CC Lecture 11

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

Comparison

```
1. void swap (int x, int y)
2. { int temp;
3. temp = x;
4. x = y;
5. y = temp;
6. } /*swap*/
7. ...
8. { i = 1;
9. a[i] =10; /* int a[5]; */
10. print(i, a[i]);
11. swap(i, a[i]);
12. print(i, a[1]); }
```

call-by-value		**call-by-reference**		**call-by-value-result**		**call-by-name**	
1	10	1	10	1	10	1	10
1	10	10	1	10	1	Error!	

Reason for the error in the Call-by-name Example

temp = i; /* => temp = 1 */
i = a[i]; /* => i =10 since a[i] ==10 */
a[i] = temp; /* => a[10] = 1 => index out of bounds */

Comparison

```
void swap (int x, int y)
{ int temp;
temp = x;
x = y;
y = temp;
} /*swap*/
...
{ i = 1;
a[i] =10; /* int a[11]; */
print(i, a[i]);
swap(i, a[i]);
print(i, a[1]); }
```

call-by-value		**call-by-reference**		**call-by-value-result**		**call-by-name**	
1	10	1	10	1	10	1	10
1	10	10	1	10	1	10	10

Call-by-name

```
temp = i; /* => temp = 1 */
i = a[i]; /* => i =10 since a[i] ==10 */
a[i] = temp; /* => a[10] = 1 */

print(i, a[1]);  /* 10 10  =>  a[1] is unchanged*/
```

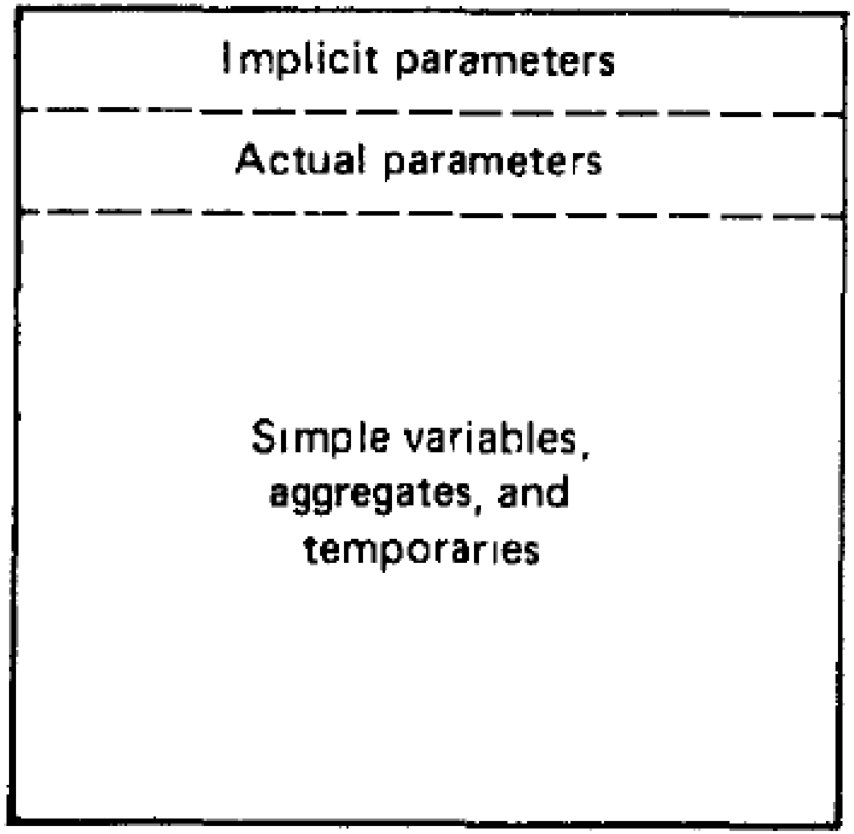
Memory map of data area in main memory

Main Program Variables
Procedure 1 variables
Procedure 2 Variables
Procedure 3 Variables
…

- These addresses are fixed, they are not going to change at any time.
- The compiler allocates the space for all the variables both local and global of all the procedures at compiled time itself.

