Run-time Environments - 2

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is run-time support? (in part 1)
- Parameter passing methods (in part 1)
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

Code and Data Area in Memory

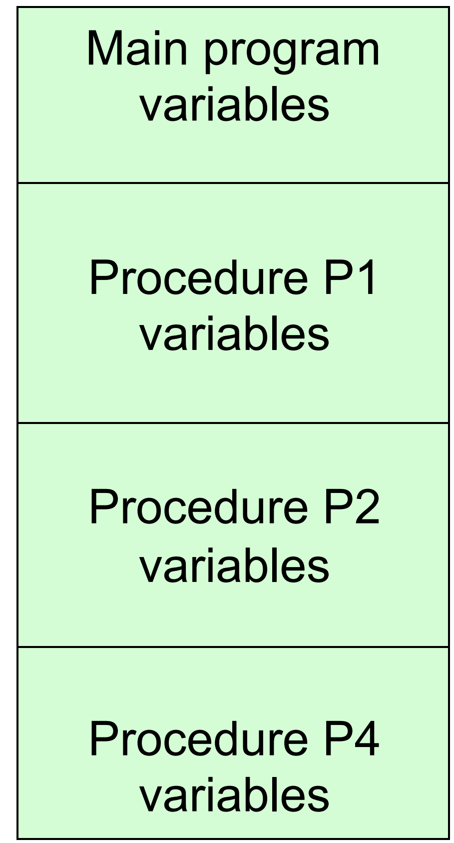
- Most programming languages distinguish between code and data
- Code consists of only machine instructions and normally does not have embedded data
  - Code area normally does not grow or shrink in size as execution proceeds
    - Unless code is loaded dynamically or code is produced dynamically
      - As in Java – dynamic loading of classes or producing classes and instantiating them dynamically through reflection
  - Memory area can be allocated to code statically
    - We will not consider Java further in this lecture
- Data area of a program may grow or shrink in size during execution

Static Versus Dynamic Storage Allocation

- Static allocation
  - Compiler makes the decision regarding storage allocation by looking only at the program text
- Dynamic allocation
  - Storage allocation decisions are made only while the program is running
  - Stack allocation
    - Names local to a procedure are allocated space on a stack
  - Heap allocation
    - Used for data that may live even after a procedure call returns
    - Ex: dynamic data structures such as symbol tables
    - Requires memory manager with garbage collection

Static Data Storage Allocation

- Compiler allocates space for all variables (local and global) of all procedures at compile time
  - No stack/heap allocation; no overheads
  - Ex: Fortran IV and Fortran 77
  - Variable access is fast since addresses are known at compile time
  - No recursion

