CC Week 6-7-8

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

Ref. Book 1 The theory and practice of Compiler Writing by Jean-Paul Tremblay and Paul G. Sorenson

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

Compiler Phases

- **Lexical analysis**: processing characters
- **Parsing**: processing the tokens and producing the syntax tree
- **Semantic analysis**: which checks whether the semantics of the programming language are satisfied by the program.
- **Intermediate code generation**
- Before **machine code generation**, **what exactly** is required for a program to execute at the time when it is put into memory and the execution begins?

What is required??

- run time arrangement
  - code generator expects that these are available at run time
- run time support
  - various parameter passing methods
  - different types of storage allocation
  - the format of activation records
  - the difference between static scope and dynamic scope
  - how to pass functions as parameters
  - heap memory management
  - garbage collection.

Run Time Support

- Interfaces between the **program** and the computer system **resources** are needed
- There is a need to **manage memory** when a program is running:-
  - Memory management must connect to the data objects of programs
  - Programs request for memory blocks and release memory blocks
  - Passing parameters to functions
- Other resources such as printers, file systems, etc., also need to be accessed (done by operating system)

Parameter Passing Methods

1. Call by value
2. Call by reference
3. Call by value result
4. Call by name

Call-by-value

- At runtime, prior to the call, the parameter is evaluated, and its actual value is put in a location private to the called procedure
- There is no way to change the actual parameters.
- C has only call-by-value method available
  - Passing pointers does not constitute call-by-reference
  - Pointers are also copied to another location
  - Hence in C, there is no way to write a function to insert a node at the front of a linked list (just after the header) without using pointers to pointers
- Found in C and C++

Call-by-Reference

- At runtime, prior to the call, the parameter is evaluated and put in a temporary location, if it is not a variable.
- The **address of the variable** (or the temporary) is passed to the called procedure.
- Thus, the actual parameter may get changed due to changes to the parameter in the called procedure.
- Found in C++ and Java

Call-by-Value-Result

- Call-by-value-result is a hybrid of Call-by-value and Call-by-reference
- Actual parameter is calculated by the calling procedure and is copied to a local location of the called procedure
- Actual parameter’s value is not affected during execution of the called procedure
- At return, the value of the formal parameter is copied to the actual parameter, if the actual parameter is a variable

Call-by-Value-Result

- Becomes different from **call-by-reference** method
  - when global variables are passed as parameters to the called procedure and
  - the same global variables are also updated in another procedure invoked by the called procedure
- Found in Ada

Example: Call-by-value vs. Call-by-reference vs. Call-by-Value-Result

```
int a;
void Q() {
    a = a+1;
}
void R(int x){
    x = x+10;
    Q();
}
main(){
    a = 1;
    R(a);
    print(a);
}
```

Call by value	Call by reference	Call be value result
2	12	11

Note:
In Call-by-V-R, value of x is copied into a, when proc R returns.

Hence, a=11.

Call-by-Name

- Use of a call-by-name parameter implies a textual substitution of the formal parameter name by the actual parameter.
- Hence, we cannot evaluate the address of the actual parameter just once and use it.
- It must be recomputed every time, we reference the formal parameter within the procedure.
- A separate routine (called thunk) is used to evaluate the parameters whenever they are used.
- Found in ALGOL and functional languages

Call-by-Name

- For example, if the procedure

  void R (int X, int I);

  { I = 2; X = 5; I = 3; X = 1; }
- is called by **R(B[J*2], J)**
- this would result in (effectively) changing the body to

  { J = 2; B[J*2] = 5; J = 3; B[J*2] = 1; }
- just before executing it

- the actual parameter corresponding to *X changes whenever J changes*

Comparison

```
void swap (int x, int y)
{ int temp;
temp = x;
x = y;
y = temp;
} /*swap*/
...
{ i = 1;
a[i] =10; /* int a[5]; */
print(i, a[i]);
swap(i, a[i]);
print(i, a[1]); }
```

<table>
<tr><th colspan="2">call-by-value</th><th colspan="2">call-by-reference</th><th colspan="2">call-by-value-result</th><th colspan="2">call-by-name</th></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

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

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[11]; */
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	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*/

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 (e.g. dynamic loading of classes)
- Memory area can be allocated to code statically
- Data area of a program may grow or shrink in size during execution

Static vs. Dynamic 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 Storage Allocation

- In a static storage-allocation strategy, it is necessary to be able to decide at compile time exactly where each data object will reside at run time.

