# CC Lecture 12

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

# Runtime Environment
(topics covered so far)

- Parameter passing methods
- Static storage allocation
- Dynamic stack storage allocation
  - Activation record structure
  - Offset calculation for overlapped storage

# Allocation of nested procedure

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

- P is nested in RTST
- Q and R are nested in P
- Q and R are at same level
- Q calls R and R calls Q
- P calls R
- Main program RTST calls P

# Allocation of nested procedure

```
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 are destroyed at exit time
- How to access variables declared in various procedures?

# Allocation of nested procedure

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

- **Call sequence**

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

- Main program RTST cannot access variables of P,Q and R.
- P can access its own and main program variables but not of Q and R
- Q cannot access variables of R but can access variables of P and main
- R cannot access variables of Q but can access variables of P and main

# Allocation of nested procedure

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

- **Call sequence**

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

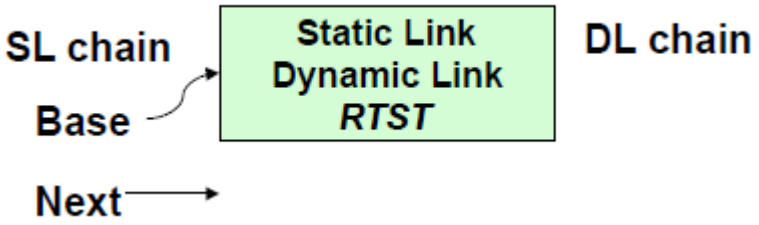
- When P is called, activation record of P is made
- Base pointer + offset can be used to access local variables of P
- But what about variables of main??
- Can base pointer be use in this case??

# Allocation of nested procedure

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

- **Call sequence**

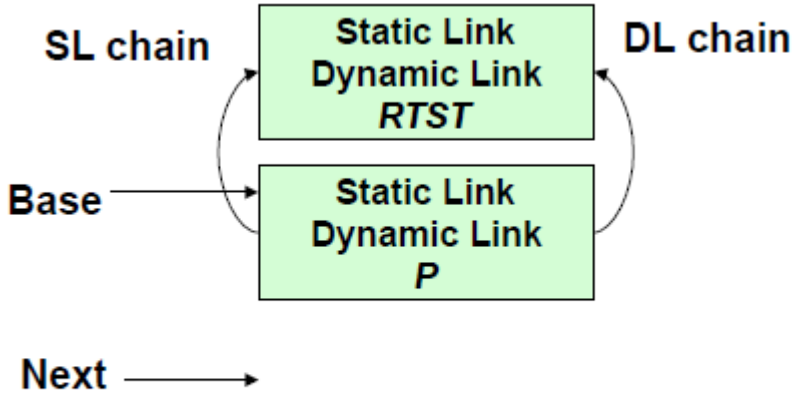
  **RTST -> P -> R -> Q -> R**




- The **DL chain** chains all the activation records in order to maintain a stack structure.
- To access the variables of RTST, the **SL field** of the activation record has to be put into a register, and the contents of that activation of that register will now point to the beginning of the activation record for RTST.
- Consider this particular value and then access the variables of RTST using the offset.

# Allocation of nested procedure

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




- **Call sequence**

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

For variables of RTST:
SL field of P → register → beginning of RTST + offset

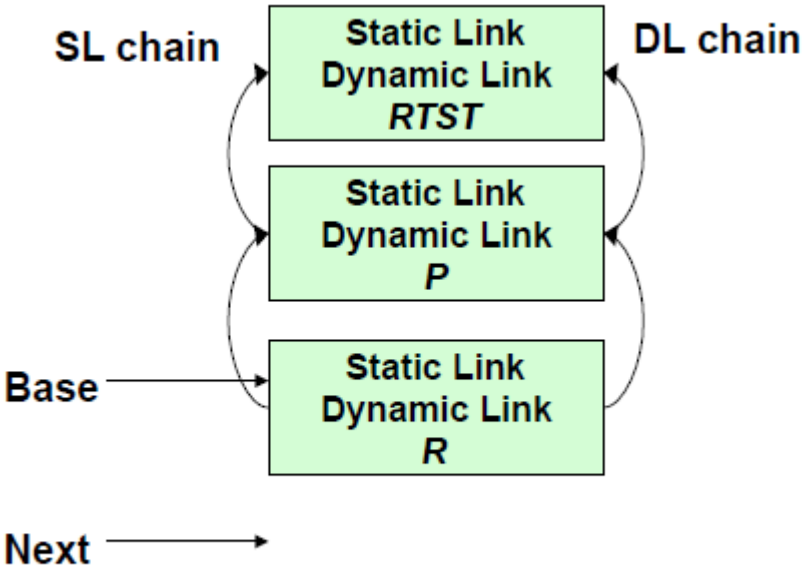
For variables of P:
