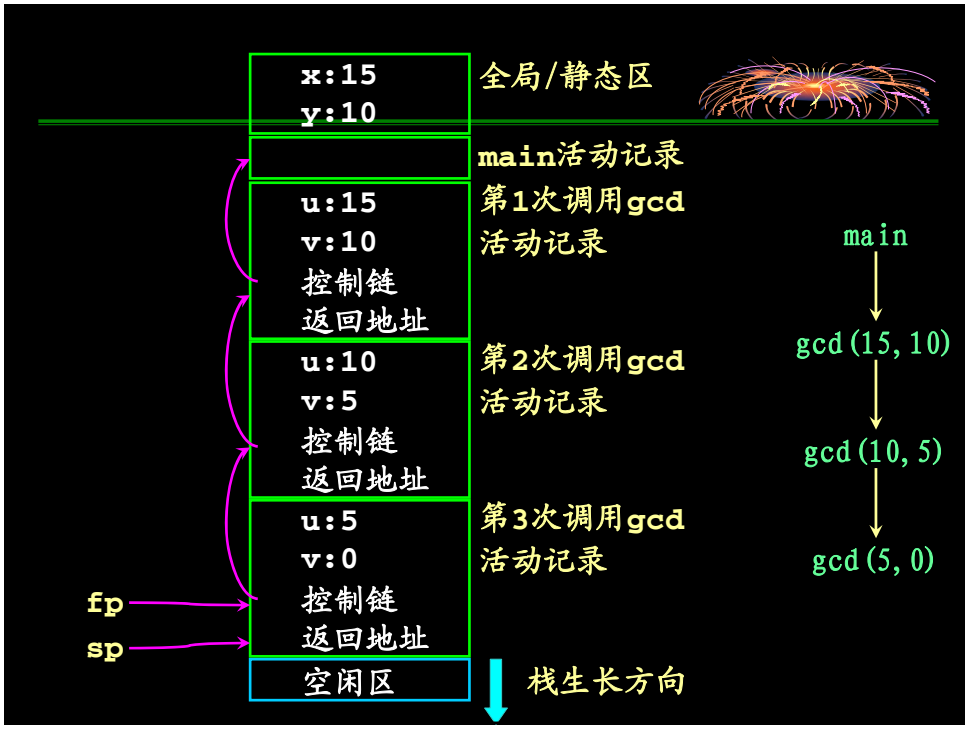
- 没有分程序结构;
- 过程定义不允许嵌套;
- 允许过程递规调用;
- 允许过程含有可变数组;(但不允许全局可变数组)

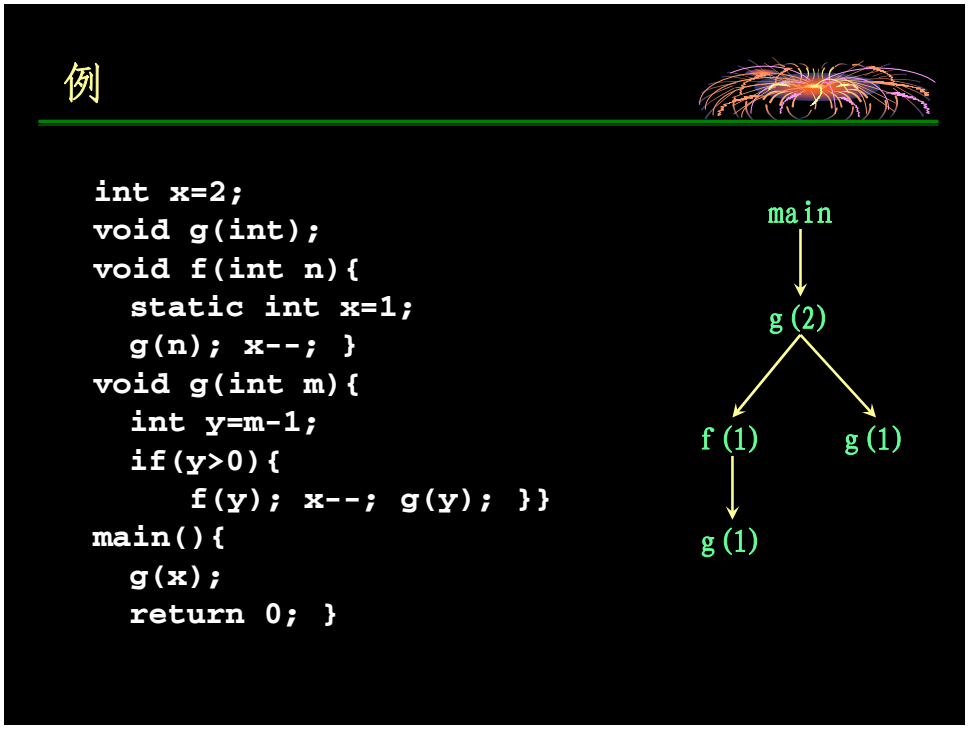
```
全局数据说明
main( ){
    数据说明
}
void R( ){
    数据说明
}
…
void Q( ){
    数据说明
}
```

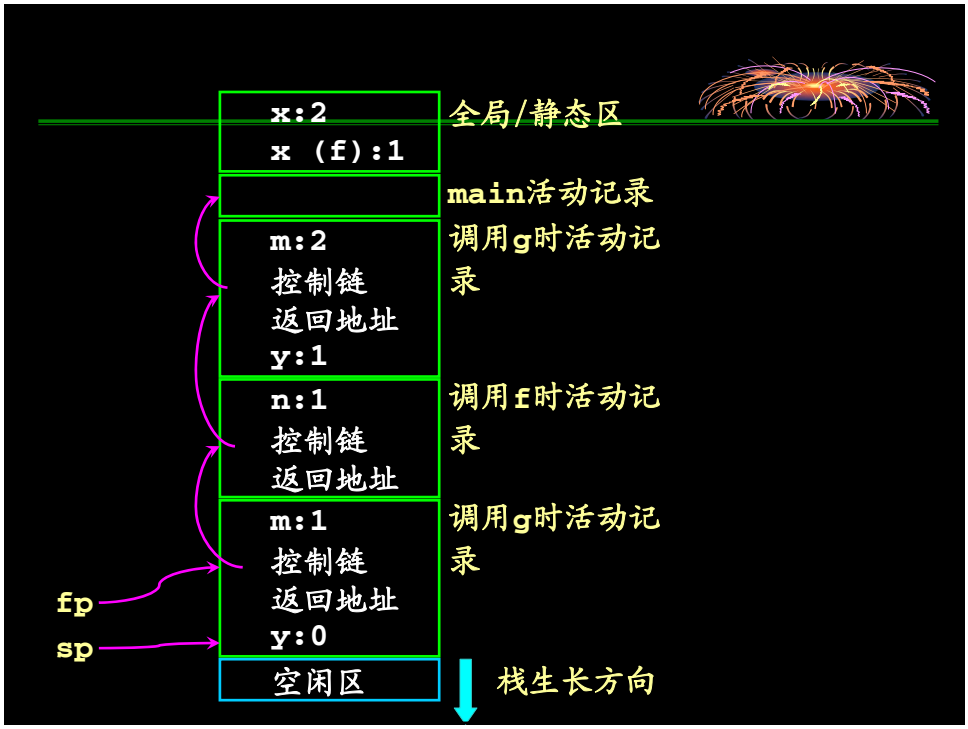
例

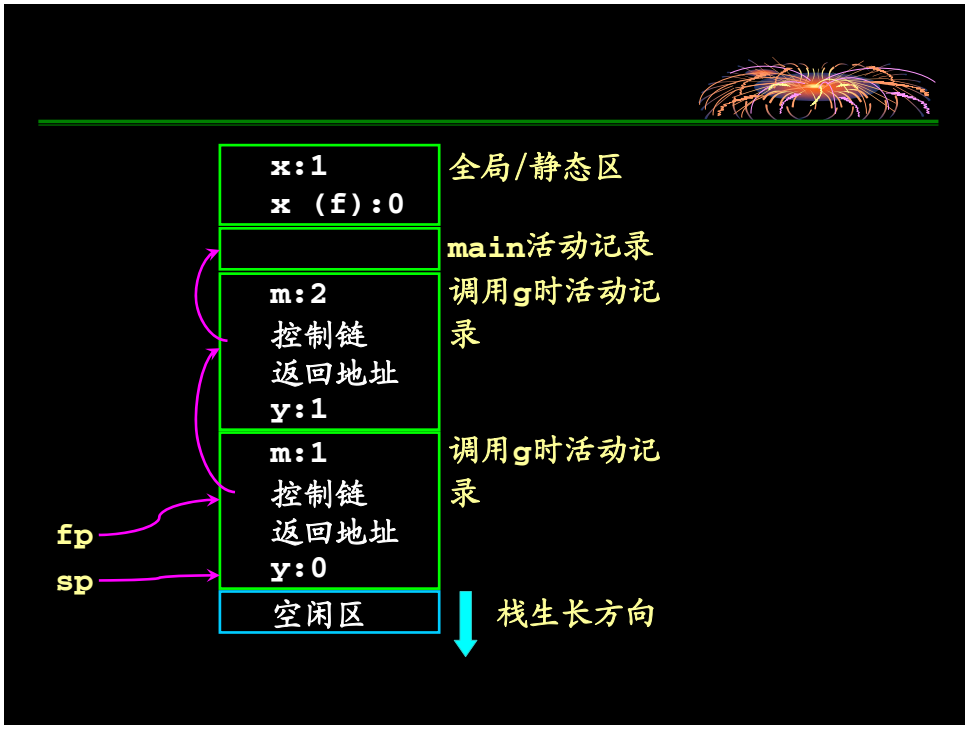
```
#include <stdio.h>
int x,y;
int gcd(int u, int v){
  if(v==0)return u;
  else return gcd(v,u%v);
}
main(){
  scanf("%d%d",&x,&y);
  printf("%d\n",gcd(x,y));
  return 0;
}
```