- Suppose, procedure P1 calls itself
- The data area of P1 is fixed.
- The same area will be used by the second instance of P1 which being recursively is called the original instance of P1.
- And the second instances are both alive at the same time.
- But the data area being simple single.
- The second instance of procedure P1 will over write all the data created by the first instance.

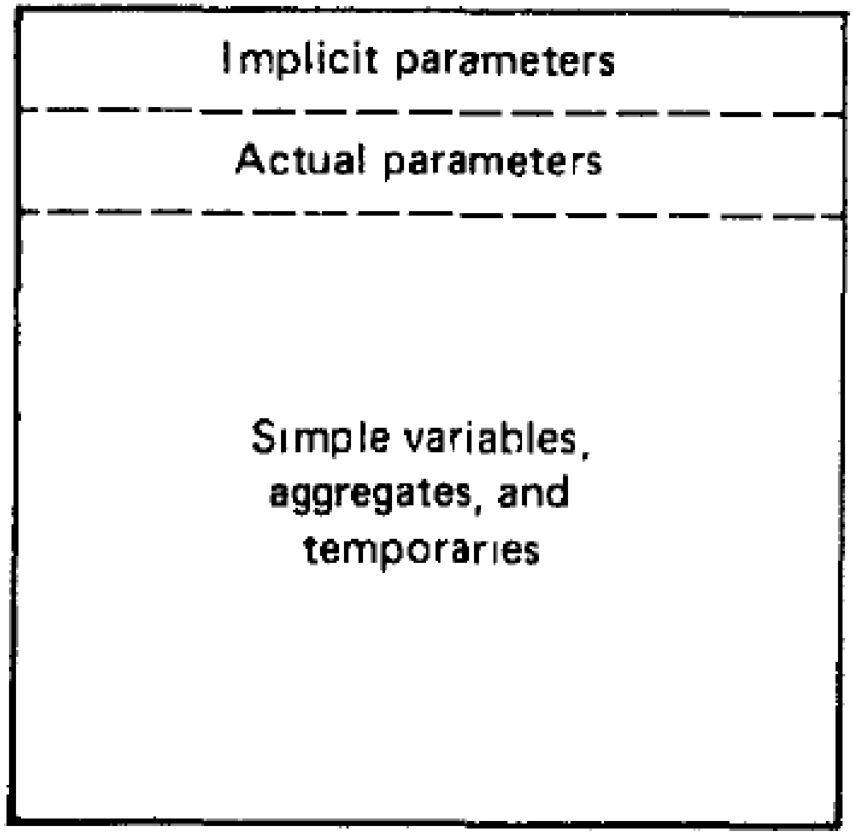
Typical data-area for static storage-allocation strategy



- An **implicit parameter** is primarily used for communication with the calling module.
- Typically such a parameter is the return address to the calling procedure, or the return value of a functional procedure, when it is not convenient to return this value in a register.

- An **actual parameter** contains the value or address of the value of an argument that is designated in a call to the module.

Typical data-area for static storage-allocation strategy



- The program variables' section contains the storage space for the simple variables, aggregates (i.e., arrays and records), compiler-generated temporary variables, etc.

A call to a procedure consists of the following steps:

1. Bring forward the values or evaluate the addresses of the actual parameters (i.e., arguments from the calling procedure) and store them in a list in the calling procedure's data area.
2. Place the address of the parameter list in a register.
3. Branch to the procedure.

- Prior to the execution of the procedure both the implicit and explicit parameters must be moved into the special locations that have been previously reserved in the data area.
- When returning to the calling procedure, the implicit parameters are loaded into registers and a jump back to the calling procedure occurs as dictated by the return address.