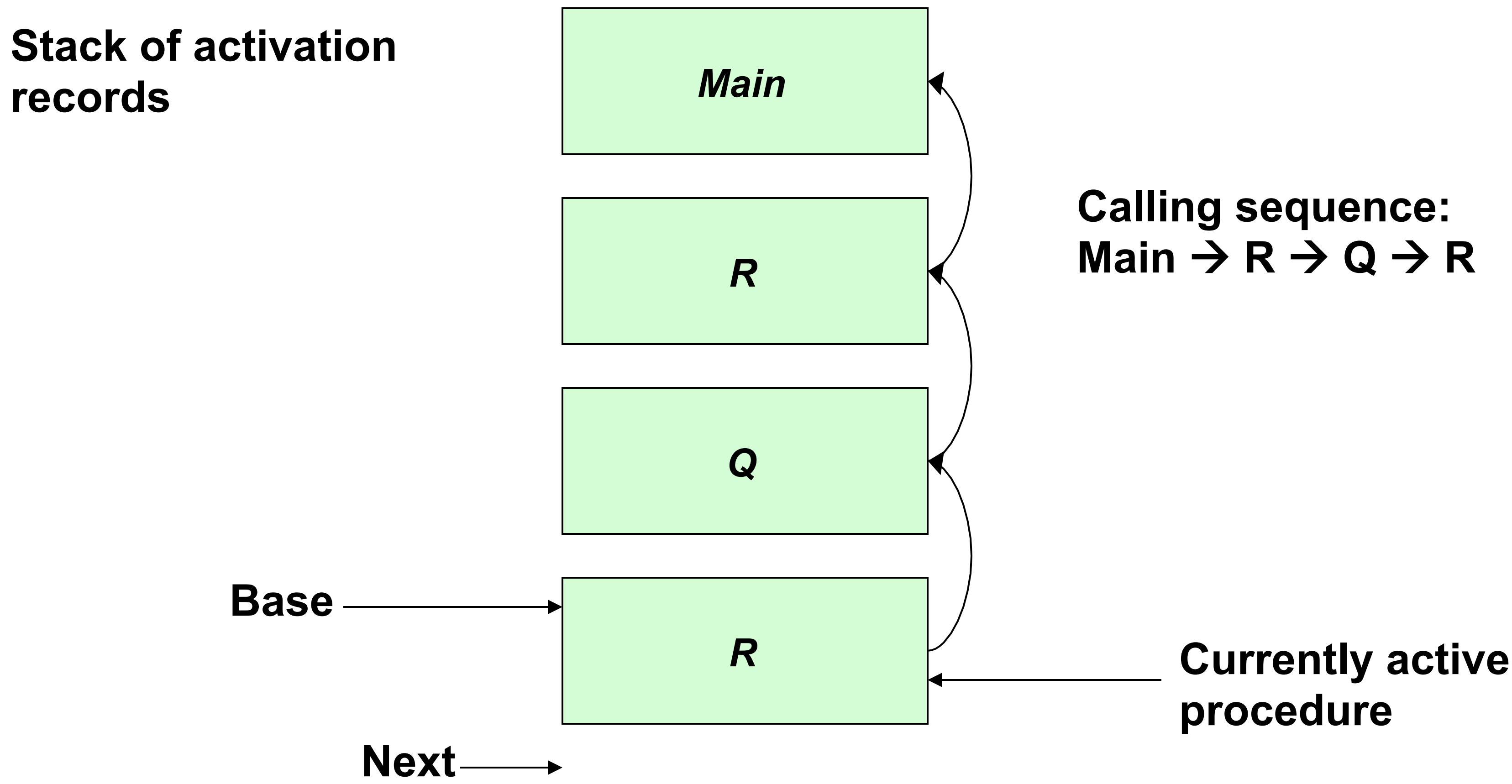


Main memory

Dynamic Data Storage Allocation

- Compiler allocates space only for global variables at compile time
- Space for variables of procedures will be allocated at run-time
  - Stack/heap allocation
  - Ex: C, C++, Java, Fortran 8/9
  - Variable access is slow (compared to static allocation) since addresses are accessed through the stack/heap pointer
  - Recursion can be implemened

Dynamic Stack Storage Allocation



Activation Record Structure

Return address
Static and Dynamic links (also called Access and Control link resp.)
(Address of) function result
Actual parameters
Local variables
Temporaries
Saved machine status
Space for local arrays

Note:

The position of the fields of the act. record as shown are only notional.

Implementations can choose different orders; e.g., function result could be after local var.

Variable Storage Offset Computation

- The compiler should compute
  - the offsets at which variables and constants will be stored in the activation record (AR)
- These offsets will be with respect to the pointer pointing to the beginning of the AR
- Variables are usually stored in the AR in the declaration order
- Offsets can be easily computed while performing semantic analysis of declarations

Overlapped Variable Storage for Blocks in C

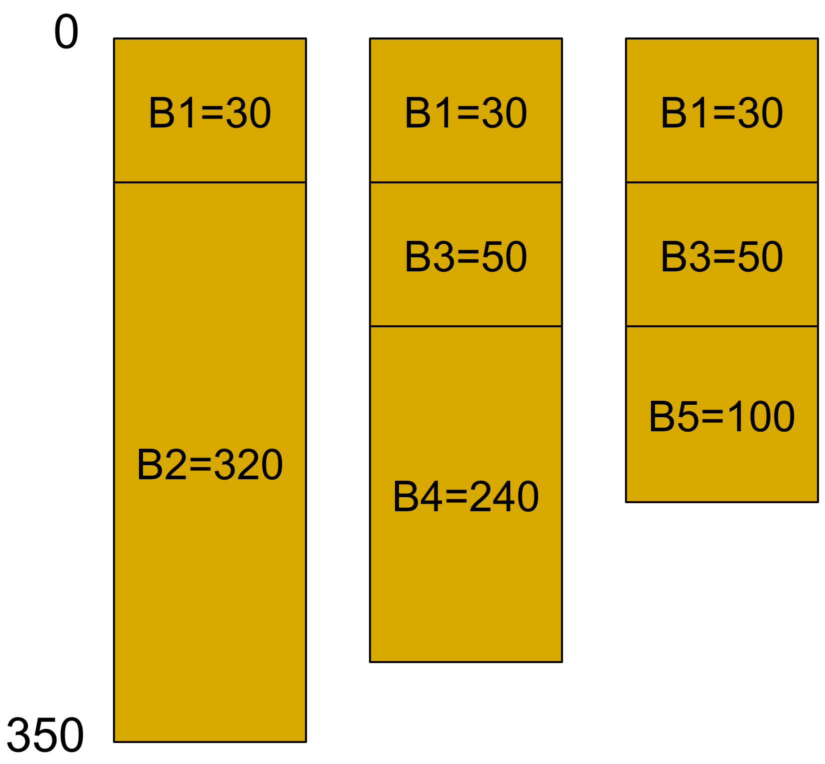
```
Storage required =
B1+max(B2,(B3+max(B4,B5))) =
30+max(320,(50+max(240,100))) =
        30+max(320, (50+240)) =
        30+max(320,290) = 350
```

```
int example(int p1, int p2)
B1 { a,b,c; /* sizes - 10,10,10;
               offsets 0,10,20 */
     ...
     B2 { d,e,f; /* sizes - 100, 180, 40;
                    offsets 30, 130, 310 */
          ...}
     B3 { g,h,i; /* sizes - 20,20,10;
                    offsets 30, 50, 70 */
          ...
          B4 { j,k,l; /* sizes - 70, 150, 20;
                         offsets 80, 150, 300 */
               ... }
          B5 { m,n,p; /* sizes - 20, 50, 30;
                         offsets 80, 100, 150 */
               ... }
     }
}
```