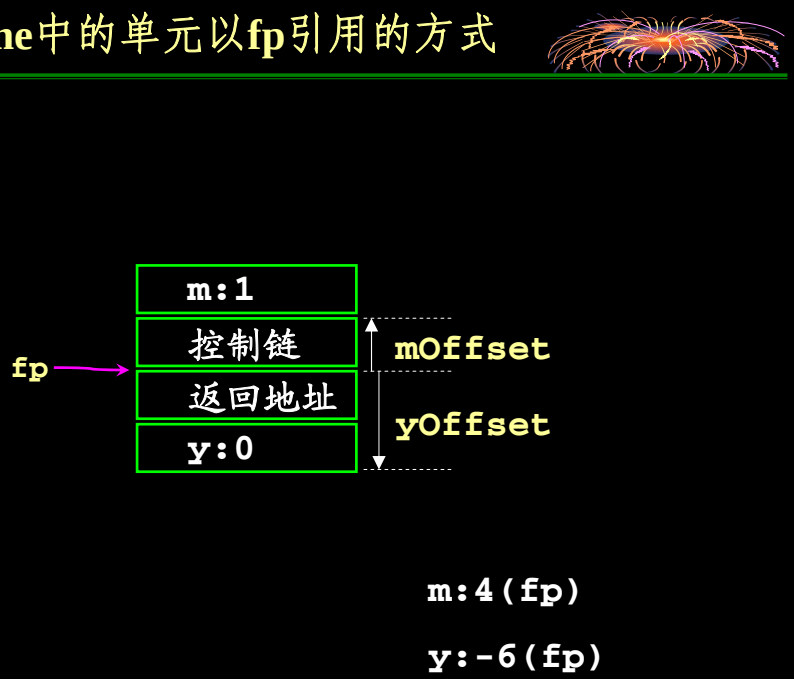

x:15
y:10
全局/静态区
main活动记录
u:15
v:10
控制链
返回地址
第1次调用gcd
活动记录
u:10
v:5
控制链
返回地址
第2次调用gcd
活动记录
u:5
v:0
控制链
返回地址
第3次调用gcd
活动记录
fp
sp
空闲区
栈生长方向
main
gcd(15, 10)
gcd(10, 5)
gcd(5, 0)


例
int x=2;
void g(int);
void f(int n){
static int x=1;
g(n); x--; }
void g(int m){
int y=m-1;
if(y>0){
f(y); x--; g(y); }}
main(){
g(x);
return 0; }
main
g(2)
f(1)
g(1)
g(1)

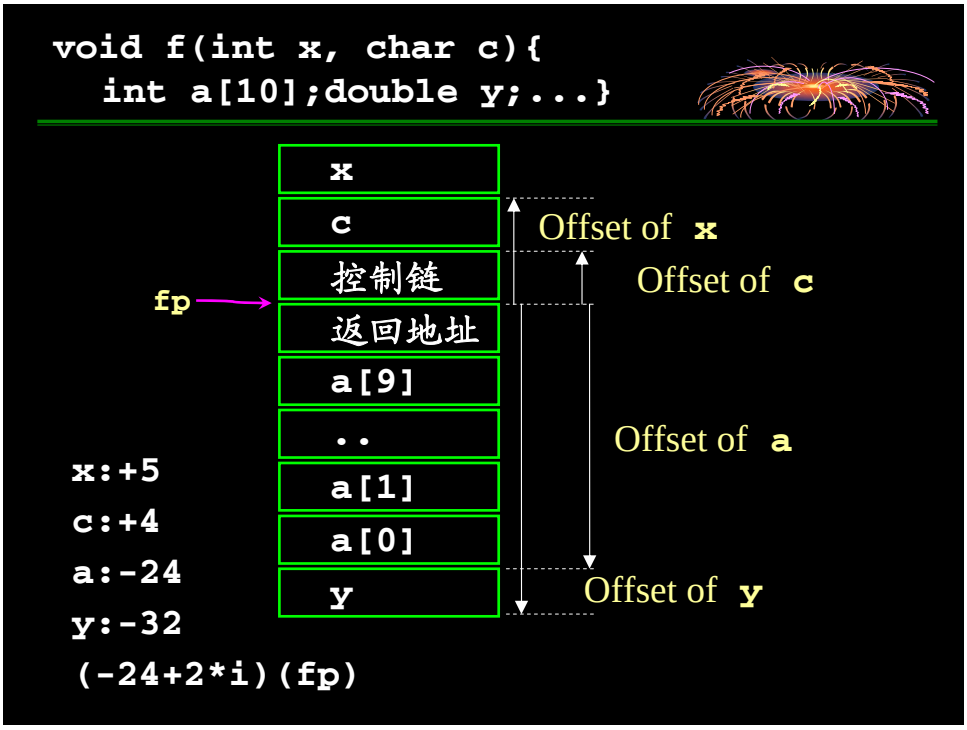

x:2
x (f):1
全局/静态区
main活动记录
m:2
控制链
返回地址
y:1
调用g时活动记录
n:1
控制链
返回地址
调用f时活动记录
m:1
控制链
返回地址
y:0
调用g时活动记录
fp
sp
空闲区
栈生长方向


x:1
x (f):0
全局/静态区
main活动记录
m:2
控制链
返回地址
y:1
调用g时活动记录
m:1
控制链
返回地址
y:0
调用g时活动记录
fp
sp
空闲区
栈生长方向