Base is beginning of P + offset

# Allocation of nested procedure

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

- **Call sequence**

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




For variables of R: Base

For variables of P: use SL

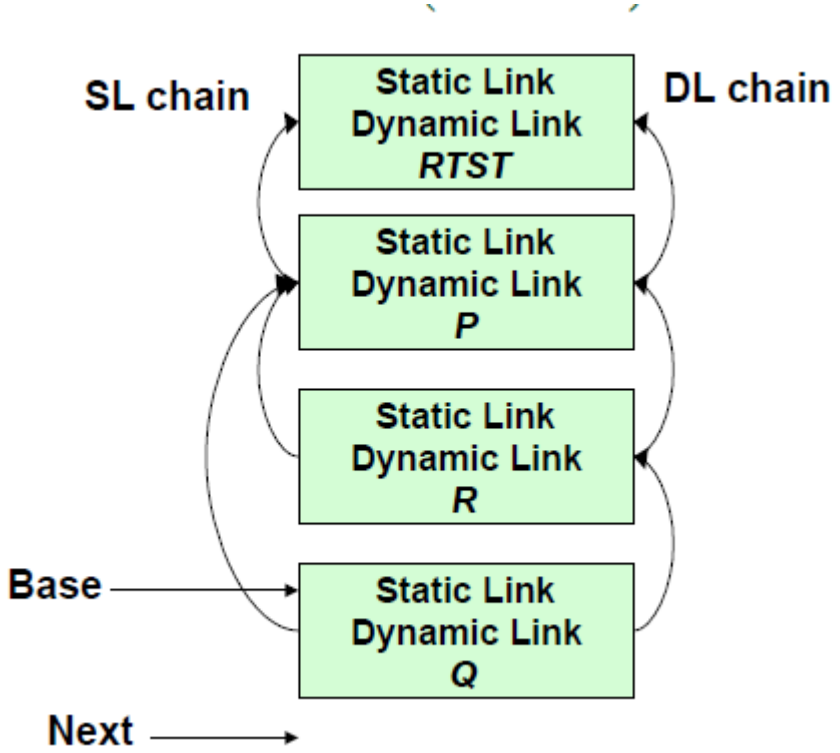
For variables of RTST: one more level of indirection using SL

# Allocation of nested procedure

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

- **Call sequence**

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




No static link from Q→R, as Q cannot access variables of R

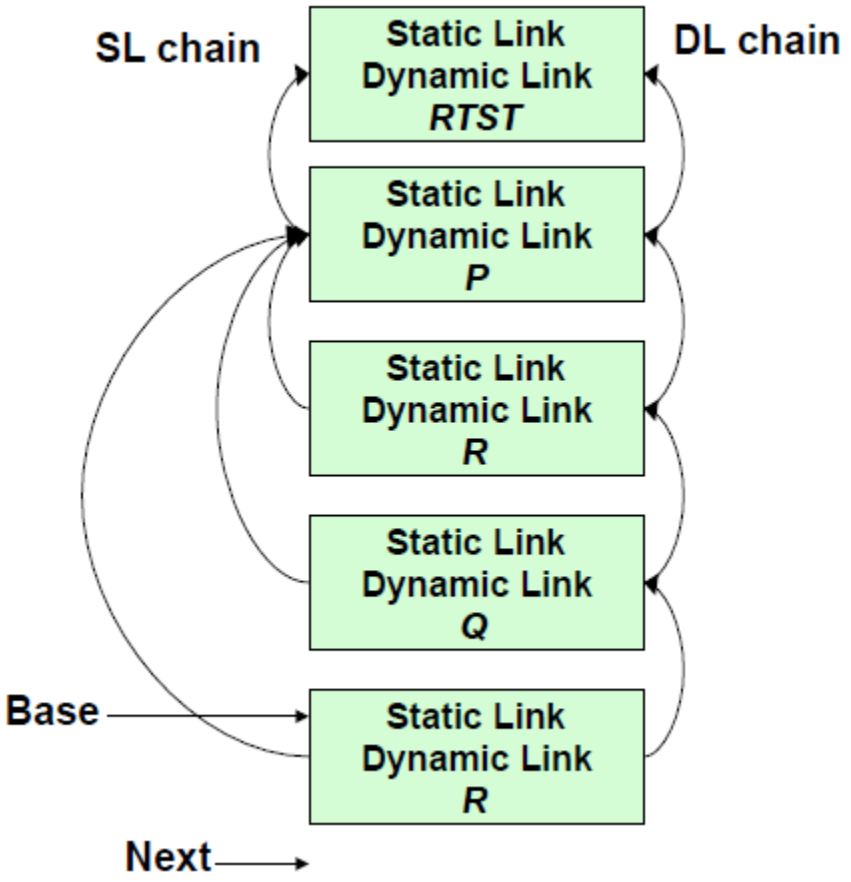
But SL from Q→P, as Q can access variables of P and RTST

# Allocation of nested procedure

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

- **Call sequence**

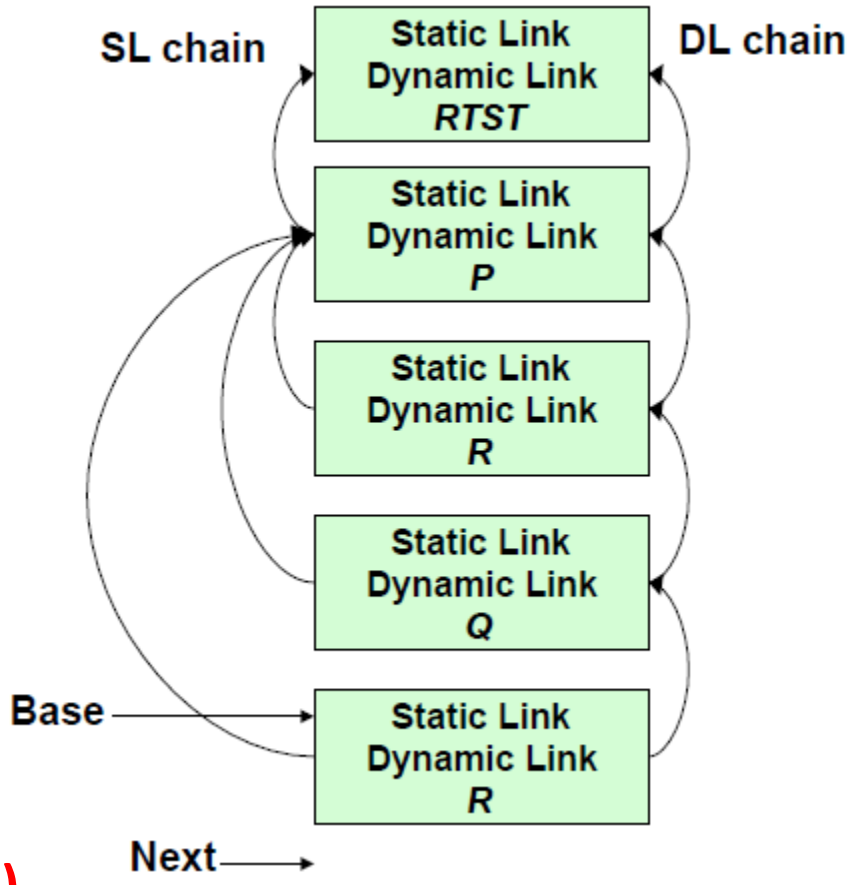
  **RTST -> P -> R -> Q -> R**




No SL: R→Q and R→R

# Allocation of nested procedure

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




- **Call sequence**

**RTST(1) -> P(2) -> R(3) -> Q(3) -> R(3)**

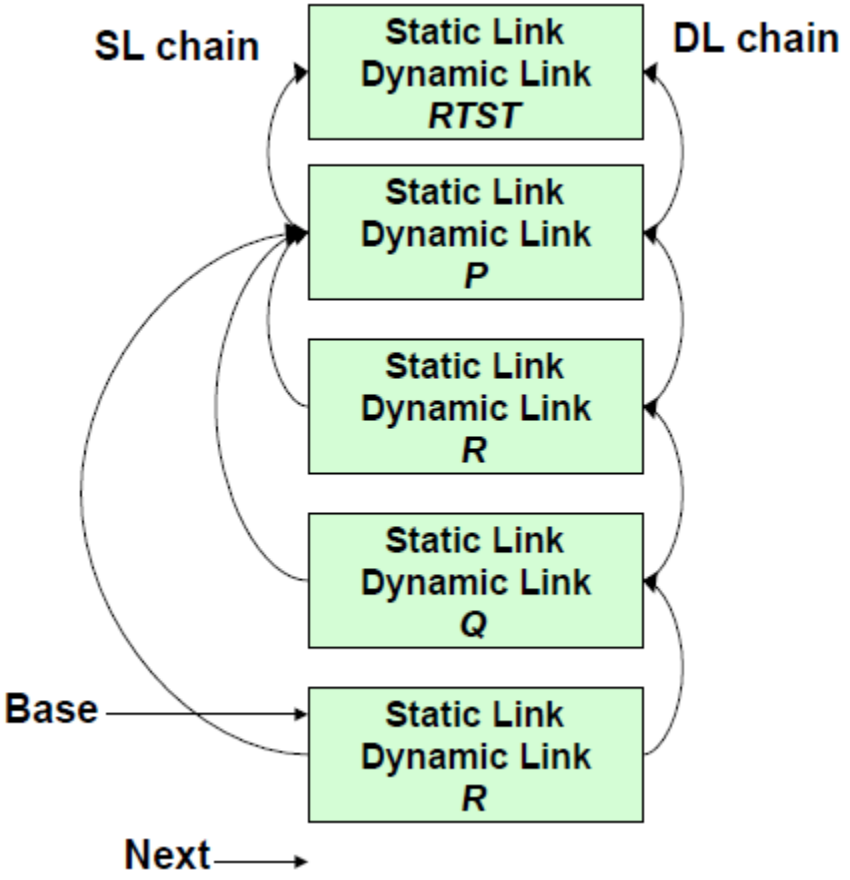
# Allocation of nested procedure

How SL is determined?

- Skip **L1 - L2 + 1** records starting from the caller's AR and establish the static link to the AR reached