Overlapped storage

Overlapped storage

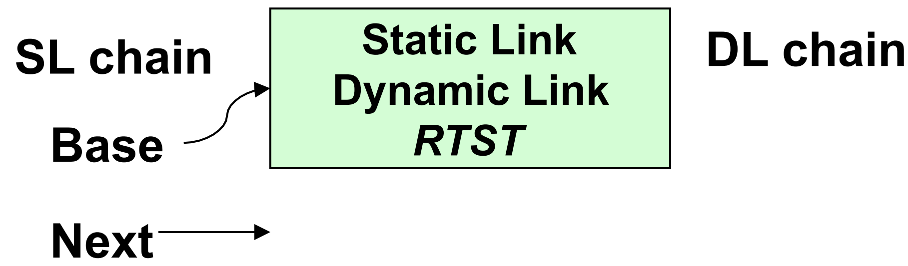
Overlapped Variable Storage for Blocks in C (Ex.)



Storage required =
B1+max(B2,(B3+max(B4,B5))) =
30+max(320,(50+max(240,100))) =
30+max(320, (50+240)) =
30+max(320,290) = 350

Allocation of Activation Records (nested procedures)

```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```

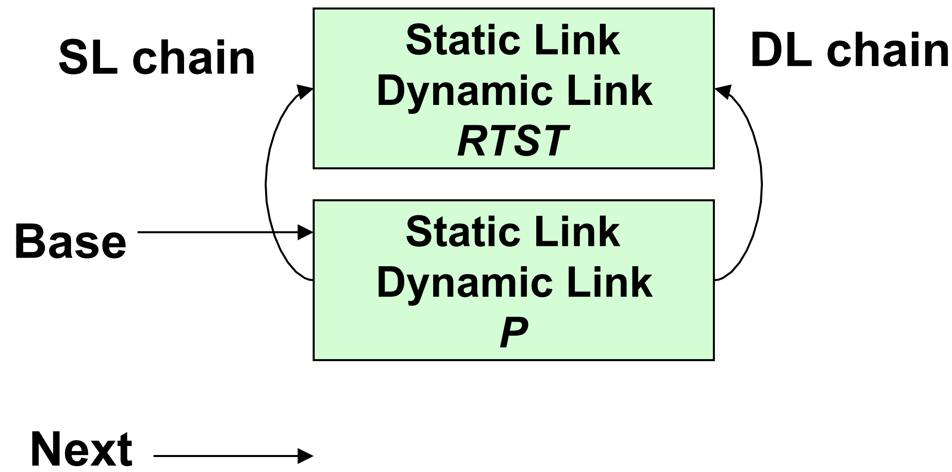


Activation records are created at procedure entry time and destroyed at procedure exit time

RTST -> P -> R -> Q -> R

Allocation of Activation Records (contd.)

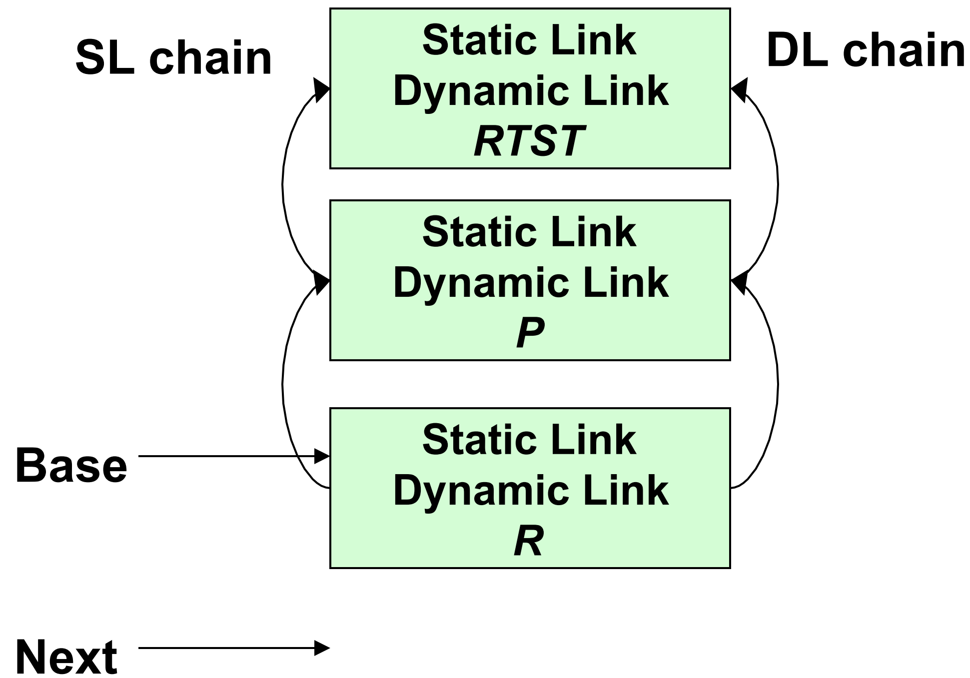
program *RTST*;