Frame中的单元以fp引用的方式




```
void f(int x, char c){
  int a[10];double y;...}
```

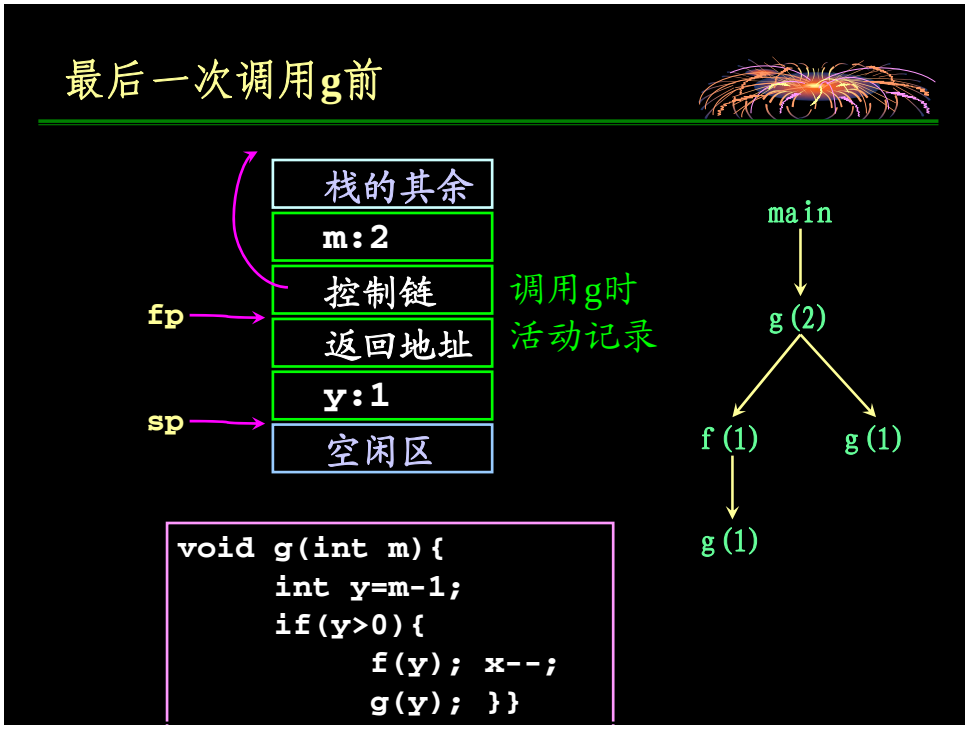


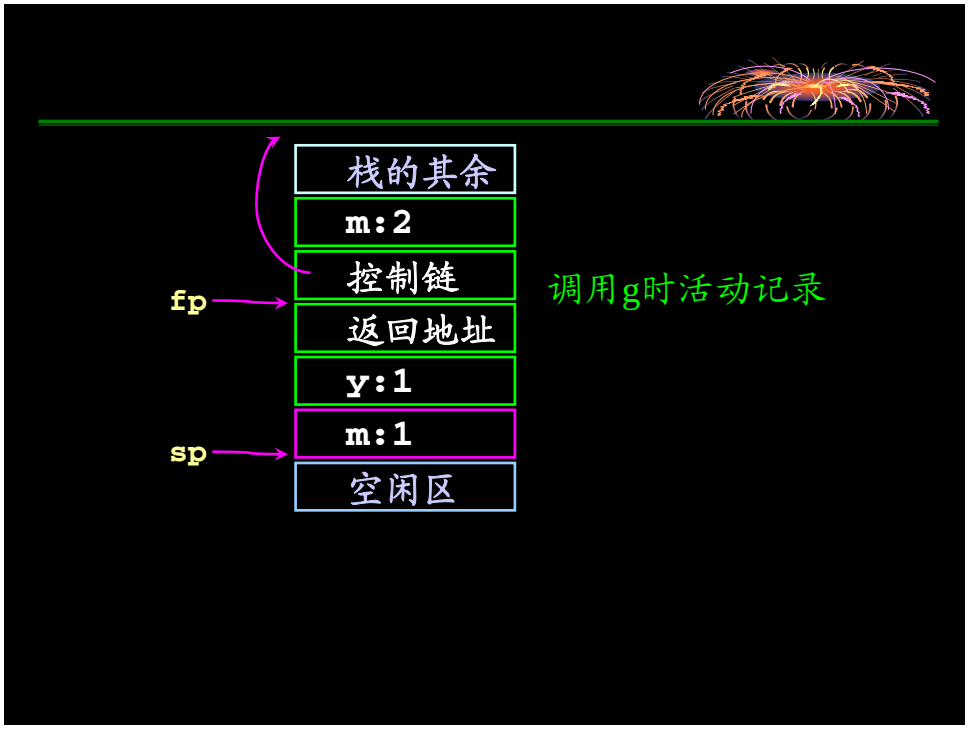
Calling Sequence

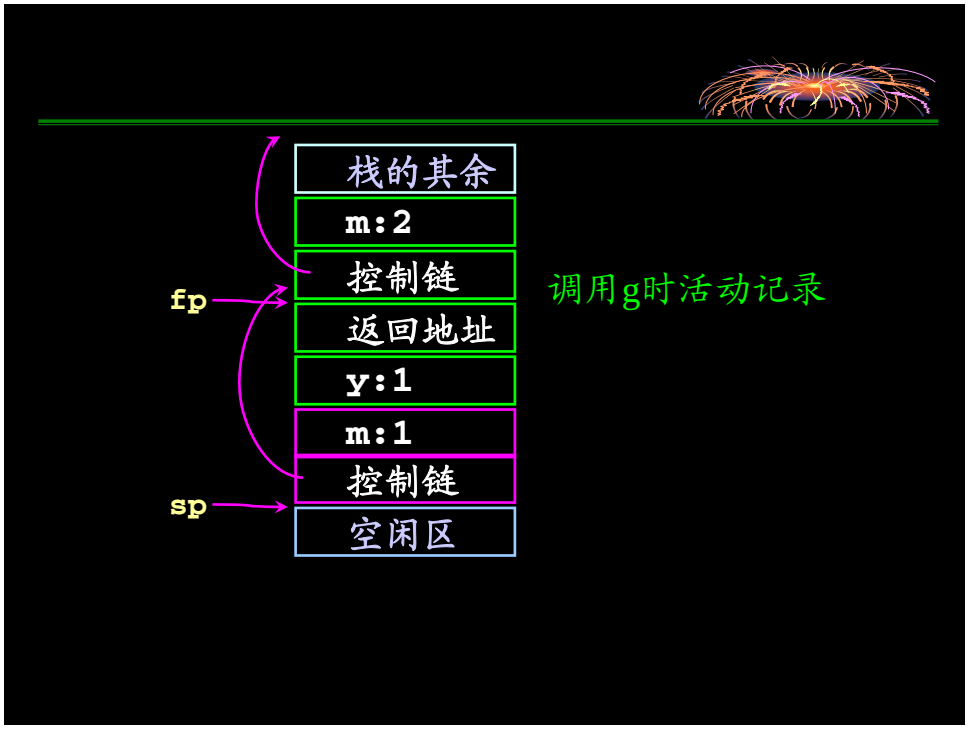
- 计算实参并将他们存放在刚建立的活动记录的正确位置
- 将**fp**存放在新活动记录的控制链中
- 修改**fp**指向新活动记录的开始位置（即**sp**赋给**fp**）
- 保存返回地址到新活动记录中（如需要）
- 跳转到被调过程的代码

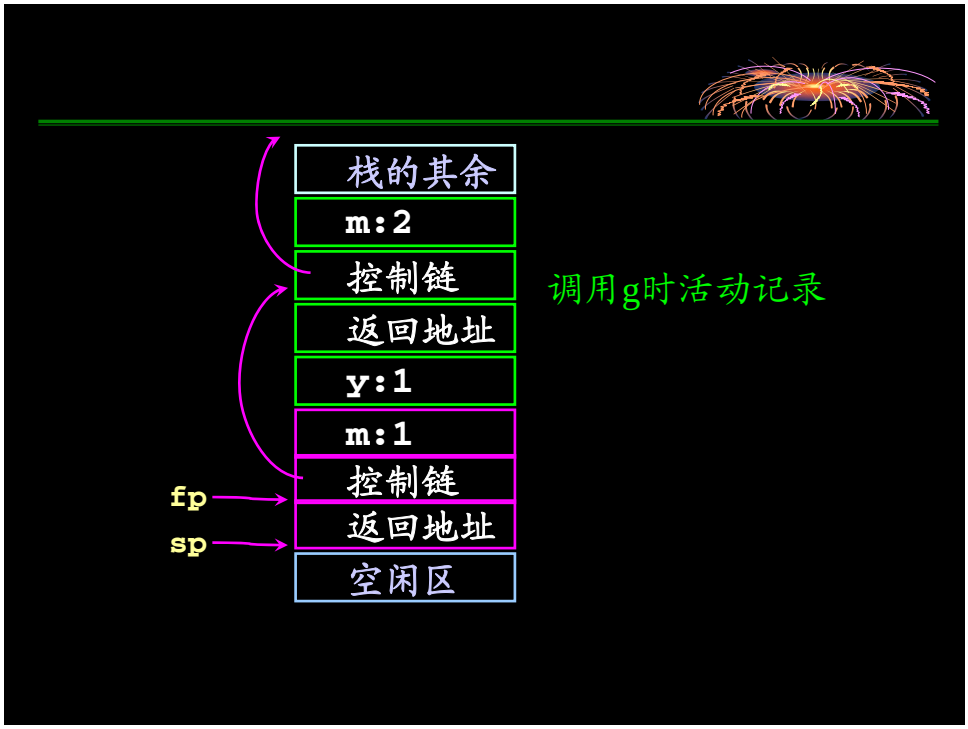
Return Sequence

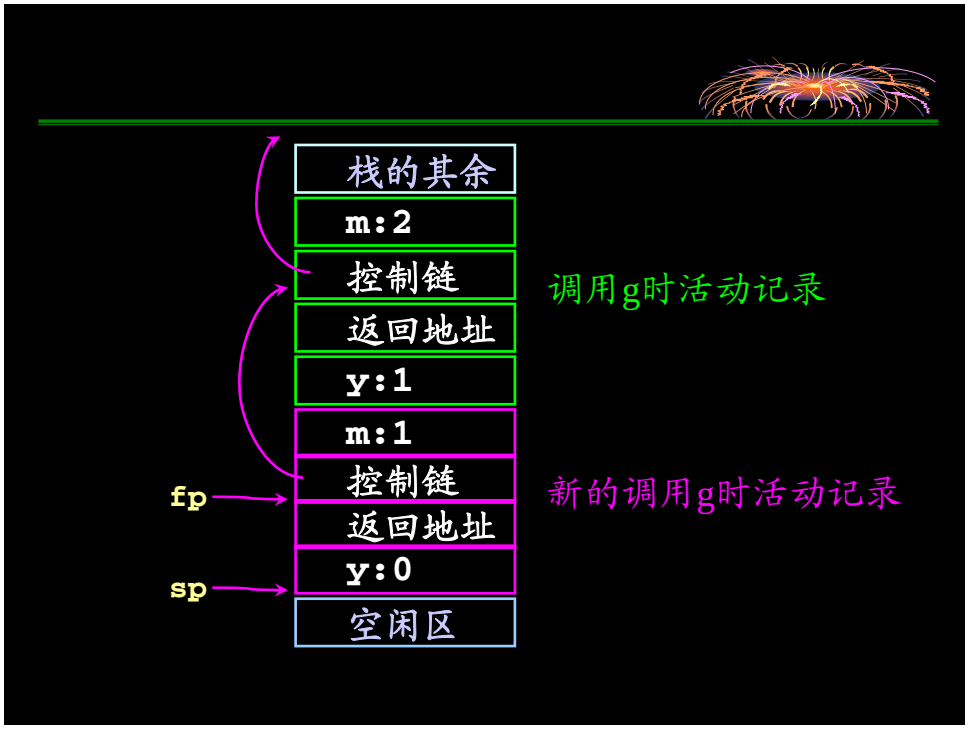
- 将**fp**复制到**sp**
- 将控制链装载给**fp**
- 跳转到返回地址
- 修改**sp**到弹出所有形参位置