Advantages

- The **memory size** allocated to “data” is **static**.
  - But it is possible to **change content** of a static structure without increasing the memory space allocated to it.
- **Global variables** are declared “ahead of time,” such as fixed array.
- **Lifetime** of static allocation is the **entire runtime** of program.
- It has **efficient execution**.

Disadvantages

- In case more static data space is declared than needed,
  - there is waste of space.

- In case less static space is declared than needed
  - then it becomes impossible to expand this fixed size during run time.

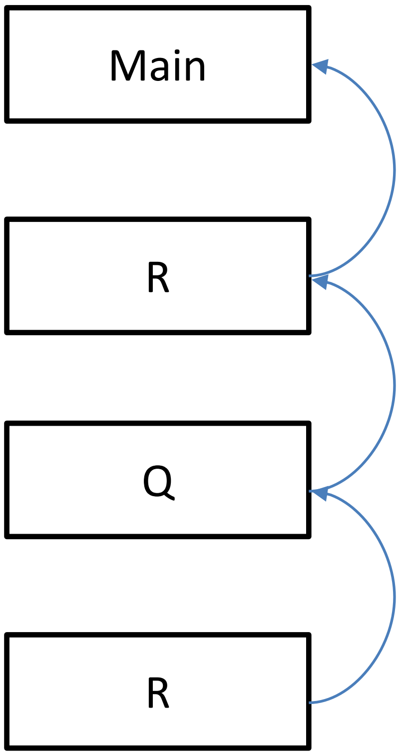
In a nutshell

- Compiler allocates space for all variables (local and global) of all procedures at compile time
- No stack/heap allocation; no overheads; no recursion
- Variable access is fast since addresses are known at compile time
- Examples:-
  - code in languages without dynamic compilation
  - all variables in FORTRAN IV
  - global variables in C, Ada, Algol
  - constants in C, Ada, Algol

Dynamic Data Storage Allocation

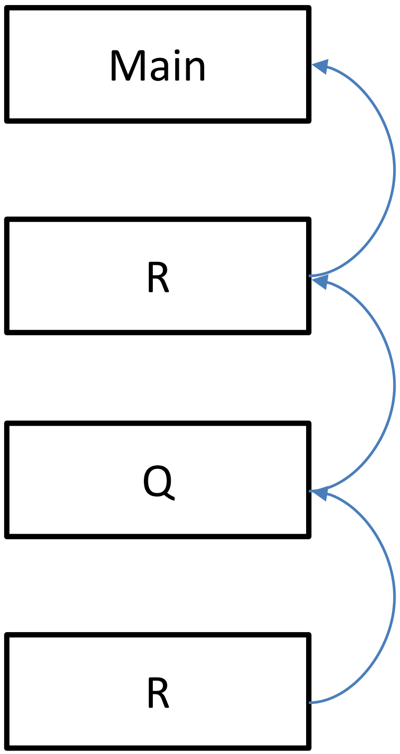
- Compiler allocates space only for global variables at compile time
- Space for variables of procedures will be allocated at run-time
  - Stack/heap allocation
  - Ex: C, C++, Java, Fortran 8/9
  - Variable access is slow (compared to static allocation)
    - addresses are accessed through the stack/heap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

Dynamic Stack Storage Allocation



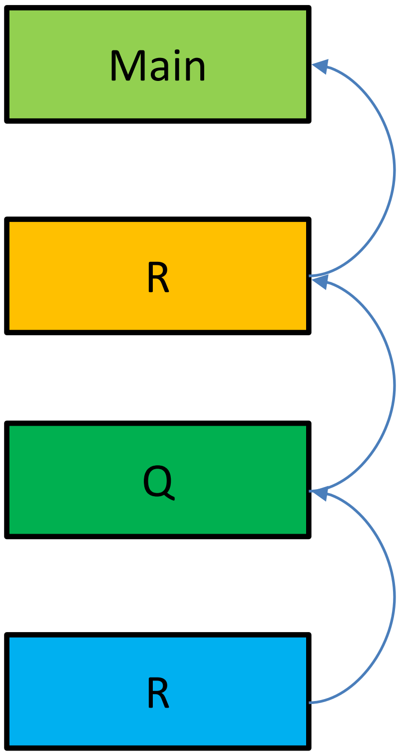
- Calling sequence
  - Main →R →Q →R
- Stack of activation records
  - data areas for the various activations of the functions or procedures
- the variable space for the second instance and the first instances are different, there by useful work can be done by both the instances.