  procedure *P*;
    procedure *Q*;
      begin *R*; end
    procedure *R*;
      begin *Q*; end
  begin *R*; end
begin *P*; end



RTST -> **P** -> R -> Q -> R

Allocation of Activation Records (contd.)

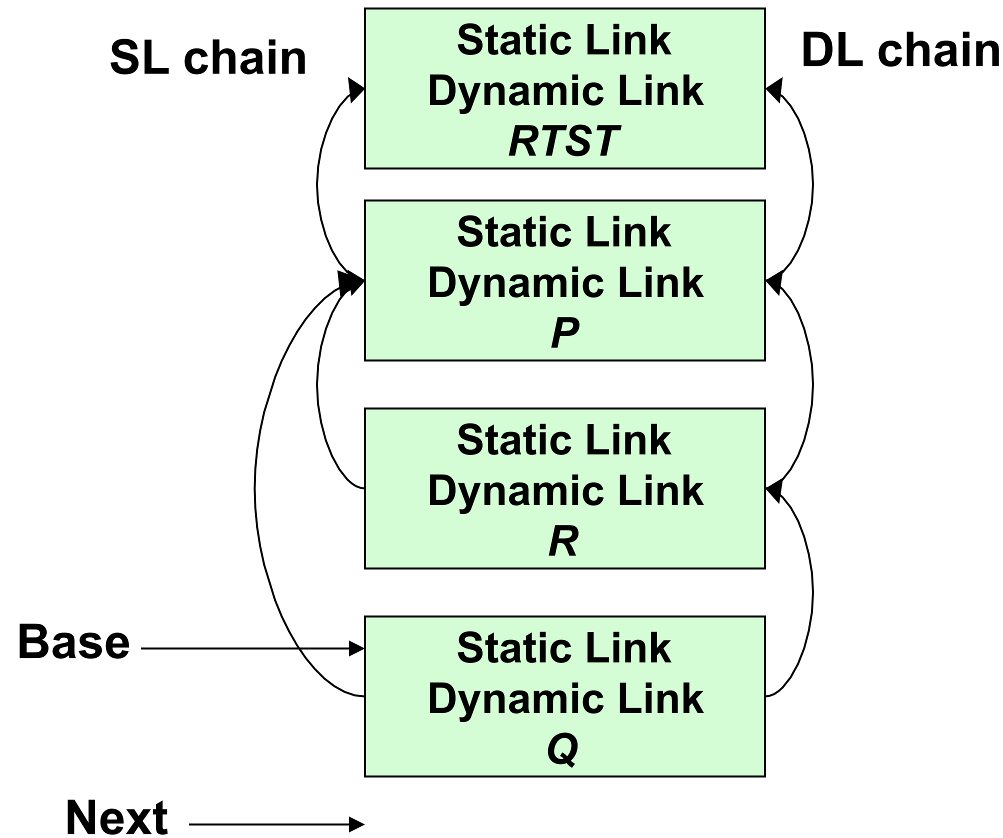
program *RTST*;
  procedure *P*;
    procedure *Q*;
      begin *R*; end
    procedure *R*;
      begin *Q*; end
  begin *R*; end
begin *P*; end



RTST -> **P** -> **R** -> Q -> R

Allocation of Activation Records (contd.)

program *RTST*;
  procedure *P*;
    procedure *Q*;
      begin *R*; end
    procedure *R*;
      begin *Q*; end
  begin *R*; end
begin *P*; end