最后一次调用g前
栈的其余
m:2
控制链
返回地址
y:1
空闲区
fp
sp
调用g时
活动记录
main
g(2)
f(1)
g(1)
g(1)
void g(int m){
int y=m-1;
if(y>0){
f(y); x--;
g(y); }}


栈的其余
m:2
控制链
返回地址
y:1
m:1
空闲区
fp
sp
调用g时活动记录


栈的其余
m:2
控制链
返回地址
y:1
m:1
控制链
空闲区
fp
sp
调用g时活动记录


栈的其余
m:2
控制链
返回地址
y:1
m:1
控制链
返回地址
空闲区
fp
sp
调用g时活动记录



处理变长数据

- 数据对象个数变化
- 每个数据对象大小变化

处理变长数据

- 实参的个数每次调用时不一样

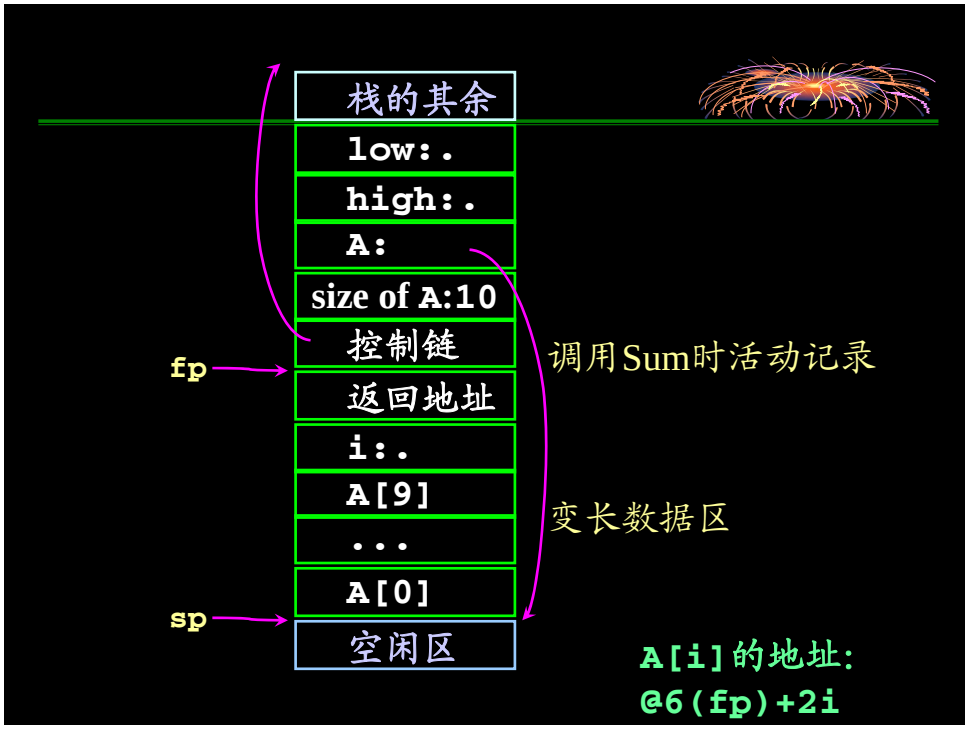
```
printf("%d%s%c",n,prompt,ch);
printf("Hollo World\n");
```

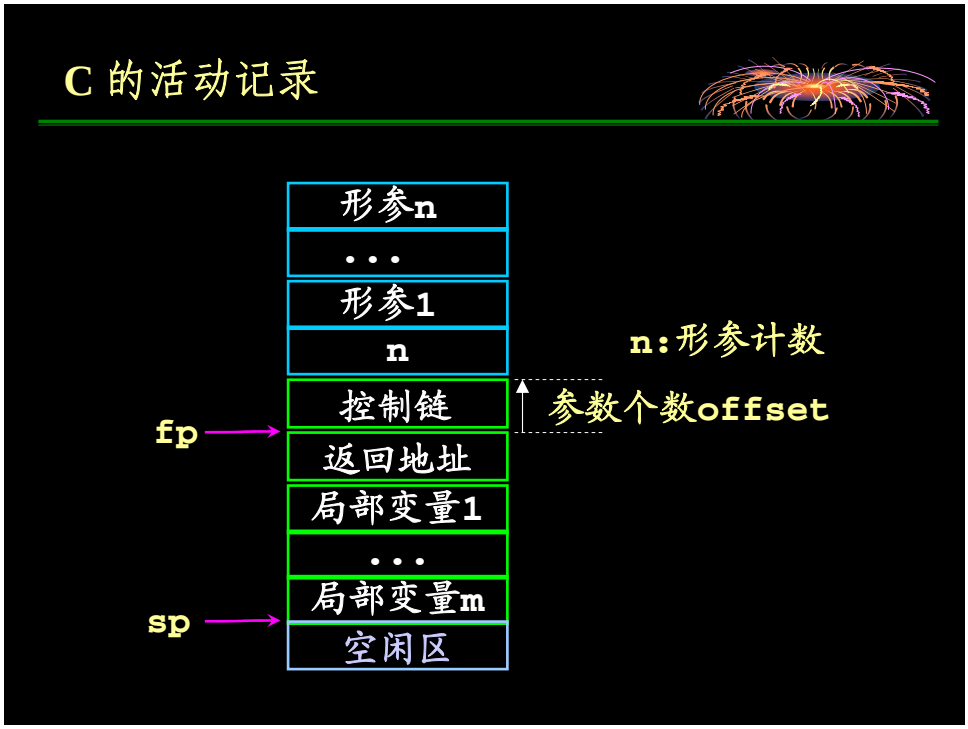
- 处理方式
  - 实参按照相反次序进栈；
  - 第一个参数位置放参数个数计数；
  - 从第二个参数位置开始放实在参数；
  - 参数个数所在单元：+4(fp)

处理变长数据

- 数组形参的大小每次调用时不一样
- 局部数组变量的大小每次调用时不一样

```
type Int-vector is
     array(INTEGER rang<>)of INTEGER;
procedure Sum(low,high:INTEGER;
              A:Int_vector)return INTEGER
is
  temp: Int_Array(low..high);
begin …
end Sum;
```


栈的其余
low:.
high:.
A:
size of A:10
控制链
fp
返回地址
i:.
A[9]
...
A[0]
sp
空闲区
调用Sum时活动记录
变长数据区
A[i]的地址:
@6(fp)+2i