- L1=caller and L2=Callee

**RTST(1) → P(2) → R(3) → Q(3) → R(3)**

- for **P(2)→R(3),** 2 – 3 + 1 = 0;
  hence the SL of R points to P
- for **R(3)→Q(3),** 3 – 3 + 1 = 1;
  skipping 1 link starting from R,
  we get P;
  SL of Q points to P

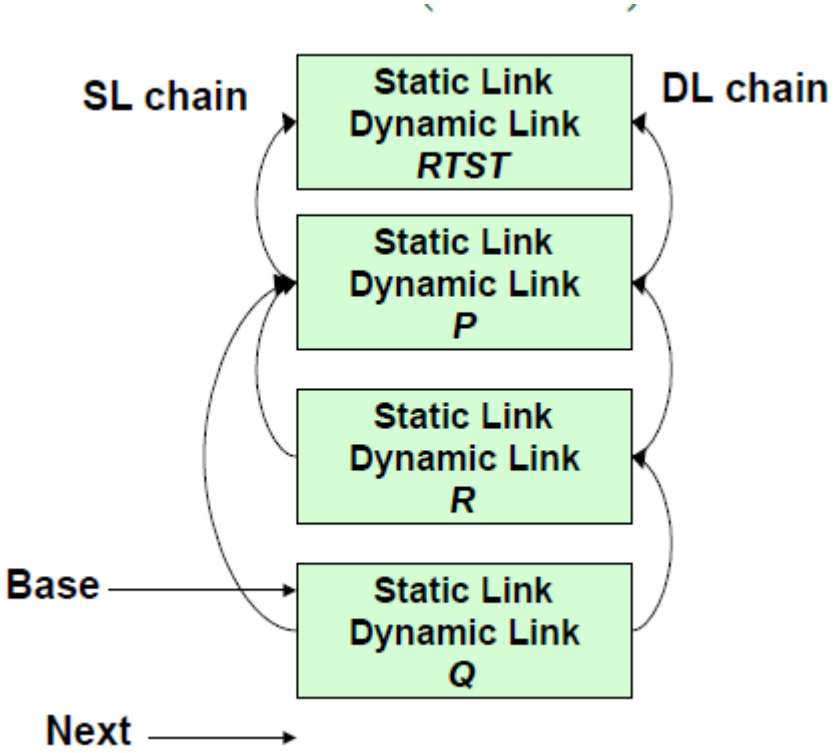


# Creation of activation record happens in callee code

- The creation of activation record takes place after the callee assumes control, because the exact size of the activation record will be known to the callee function.
- It will not be known to the caller.
- Callee functions can possibly be compiled separately.
- So, the total area for the variables of the function, its temporaries will not be known to the caller.
- Callers will only the size of the parameter list.
- So, the complete creation of the activation record really happens in the callee code.

# Allocation of nested procedure

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

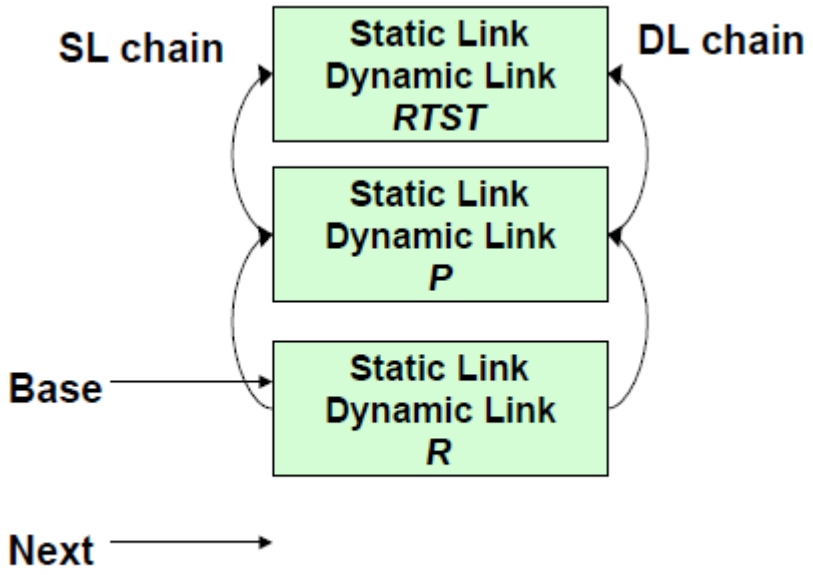



- **Call sequence**

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

# Allocation of nested procedure

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

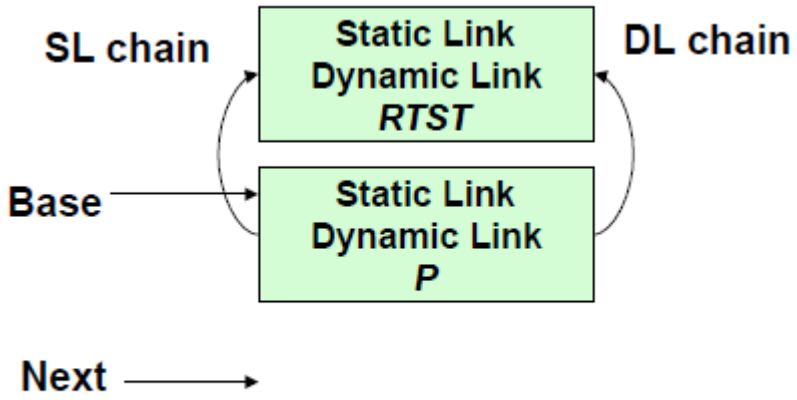