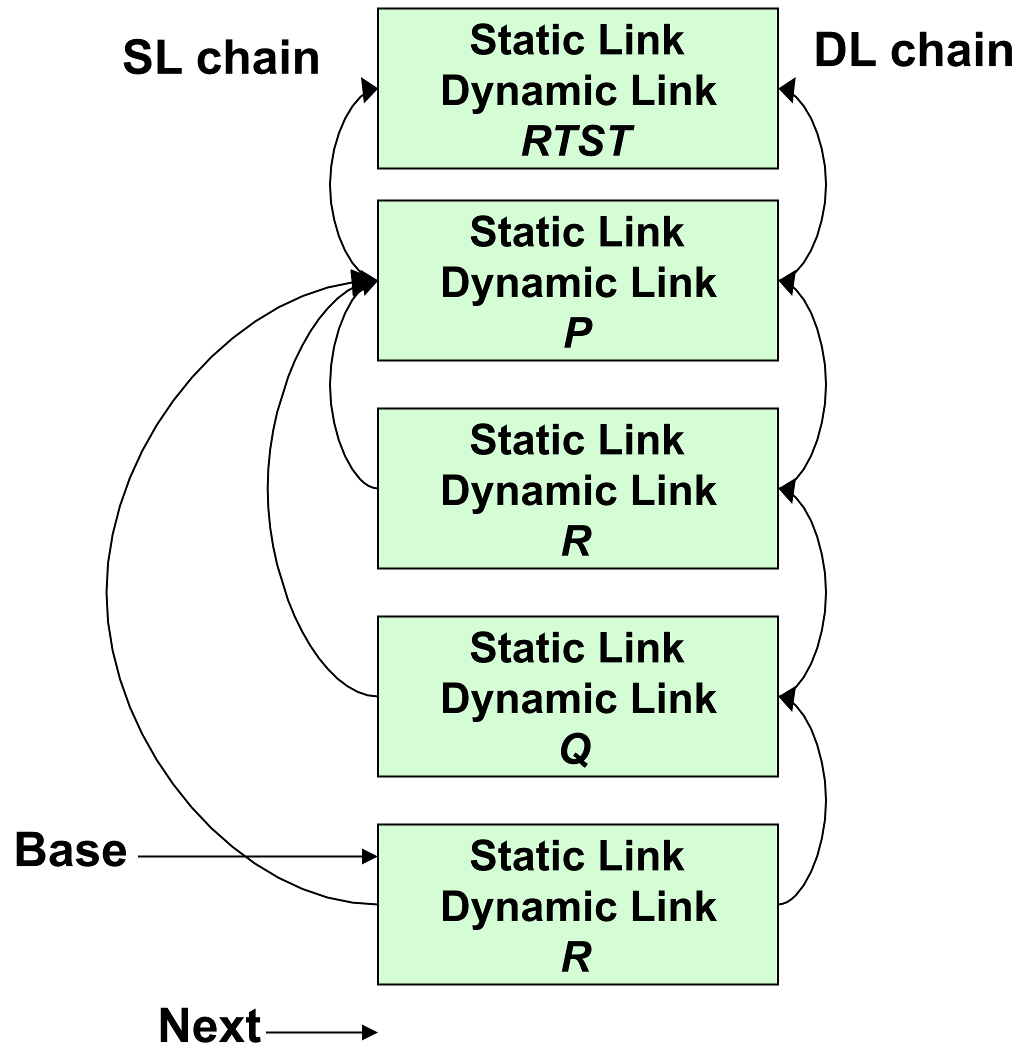


RTST -> **P** -> **R** -> **Q** -> R

Allocation of Activation Records (contd.)

1 program *RTST*;
2 procedure *P*;
3 procedure *Q*;
begin *R*; end
3 procedure *R*;
begin *Q*; end
begin *R*; end
begin *P*; end

RTST^1 -> P^2 -> R^3 -> Q^3 -> R^3



Allocation of Activation Records (contd.)

Skip L_1-L_2+1 records starting from the caller's AR and establish the static link to the AR reached

L_1 – caller, L_2 – Callee

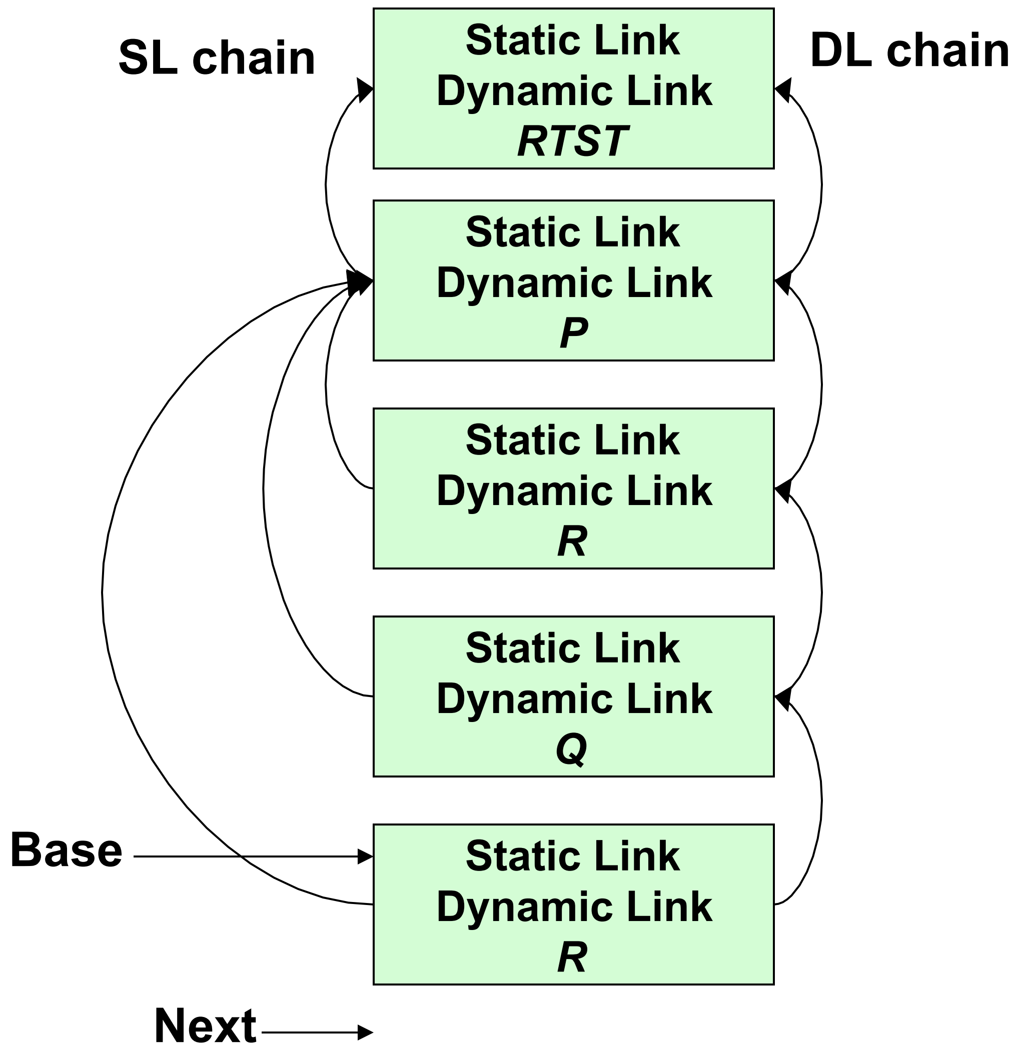
$RTST^1$ -> P^2 -> R^3 -> Q^3 -> R^3

Ex: Consider P^2 -> R^3

2-3+1=0; hence the SL of R points to P

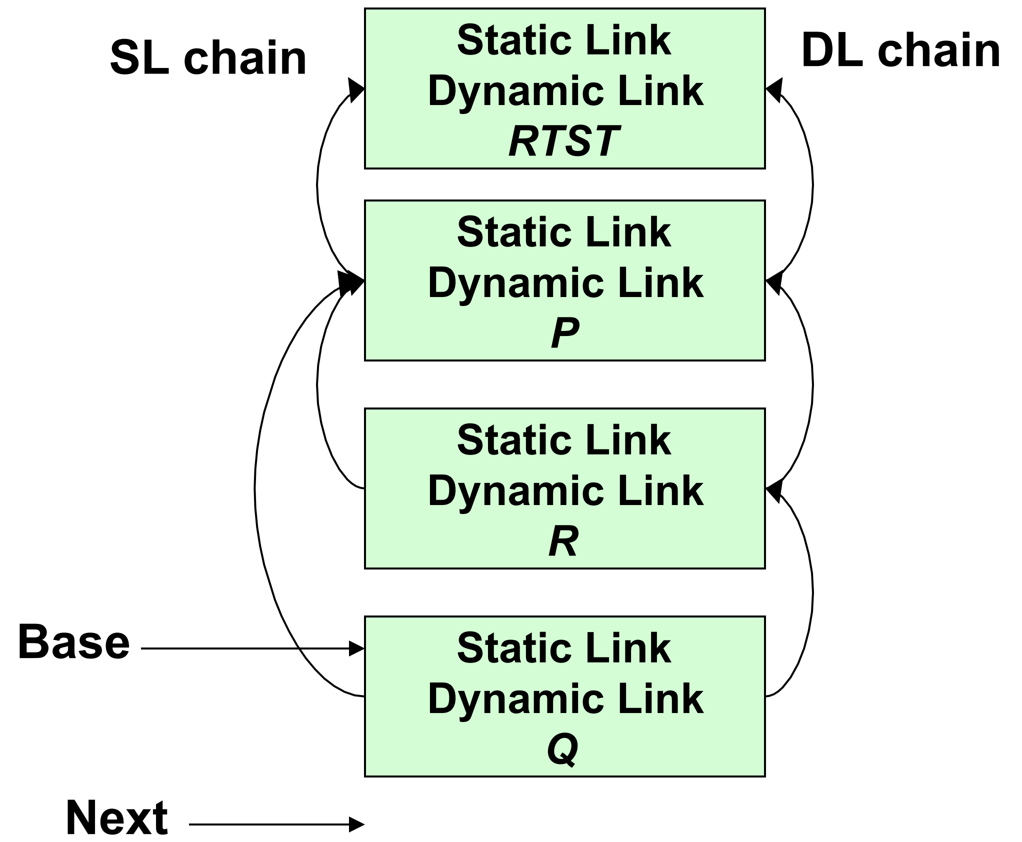