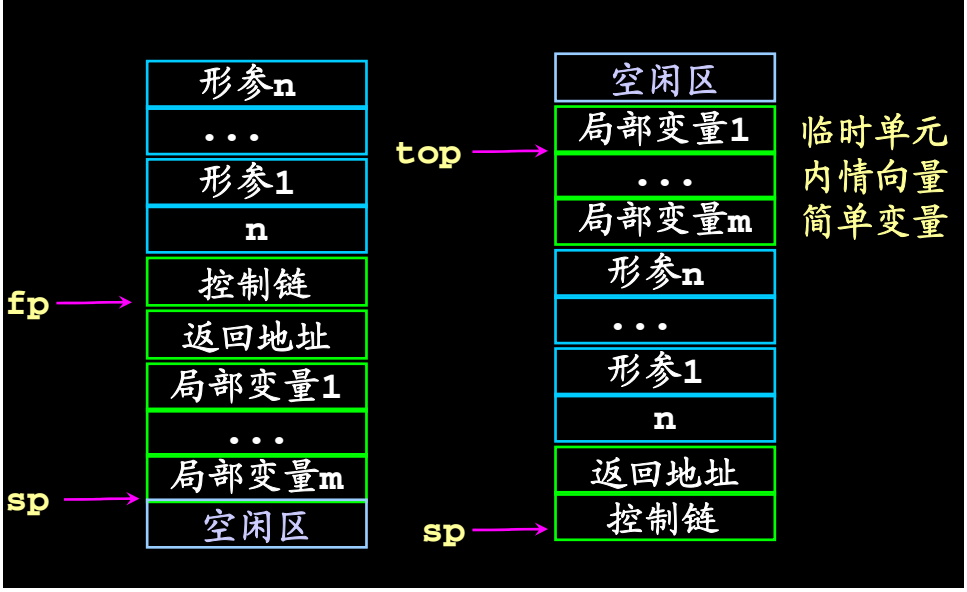

C 的活动记录
形参n
...
形参1
n
控制链
fp
返回地址
局部变量1
...
局部变量m
sp
空闲区
n:形参计数
参数个数offset

过程调用的四元式序列

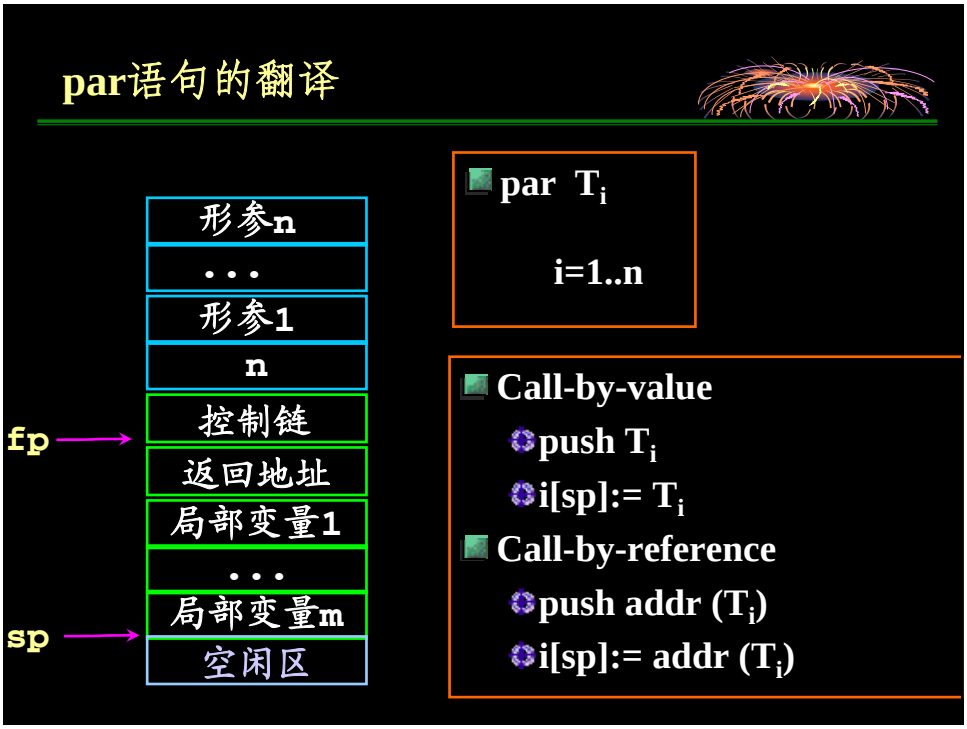
- par T_1
- par T_2
- .
- .
- .
- par T_n
- call P, n

- par T_n
- par T_{n-1}
- .
- .
- .
- par T_1
- call P, n

两种活动记录格式



par语句的翻译



call语句的翻译


return语句的翻译
return (E)
形参n
...
形参1
n
控制链
返回地址
局部变量1
...
局部变量m
空闲区
fp
sp
将E值放在结果寄存器中
x:=-1[fp]
y:=1[fp]+2
sp:=fp+y
fp:=0[fp]
UJ 0[x]


教材上的方法
par
(i+3)[top]:=Ti i=1..n
1[top]:=sp
call
3[top]:=n
JSR P
P
sp:=top+1
1[sp]:=返回地址
top:=top+L
top:=sp-1
return
sp:=0[sp]
x:=2[top]
UJ 0[x]
top
sp
空闲区
局部变量1
...
局部变量m
形参n
...
形参1
n
返回地址
控制链

9.5 嵌套过程语言的栈式实现

- 带有局部过程

```
procedure P(..){
  procedure Q(...){
    procedure R(..){
      .. x ..
    }
  }
}
```