- **Call sequence**

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

# Allocation of nested procedure

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

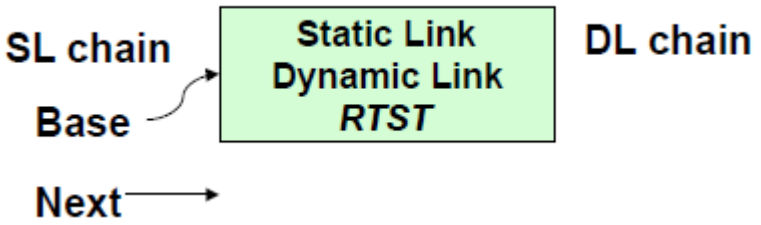



- **Call sequence**

  **RTST -> P ← R** **Return from R**

# Allocation of nested procedure

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




- **Call sequence**

  **RTST ← P** **Return from P**

# Static vs. Dynamic link

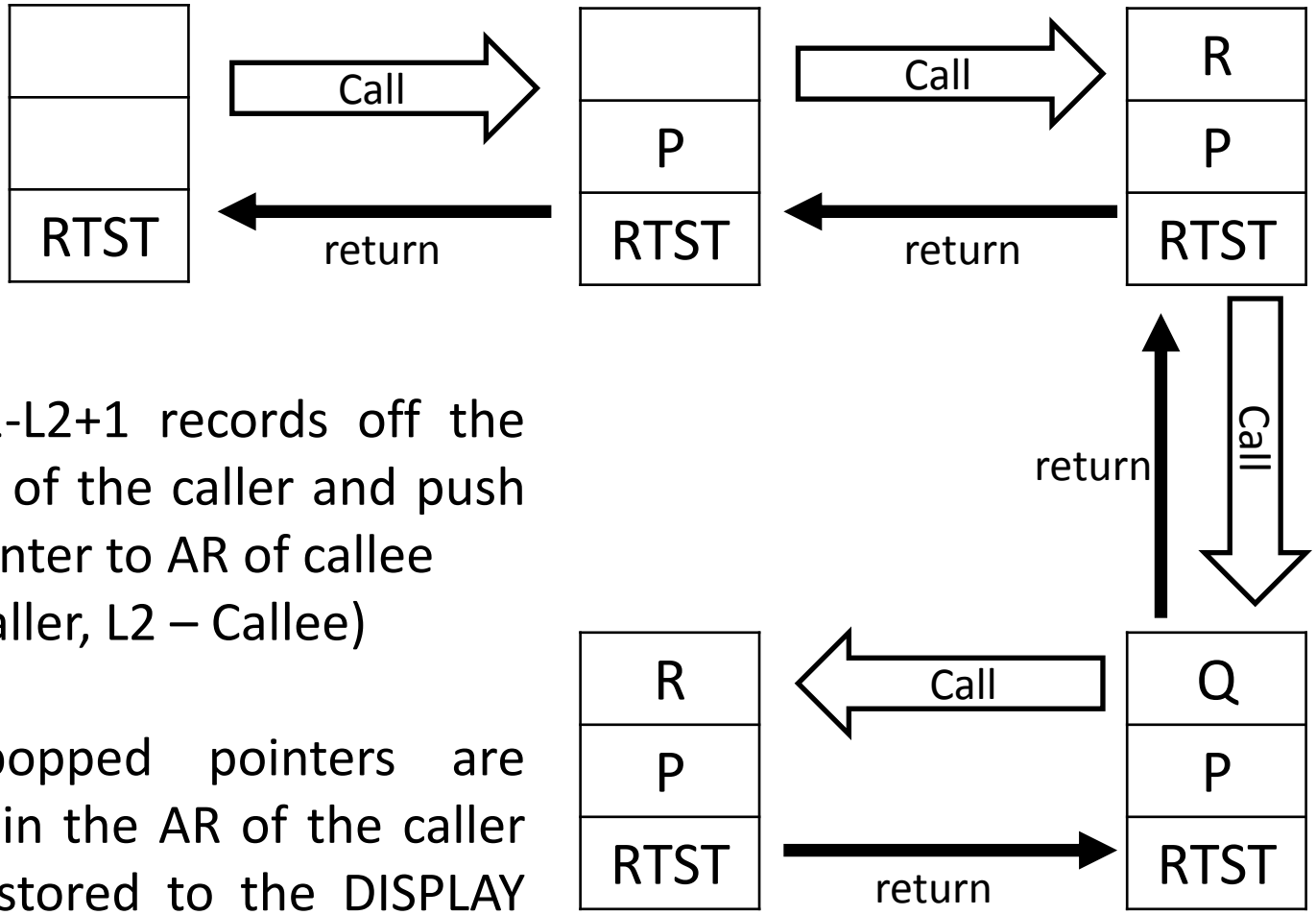
- **Static link:**
  - to access the global variables in the various activation records.

- **Dynamic link:**
  - to maintain the stack of activation records.

# Display Stack

- It is **data structure** to be used instead of **static link**.
- A stack of pointers which point to the activation records of procedures which are right now executing is maintained instead of static link.
- The most recent procedure which is activated its activation record pointer is on the top of the stack.
- The display stack structure must reflect the scope of the various functions and procedures appropriately.

# Display Stack of Activation Records (without SL)




Pop L1-L2+1 records off the display of the caller and push the pointer to AR of callee (L1 – caller, L2 – Callee)

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

# What about languages that don't support nested procedures?

- Example:- C language

- No requirement for static link

- Two links are needed
  1. To the beginning of the activation record
  2. To the static area containing global variables

- Dynamic link structure will handle the allocation and deallocation of the stack.

| Static (lexical) scope | Dynamic scope |
| --- | --- |
| C, C++, Java, Pascal, Python | Lisp, Perl, Logo, LaTeX |
| The name resolution depends on the location in the source code and the lexical context, which is defined by where the named variable or function is defined. | The name resolution depends upon the program state when the name is encountered which is determined by the execution context or calling context. |
| A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text. | A global identifier refers to the identifier associated with the most recent activation record. |
| Uses the static (unchanging) relationship between blocks in the program text. | Uses the actual sequence of calls that are executed in the dynamic (changing) execution of the program. |

# Example 1 (C-like structure is used)