Consider R^3 -> Q^3

3-3+1=1; hence skipping one link starting from R, we get P; SL of Q points to P



Allocation of Activation Records (contd.)

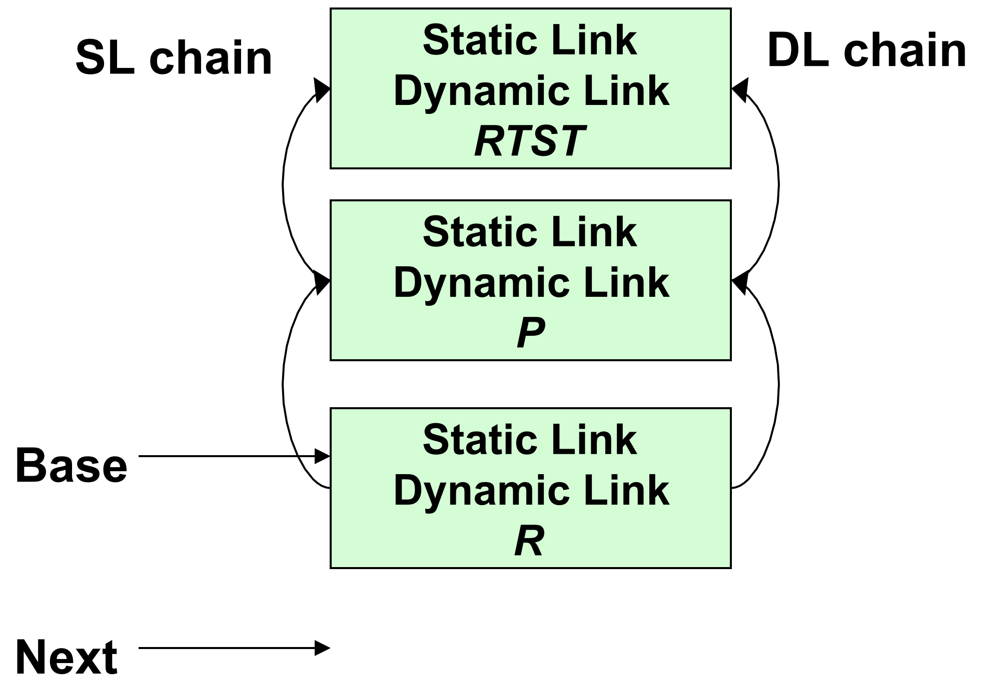
program *RTST*;
  procedure *P*;
    procedure *Q*;
      begin *R*; end
    procedure *R*;
      begin *Q*; end
  begin *R*; end
begin *P*; end



RTST -> **P** -> **R** -> **Q** <- R   **Return from R**

Allocation of Activation Records (contd.)

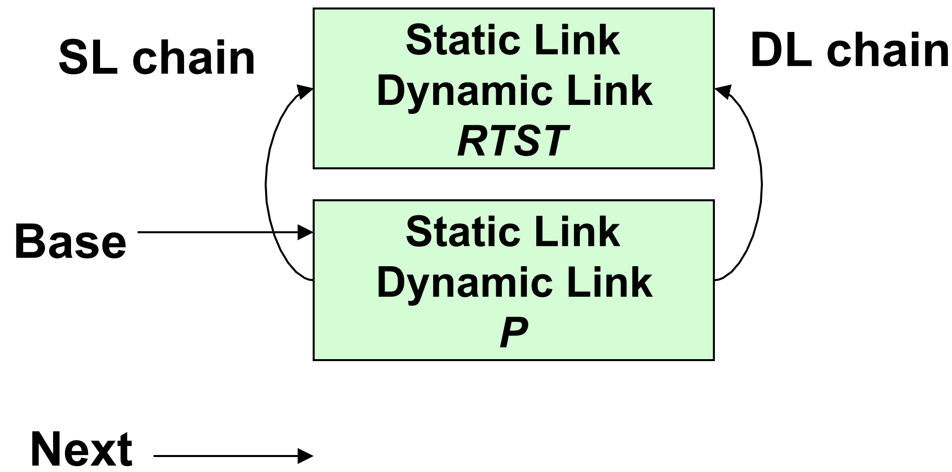
program *RTST*;
  procedure *P*;
    procedure *Q*;
      begin *R*; end
    procedure *R*;
      begin *Q*; end
  begin *R*; end