- In order to make such a decision, at least two criteria must be met:
  1. The size of each object must be known at compile time.
  2. Only one occurrence of each object is allowable at a given moment during program execution.

Due to these criteria, the following are **not allowed** for static allocation strategy:

- **Criterion One:**
  - Variable length strings (length cannot be determined at compile time)
  - Dynamic arrays (bounds and hence the size of data object unknown at compile time)

- **Criterion Two**:
  - Nested procedures
  - Recursive procedures

  (as which and how many times the procedure will be called is unknown at compile time)

FORTRAN

- FORTRAN typifies those languages in which a static storage-allocation policy is sufficient to handle the storage requirements of the data objects in a program.

- Because FORTRAN does not provide
  - variable-length strings
  - dynamic arrays
  - nested procedures

- Ex: FORTRAN IV and FORTRAN 77

Static storage-allocation strategy

- Very simple to implement
- During an initial pass of the source text, a symbol-table entry is created for each variable and the set of attributes
- Because the precise amount of space required by each variable is known at compile time, the object address for a variable can be assigned according to the following simple scheme.
  - The first variable is assigned some address A near the beginning of an allocated data area,
  - The second variable is assigned address A + n1 assuming the first variable requires n1 storage units (e.g., bytes),
  - The third variable is assigned address A + n1 + n2 assuming the second variable requires n2 storage units, and so on.

Part of a symbol table that would be created for the given FORTRAN program segment assuming integer values require four storage units and real values require eight.

```
REAL MAXPRN, RATE
INTEGER IND1, IND2
REAL PRIN (100), YRINT (5,100), TOTINT
        •
        •
        •
```

(a)

Name	Type	Dimension		Address
MAXPRN	R	0		264
RATE	R	0		272
IND1	I	0		280
IND2	I	0		284
PRIN	R	1		288
YRINT	R	2		1088
TOTINT	R	0		5088

Object Address

- **Absolute address**
  - If the compiler is written for a single-job-at-a-time environment
  - The initial address A is set such that the program and data area reside in a section of memory separate from the resident parts of the operating system.
- **Relative address**
  - If the compiler resides in a multiprogramming environment
  - a program and its data area may reside at a different set of memory locations each time the program is executed.
  - The loader reserves a set of memory locations for the program and sets a base register to the address of the first location in the data area.

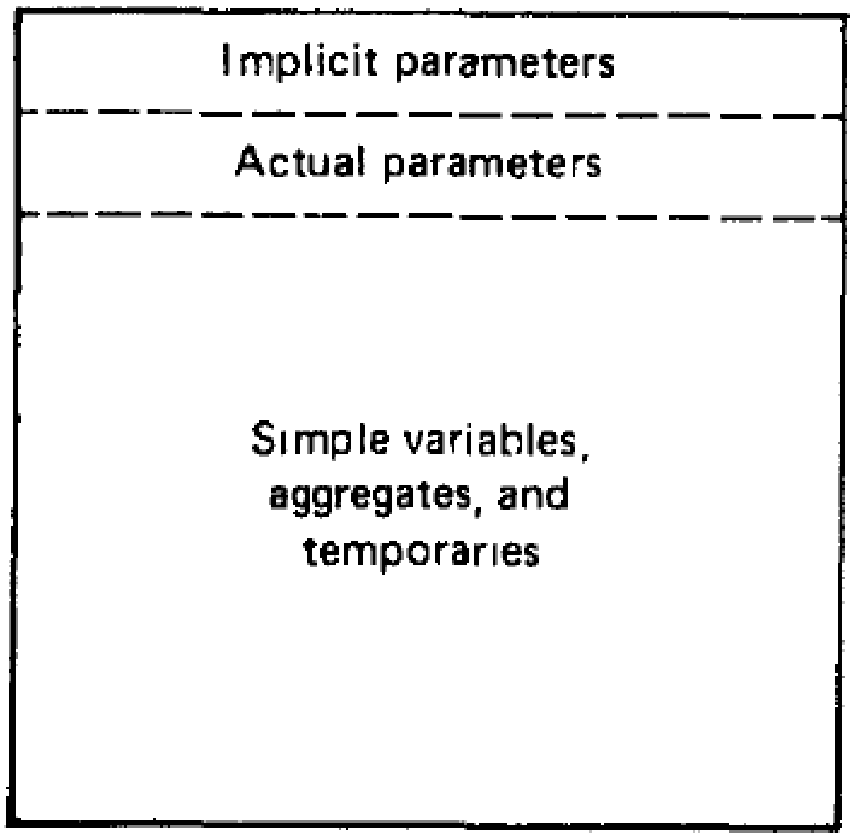
Memory map of data area in main memory

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.