```
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**

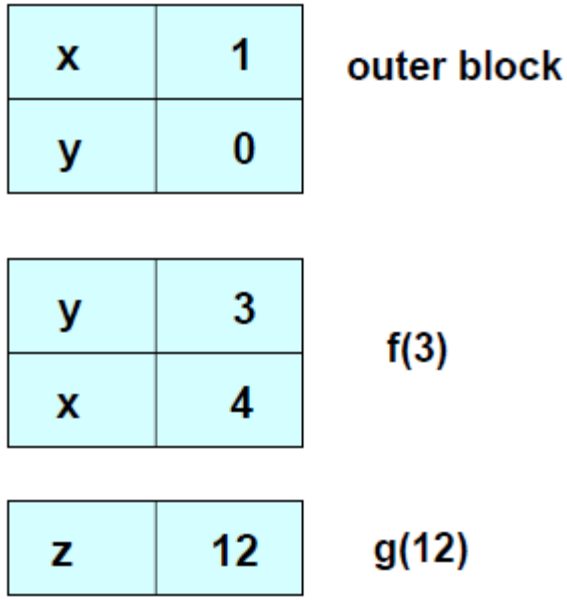
x = 1

So, y = 1+ 12 = **13**

**Dynamic scope**

x = 4

So, y = 4 + 12 = **16**




Stack of activation records after the call to *g*

# Example 2 (C-like structure is used)

```
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");
}
```

| Output Static Scope | Output Dynamic Scope |
|---|---|
| ? | ? |

# Example 2 (C-like structure is used)

```
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");
}
```

| Output Static Scope | Output Dynamic Scope |
|---|---|
| 0.25 0.25<br>0.25 0.25 | 0.25 0.125<br>0.25 0.125 |

# Implementing Dynamic scope

1. Deep access method

2. Shallow access method

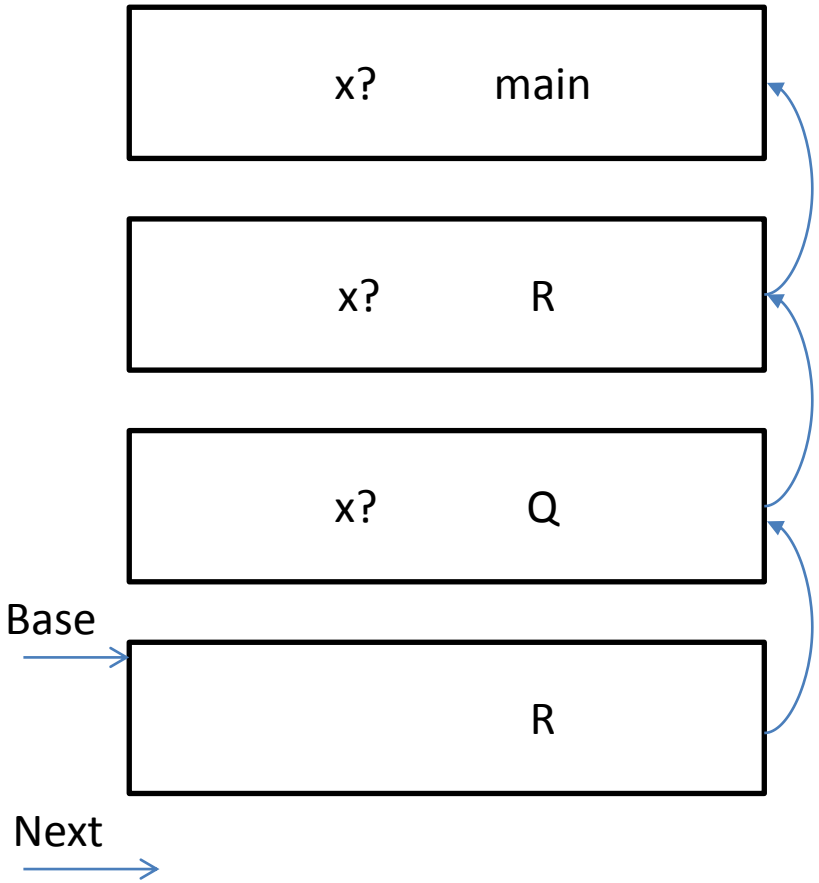
## Deep Access Method

- The idea is to keep a stack of active variables.
- Use control links instead of access links and to find a variable, search the stack from top to bottom looking for most recent activation record that contains the space for desired variables.
- Since search for nonlocal variables is made “deep” in the stack, the method is called deep access.
- Here, a symbol table should be used at runtime.

## Shallow Access Method

- The idea is to keep a central storage and allot one slot for every variable name.
- If the names are not created at runtime then the storage layout can be fixed at compile time.
- Else, when new activation procedure occurs, then that procedure changes the storage entries for its local at entry and exit.
- Shallow access allows fast access but has a overhead of handling procedure entry and exit.