Allocation is done when call to a procedure is made



- To begin with allocation only for the global variables, and Main.
- When Main calls R, the data space for R is created
- and then when it calls Q the space for Q gets created.
- And then when there is a recursive call to R another space for gets created.

Termination of procedure, releases space



- When R terminates the space for R will be released
- When Q returns the space used by Q will be returned.
- Then, the space of R of will be released.
- Finally, when the Main program terminates all the space will be released by the runtime system.

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

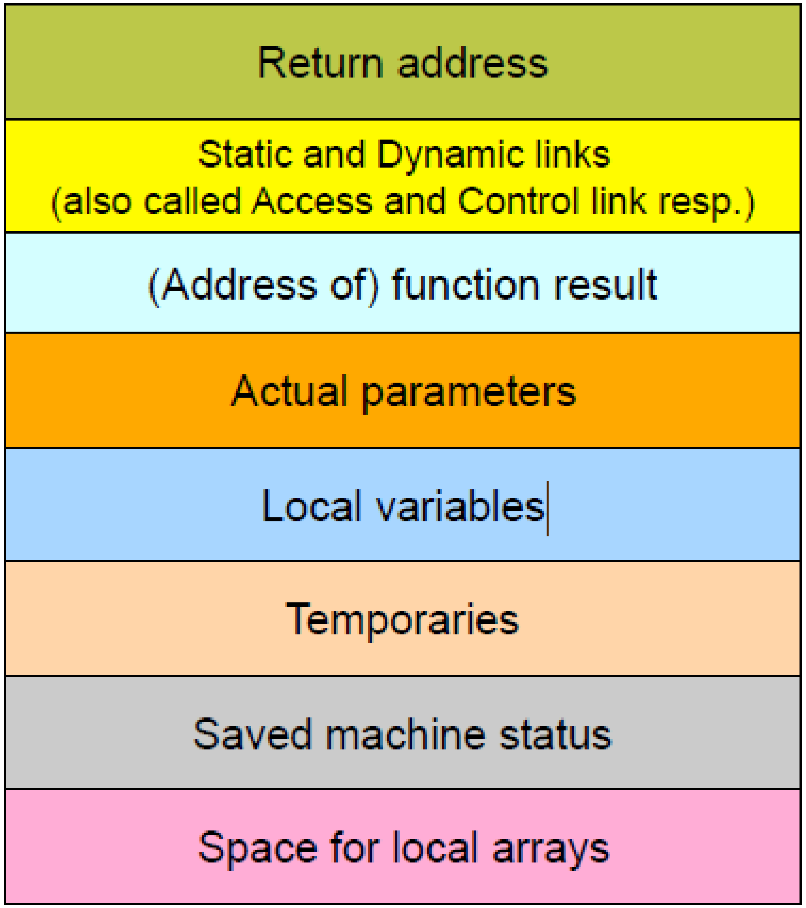
- The position of the fields of the activation record as shown are only notional.
- Implementations can choose different orders; e.g., function result could be after local variable.
- It is possible to change the location of these fields without affecting either the efficiency or speed of the program itself.

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

- Return address
  - required by the program to return to the caller
- static and dynamic link
  - used to access global variables from the current procedure
- address of the function result
  - the variable which contains the function result.
  - the address of that variable will be passed as an implicit parameter

Activation Record Structure



- the local variables or parameters which are non arrays
  - Will require known amounts of spaces
- Hence, local arrays are located in the end

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

```
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

```
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 */
        ... }
    }
}
```

What will be the storage required??