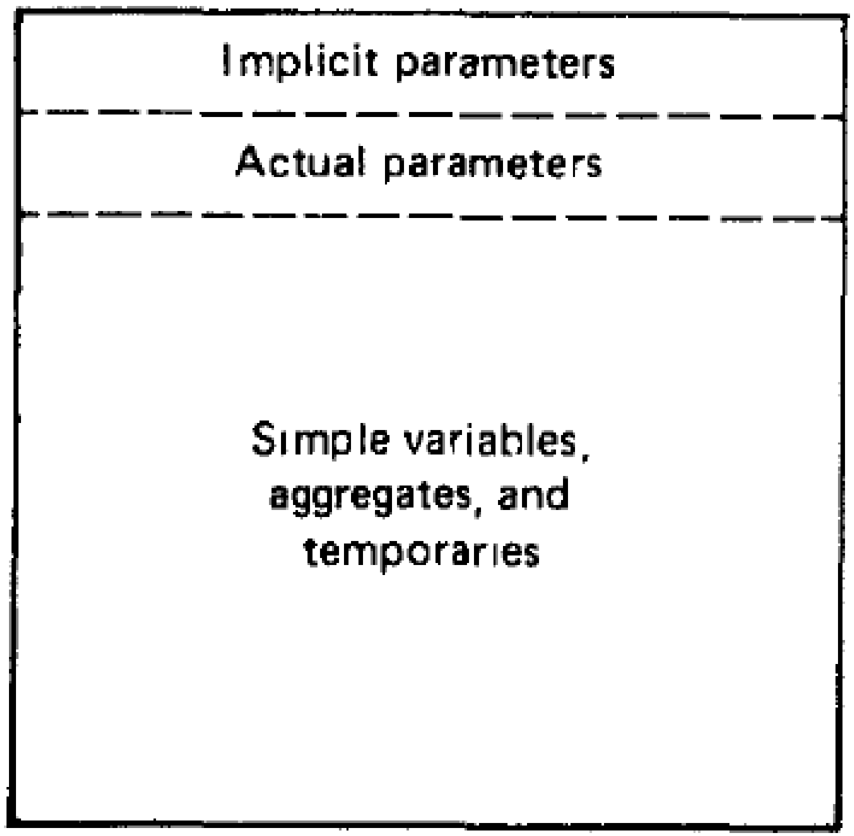
Typical data-area for static storage-allocation strategy



- 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.