# Deep Access Example




Calling Sequence
Main → R → Q → R

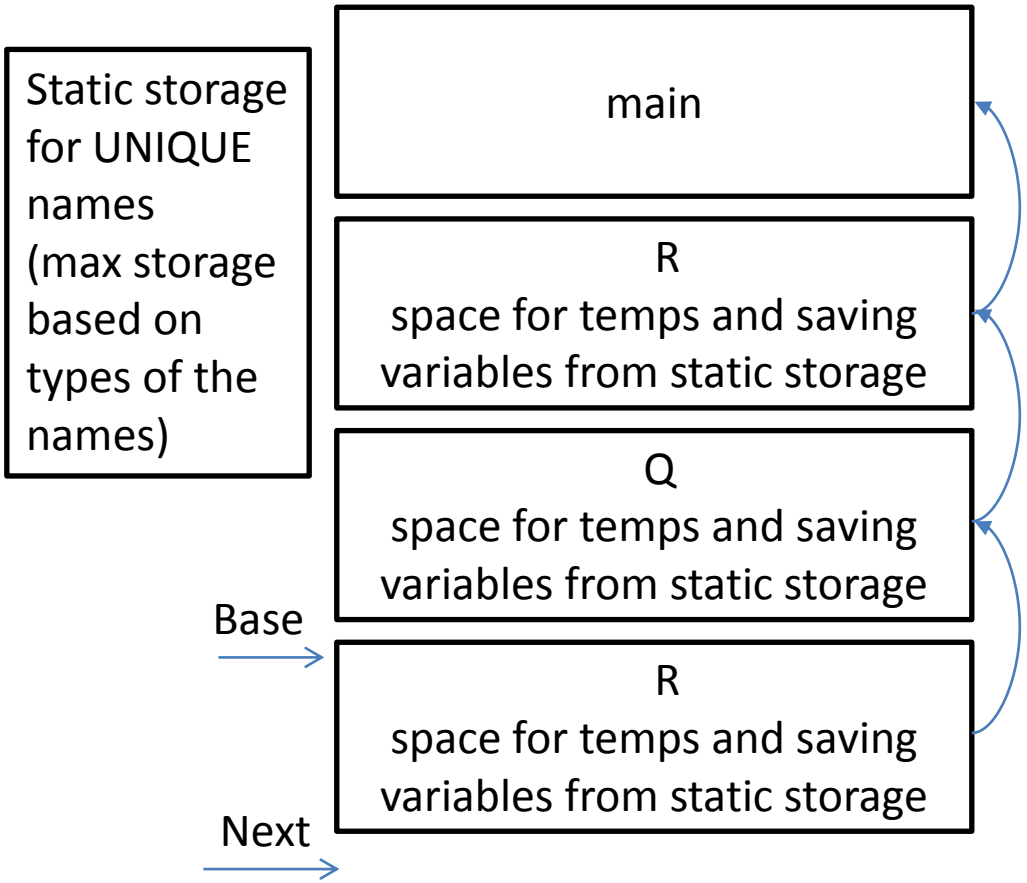
Currently, R is being accessed. (Base)

In R, if we don't find x, we search in Q then again R and then in main.

# Deep Access Method

- Dynamic link is used as static link.
- Activation records are searched on the stack to find the first activation record containing the non-local name to be found.
- The time required to access global variables is much more than the time required to access local variables.
- The time to access a global variable will depend on the sequence of calls that are made (hence can’t be determined at compile-time).
- Needs some information on the identifiers to be maintained at runtime within the ARs.

# Shallow Access Example




Calling Sequence
Main → R → Q → R

Direct and quick access to global variables, but some overhead is incurred when activations begin and end.

# Shallow Access Method

- Variables declared in the procedures are stored in a unique static storage area.
- There is exactly one fixed amount of storage for each unique name.
- If same name is declared in various procedures with different type then maximum storage required of the given types is allocated in the static storage area.
- The advantage is every name has a unique address and it is static so there is no need to use a stack pointer to access.
- But, there is an overhead of storing and restoring at begin and end of the activation record.

# Runtime Environment

- Parameter passing methods
- Static storage allocation
- Dynamic stack storage allocation
  - Activation record structure
  - Offset calculation for overlapped storage
  - **Allocation of nested procedure**
  - **Display stack structure (without static link)**
  - **Static and Dynamic scope**
    - **Deep Access Method for dynamic scope**
    - **Shallow Access Method for dynamic scope**