- 需要处理词法上的嵌套关系，以决定名字的作用域

两种栈式实现方式

- 目标
  - 过程 R 运行时必须知道它的所有外层过程的最新活动纪录的地址
- 方法
  - 跟踪每个外层过程的最新活动纪录的位置

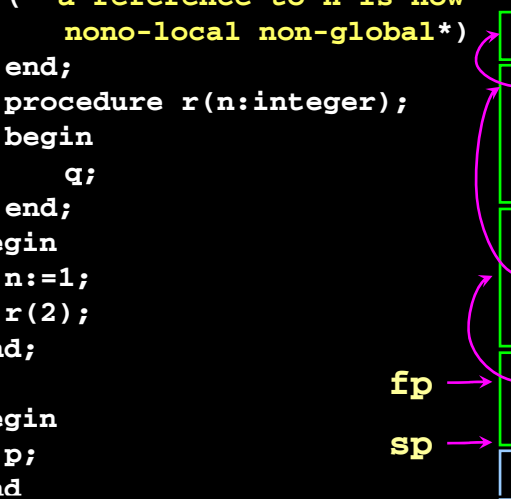
R 的现行活动纪录的地址
Q 的最新活动纪录的地址
P 的最新活动纪录的地址

1. 采用静态链表示

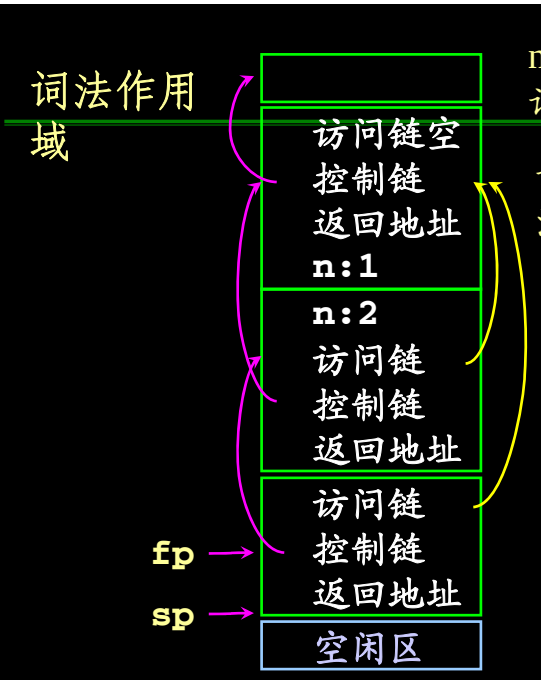
2. 嵌套层次显示表 DISPLAY 和活动纪录

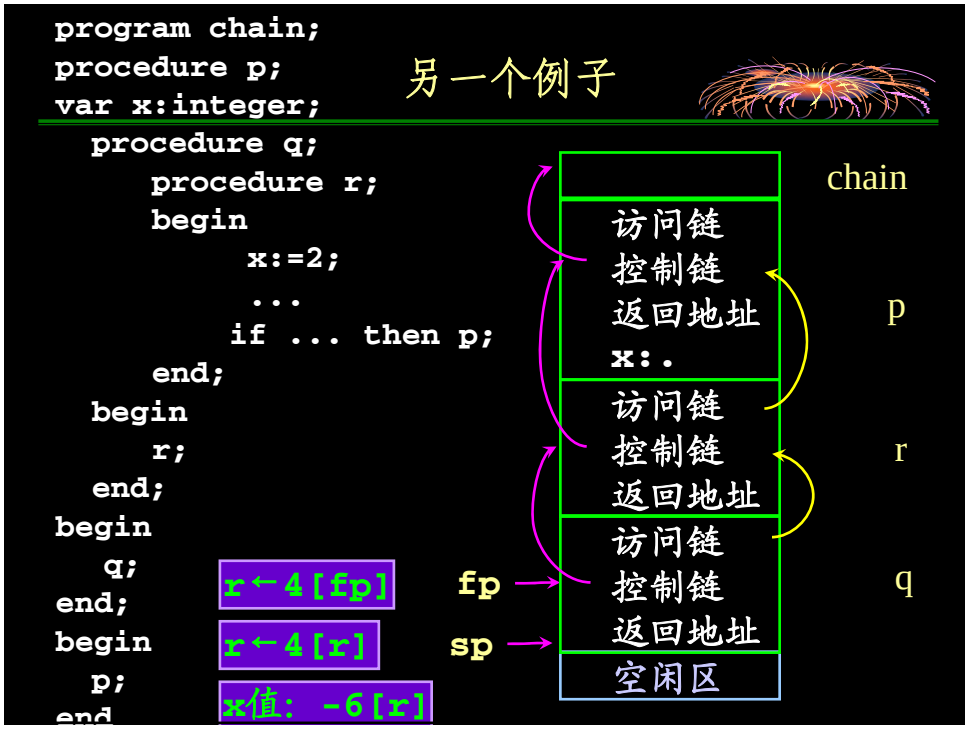
为什么采用静态链？

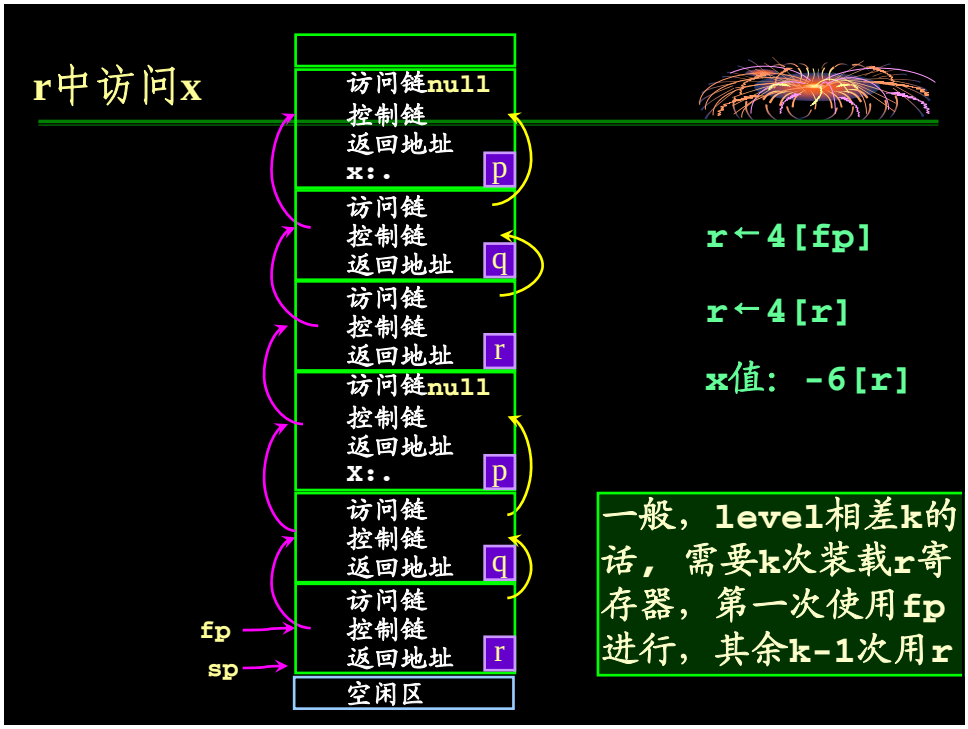
```
program nonLocalRef;
procedure p;
var n:integer;
  procedure q;
  begin
  (* a reference to n is now
     nono-local non-global*)
  end;
  procedure r(n:integer);
  begin
     q;
  end;
begin
  n:=1;
  r(2);
end;

begin
  p;
end
```



词法作用域



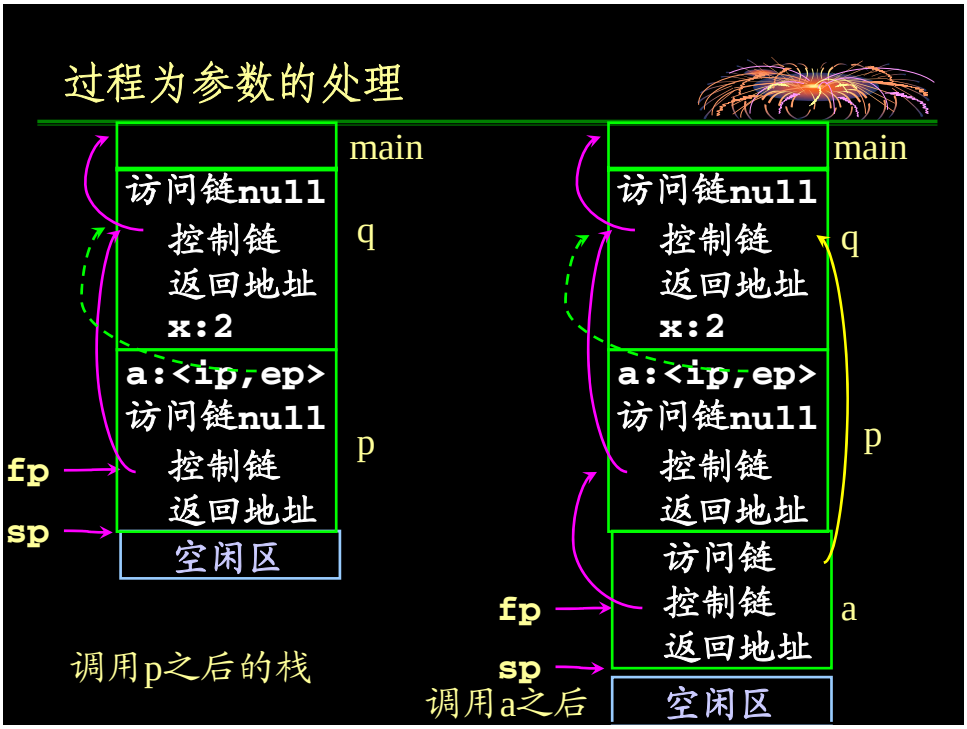

program chain;
procedure p;
var x:integer;
procedure q;
procedure r;
begin
x:=2;
...
if ... then p;
end;
begin
r;
end;
begin
q;
end;
begin
p;
另一个例子
chain
访问链
控制链
返回地址
x:.
p
访问链
控制链
返回地址
r
访问链
控制链
返回地址
q
空闲区
fp
sp
r←4[fp]
r←4[r]
x值: -6[r]


r中访问x
访问链null
控制链
返回地址
x:.
p
访问链
控制链
返回地址
q
访问链
控制链
返回地址
r
访问链null
控制链
返回地址
x:.
p
访问链
控制链
返回地址
q
访问链
控制链
返回地址
r
空闲区
fp
sp
r←4[fp]
r←4[r]
x值: -6[r]
一般，level相差k的话，需要k次装载r寄存器，第一次使用fp进行，其余k-1次用r

过程为参数

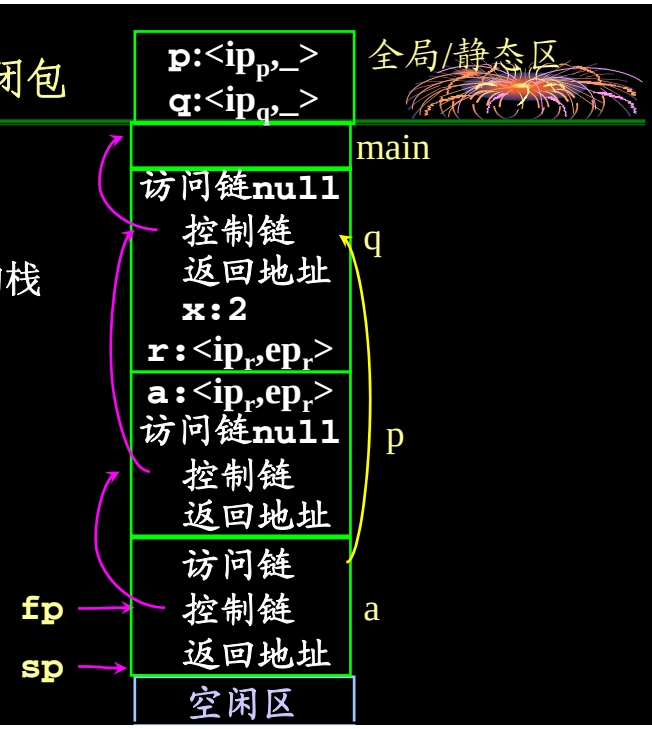
```
program closureEx(output);
procedure p(procedure a);
begin
  a;
end;
procedure q;
var x:integer;
  procedure r;
  begin
     writeln(x);
  end;
begin
  x:=2;
  p(r);
end;
begin
  q;
end.
```

过程为参数的处理



所有过程为闭包

- 调用a之后的栈



嵌套层次

- 对于自由变量的访问要沿着静态链依次找到定义该变量的活动记录中，一般效率较低
- 间接访问操作的次数 = 当前过程的嵌套层次 - 定义过程的嵌套层次

- 定义过程的嵌套层次：
  - 主程序嵌套层次为0
  - 在语法语义分析阶段，每进入一个过程，当前的嵌套层次加1，退出时减1

例.嵌套层次

- level 0
  - chain
- level 1
  - p: x
- level 2
  - q
- level 3
  - r

```
program chain;
procedure p;
var x:integer;
  procedure q;
     procedure r;
     begin
          x:=2;
          ...
         if ... then p;
     end;
  begin
     r;
  end;
begin
   q;
end;
begin
   p;
end
```