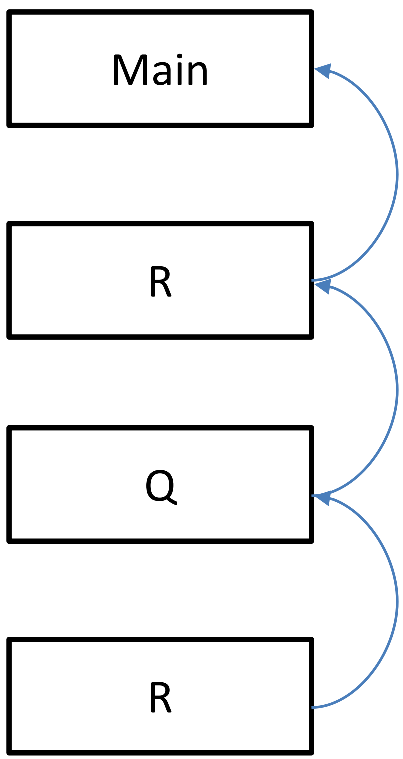
In a nutshell

- 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

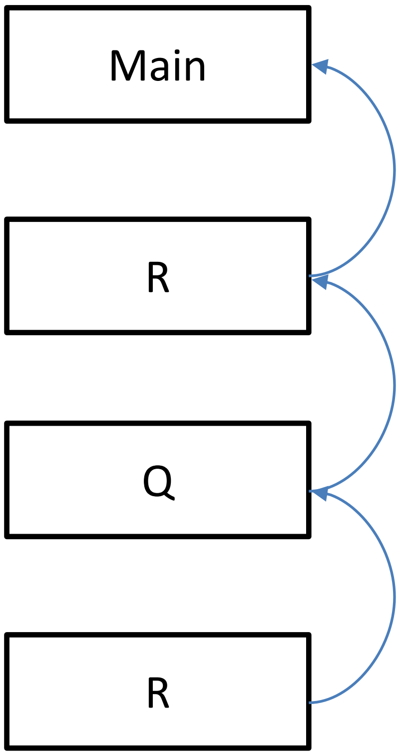
- Compiler allocates space only for global variables at compile time
- Space for variables of procedures will be allocated at run-time
  - Stack/heap allocation
  - Ex: C, C++, Java, Fortran 8/9
  - Variable access is slow (compared to static allocation)
    - addresses are accessed through the stack/heap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

Dynamic Stack Storage Allocation



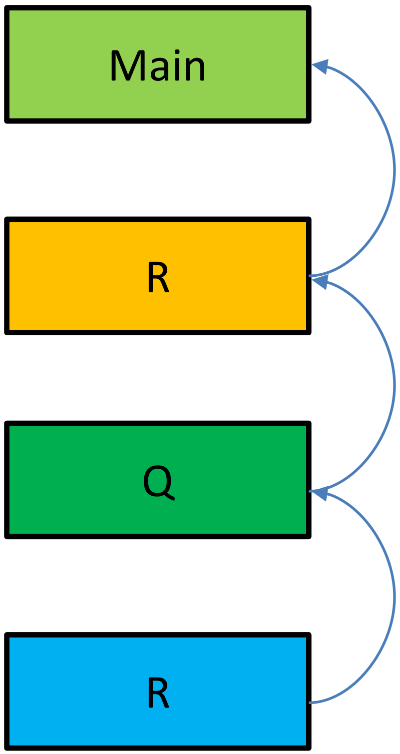
- Calling sequence
  - Main →R →Q →R
- Stack of activation records
  - data areas for the various activations of the functions or procedures
- the variable space for the second instance and the first instances are different, there by useful work can be done by both the instances.

Allocation is done when call to a procedure is made



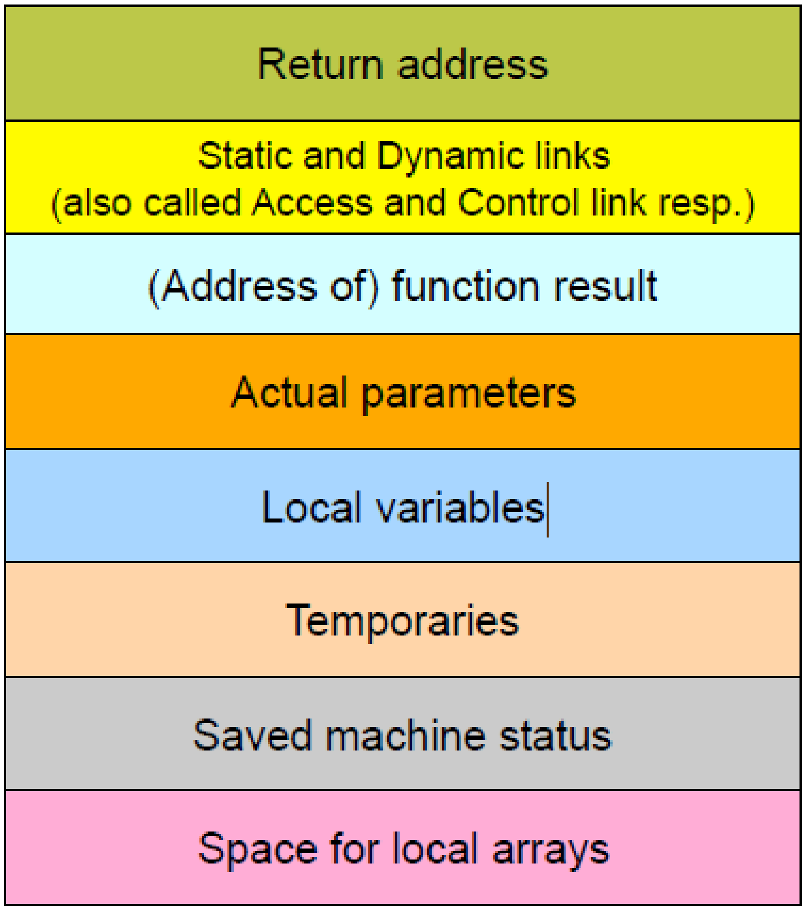
- To begin with allocation only for the global variables, and Main.
- When Main calls R, the data space for R is created
- and then when it calls Q the space for Q gets created.
- And then when there is a recursive call to R another space for gets created.