```
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 */
        ... }
    }
}
```

Storage required

=B1+max(B2,(B3+max(B4,B5)))

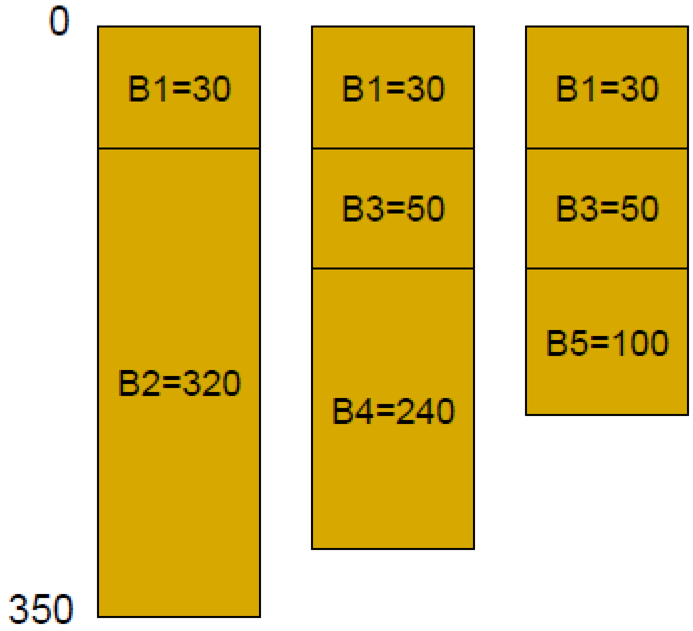
=30+max(320,(50+max(240,100)))

=30+max(320, (50+240))

=30+max(320,290)

= **350**

Overlapped Variable Storage for Blocks



Storage required

=B1+max(B2,(B3+max(B4,B5)))

=30+max(320,(50+max(240,100)))

=30+max(320, (50+240))

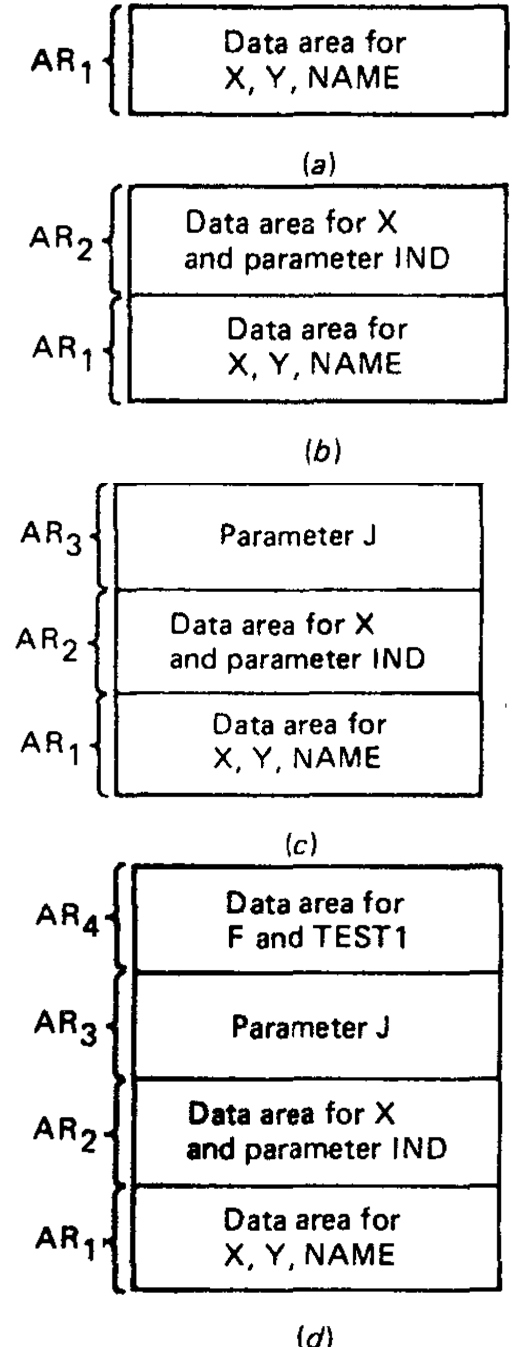
=30+max(320,290)

= **350**