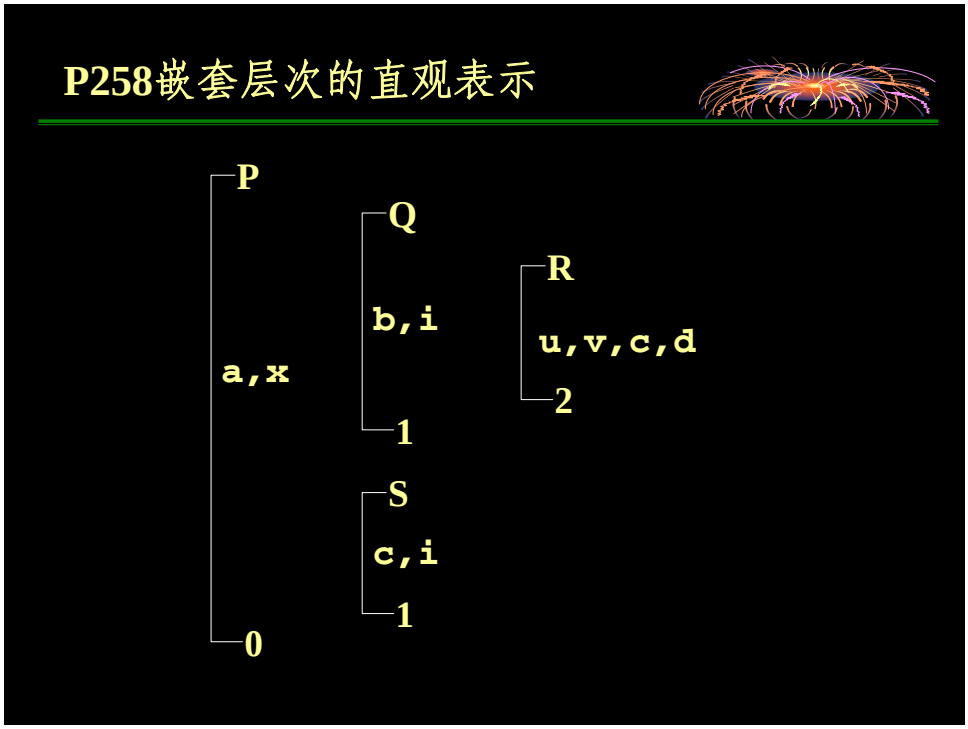
在r活动记录（level 3）中访问自由变量x需要两次间接访问寄存器找到p的活动记录(level 1)。

例. P258嵌套层次及变量作用域

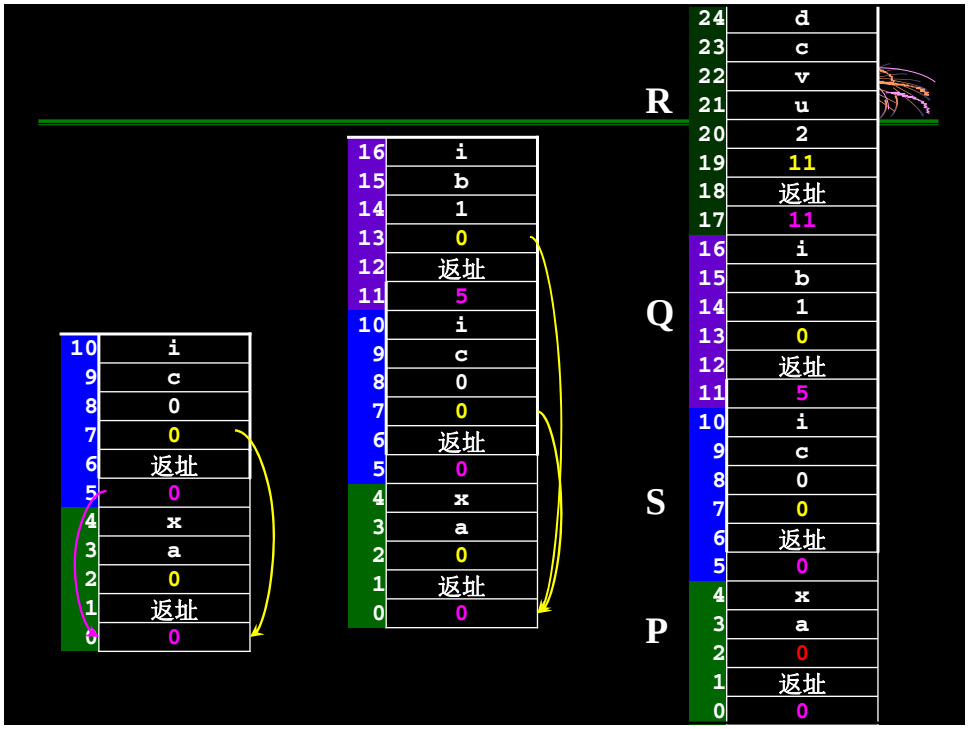
```
program P;
  var a,x:integer;
  procedure Q (b:integer);
    var i :integer;
    procedure R(u:integer; var
    v:integer);
      var c,d:integer;
      begin
      if u=1 then R(u+1,v)…
      v:=(a+c)*(b-d);
      …end;
    begin…
    R(1,x); …end;
```

```
  procedure S;
    var c:integer;
    begin
    a:=1;
    Q(c);
    …end
begin
a:=0;
S; …end
```

P258嵌套层次的直观表示

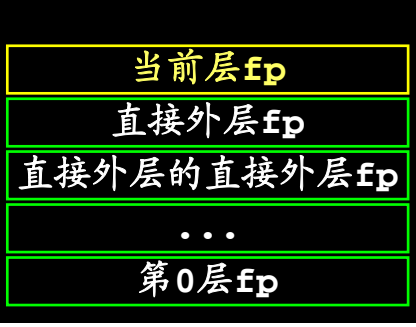


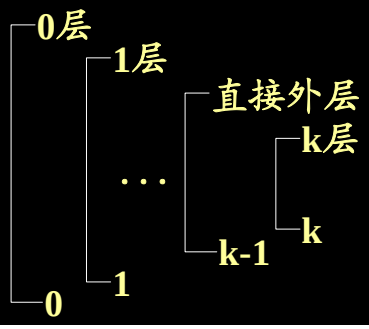
带有静态链的内情向量的结构



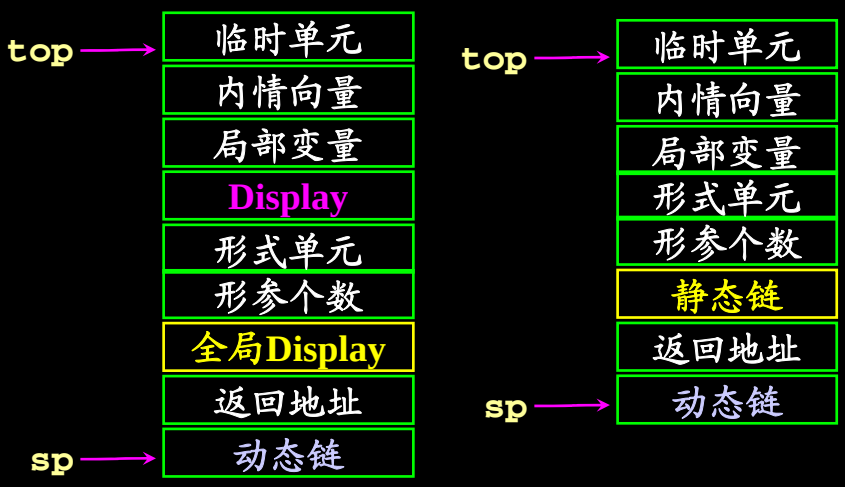
9.5.2 采用嵌套层次显示表的活动记录

- 活动记录中包含一个DISPLAY表
- 如果当前过程的嵌套层次为k则DISPLAY表有k+1个单元
- 依次从0层、1层一直到当前层，记录了当前过程的词法嵌套过程的最新活动记录位置