begin *P*; end



RTST -> **P** -> **R** <- Q   **Return from Q**

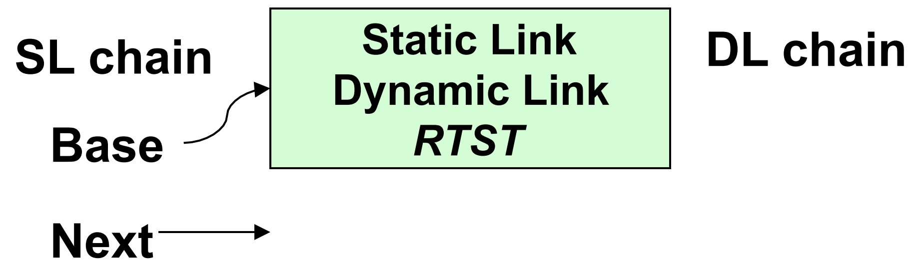
Allocation of Activation Records (contd.)

program *RTST*;
  procedure *P*;
    procedure *Q*;
      begin *R*; end
    procedure *R*;
      begin *Q*; end
  begin *R*; end
begin *P*; end



RTST -> **P** <- R   **Return from R**

Allocation of Activation Records (contd.)

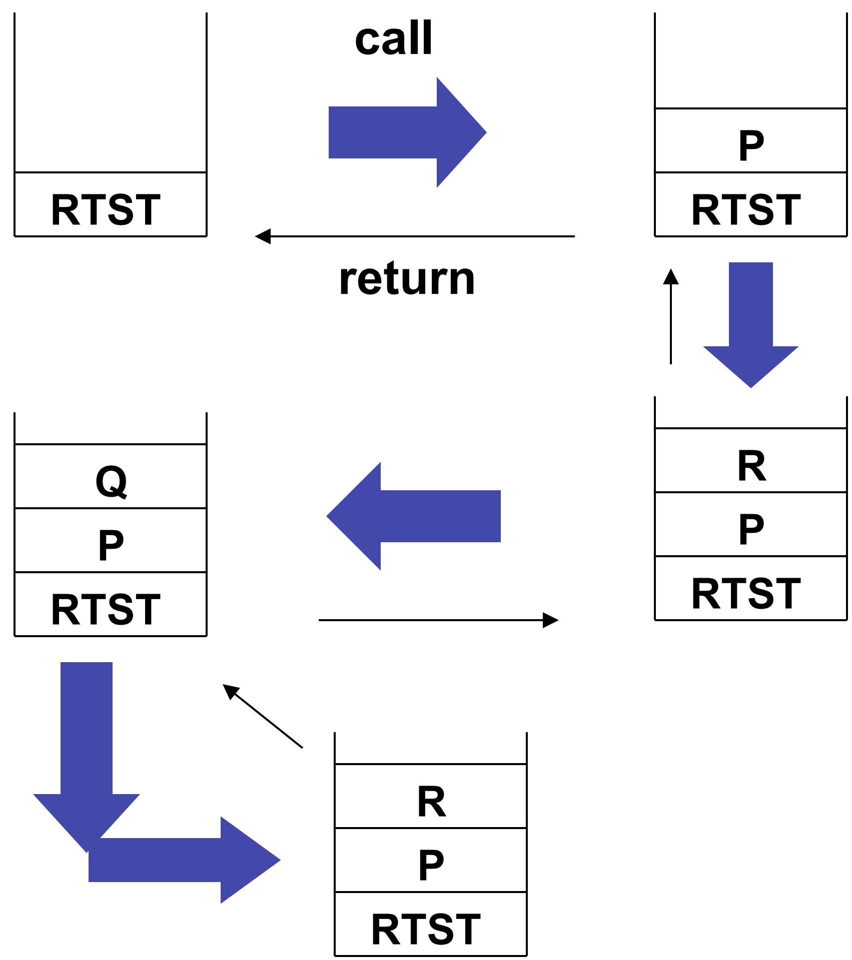


program *RTST*;
  procedure *P*;
    procedure *Q*;
      begin *R*; end
    procedure *R*;
      begin *Q*; end
  begin *R*; end
begin *P*; end

RTST <- P   **Return from P**

Display Stack of Activation Records