```
1  BBLOCK;
        REAL X, Y, STRING NAME;
        •
        •
        •
   2  M1:   PBLOCK (INTEGER IND);
            INTEGER X;
            •
            •
            •
            CALL M2(IND + 1),
            •
            •
            •
      END M1;

   3  M2:   PBLOCK (INTEGER J);
            •
            •
            •
      4  BBLOCK;
              ARRAY INTEGER F(J); LOGICAL TEST1;
              •
              •
              •
         END;
      END M2;
      •
      •
      •
      CALL M1 (X / Y);
      •
      •
      •
   END;
```

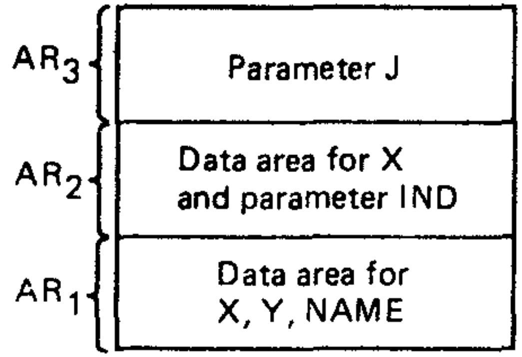
Trace of the run-time stack



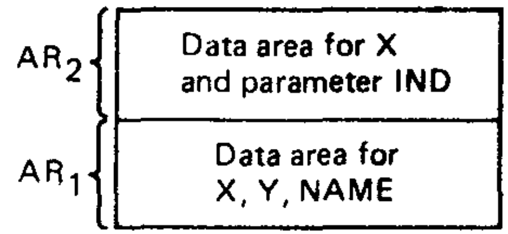
```
1  BBLOCK;
        REAL X, Y, STRING NAME;
        •
        •
        •
   2  M1:  PBLOCK (INTEGER IND);
           INTEGER X;
           •
           •
           •
           CALL M2(IND + 1),
           •
           •
           •
      END M1;

   3  M2:  PBLOCK (INTEGER J);
           •
           •
           •
      4  BBLOCK;
              ARRAY INTEGER F(J); LOGICAL TEST1;
           •
           •
           •
         END;
      END M2;
      •
      •
      •
      CALL M1 (X / Y);
      •
      •
      •
   END;
```

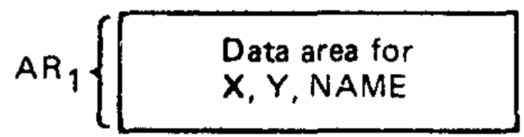
Trace of the run-time stack



(e)



(f)



(g)

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

- Here, Q and R are at same level.
- When procedure R is called it calls Q in turn and then when Q is called it calls R in turn, and this recursion will go on for a while.
- 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*

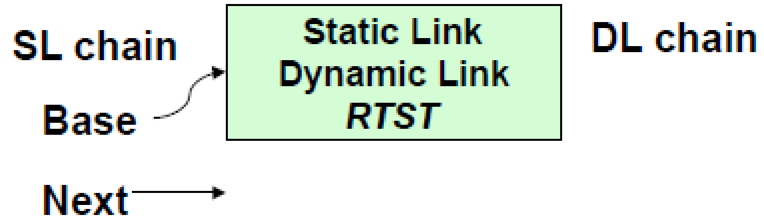
begin *P; end*

- **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.