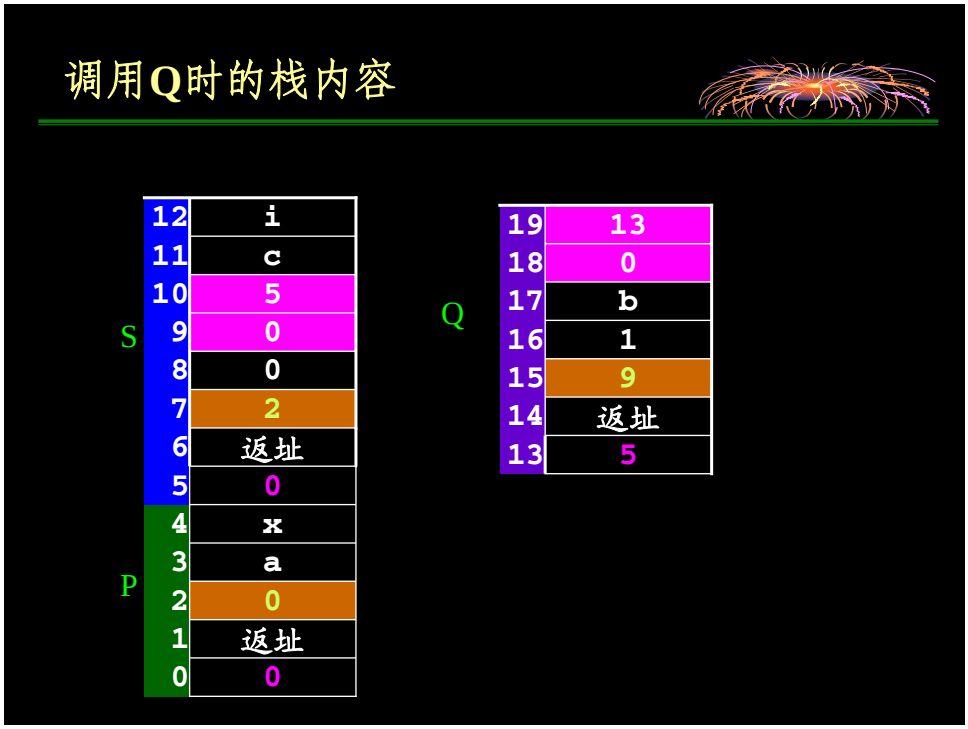
引入 DISPLAY 表以后的活动记录



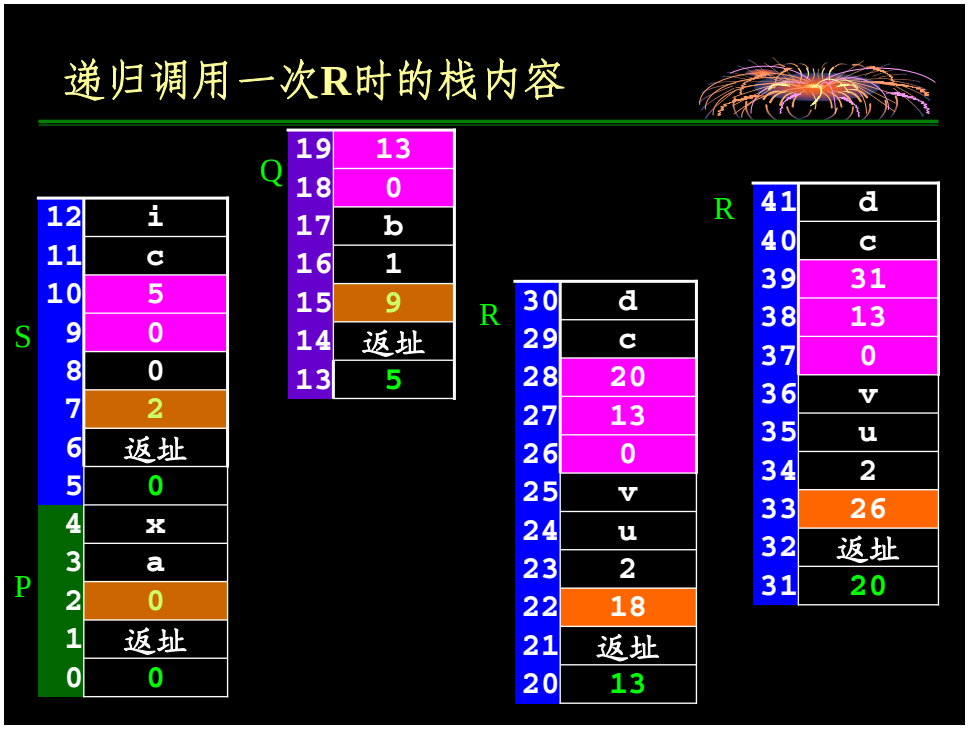
在当前过程中对非局部量的访问

- 非局部量的访问地址
  - 绝对地址 = display[静态层数]+相对地址
  - 静态层数指定义那个非局部量的层
- 访问指令
  - LD r1, (d+k)[sp]
  - LD r2, x[r1]
  - d为display的偏移量
  - k为引用的外层过程层数
  - x为引用变量的偏移量

调用Q时的栈内容

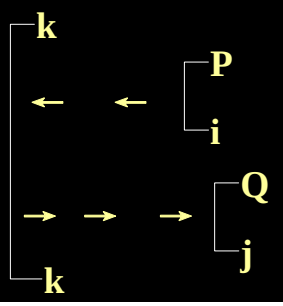


递归调用一次R时的栈内容



如何建立DISPLAY表

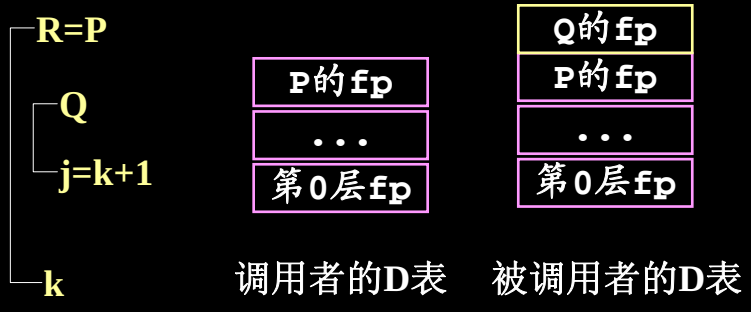
- 过程P调用过程Q，过程Q的直接外层为R，那么如何建立Q的DIPSLAY表？
  - R就是P
  - R是P的外层
  - 其它情况P看不见Q，故不可能调用Q



k<=i, k<=j

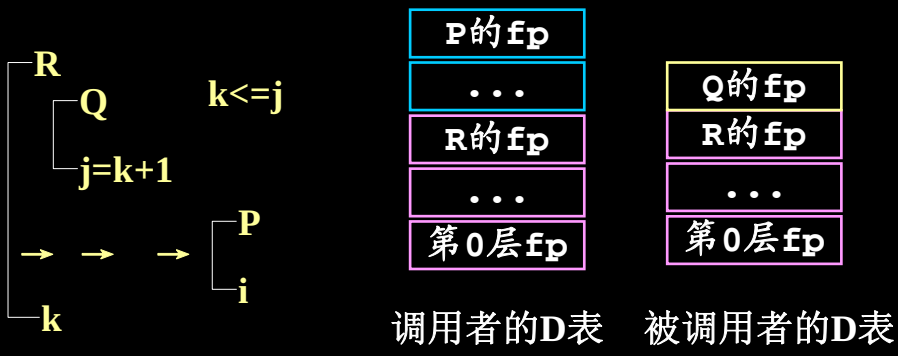
由调用者的D表构造被调用者的D表情形1

- R就是P ： P的DISLAY表+指向Q的指针



由调用者的D表构造被调用者的D表情形2

- R是P的外层
  - P的D表中的前k+1个 + 指向Q的指针



参数传递的实现

- par T, T为数组
- par T, T为过程
- par T, T为标号（略）

par T, T为数组

- 传送 T 的内情向量
  - 实数组的维数一致性和体积相容性进行动态检查
- 传送 T 的首地址

par T, T为过程

- P把R作为实参调用Q，在Q体中通过引用形参调用R，那么在调用R时如何建立它的DISPLAY表？
  - P能够引用R
  - P就是R的直接外层
  - P嵌套着R的直接外层
  - Q的全局DISPLAY单元存放着P的D表入口
- P体中Call Q(R) 时，传递给形参的是闭包
  - <R代码入口，当前过程D表入口>
- Q体中Call A时，A的值为上述闭包
  - 采用闭包中的第二项建立R的DISPLAY表

9.6 堆式动态存储分配

- 显式的动态申请
  - Pascal的new和dispose语句
  - C的malloc和free语句
  - Java的new语句
- 显式的动态释放
  - Pascal的new和dispose语句
  - C的malloc和free语句
- 隐式的动态释放
  - Java的Garbage Collection
  - Lisp的Garbage Collection

堆式动态存储分配的实现

- 定长块管理
- 变长块管理
  - 首次适应法
  - 最佳适应法
  - 最差适应法

变长块管理中空闲块选择算法的比较

- 最佳适应法
  - 请求分配的内存块大小范围较广的情况
  - 有可能产生很小的碎片
  - 保留更大的块以满足大尺寸申请
  - 分配、释放均要查链表
- 最差适应法
  - 请求分配的内存块大小范围较窄的情况
  - 保持块由大到小次序
- 首次适应法
  - 有随机性，介于二者之间
  - 保持块由小到大次序

隐式存储回收(Garbage Collection)

- **Mark-Sweep方法**
- **Stop-Copy方法**
- 实时的方法

作业

- P269-271
  - 4, 5, 6, 9