```
1  program RTST;
2    procedure P;
3      procedure Q;
         begin R; end
3      procedure R;
         begin Q; end
     begin R; end
   begin P; end
```



Pop L_1-L_2+1 records off the display of the caller and push the pointer to AR of callee (L_1 – caller, L_2 – Callee)

The popped pointers are stored in the AR of the caller and restored to the DISPLAY after the callee returns

Static Scope and Dynamic Scope

- *Static Scope*
  - A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text
  - Uses the *static* (unchanging) relationship between blocks in the program text
- *Dynamic Scope*
  - A global identifier refers to the identifier associated with the most recent activation record
  - Uses the actual sequence of calls that are executed in the *dynamic* (changing) execution of the program
- Both are identical as far as local variables are concerned

Static Scope and Dynamic Scope : An Example

```
int x = 1, y = 0;
int g(int z)
  { return x+z;}
int f(int y) {
    int x; x = y+1;
    return g(y*x);
}
y = f(3);
```

After the call to g,
Static scope: x = 1
Dynamic scope: x = 4

x	**1**	**outer block**
y	**0**	

y	**3**	**f(3)**
x	**4**	

z	**12**	**g(12)**

Stack of activation records after the call to *g*

Static Scope and Dynamic Scope: Another Example

```
float r = 0.25;
void show() { printf("%f",r); }
void small() {
   float r = 0.125; show();
}
int main (){
show(); small(); printf("\n");
show(); small(); printf("\n");
}
```

- Under static scoping, the output is

  0.25 0.25

  0.25 0.25

- Under dynamic scoping, the output is

  0.25 0.125

  0.25 0.125