Termination of procedure, releases space



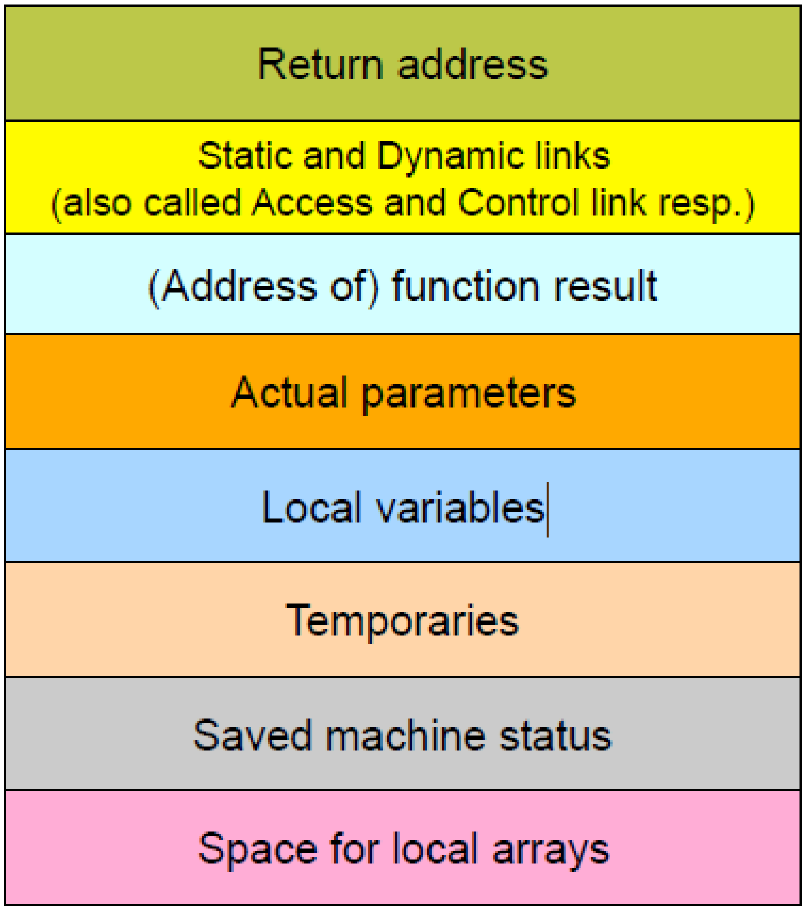
- When R terminates the space for R will be released
- When Q returns the space used by Q will be returned.
- Then, the space of R of will be released.
- Finally, when the Main program terminates all the space will be released by the runtime system.

Activation Record Structure



- 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
  - 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

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 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 */
        ... }
    }
}
```

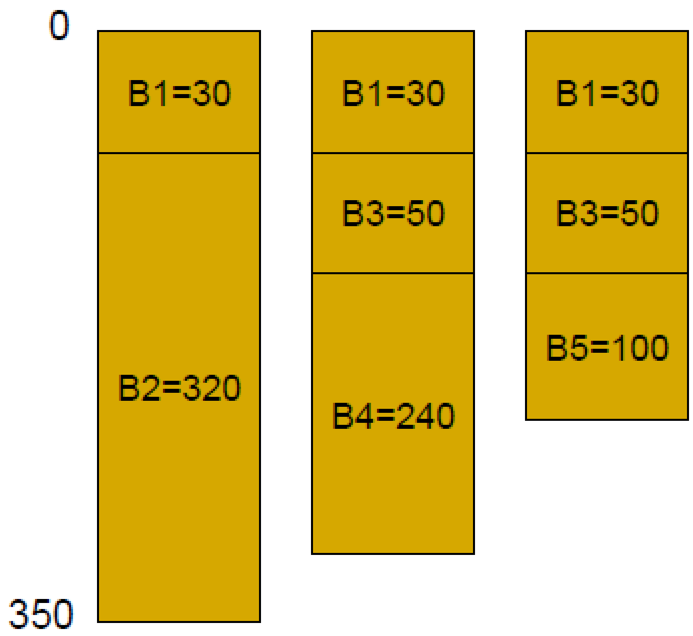
What will be the storage required??

```
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)))

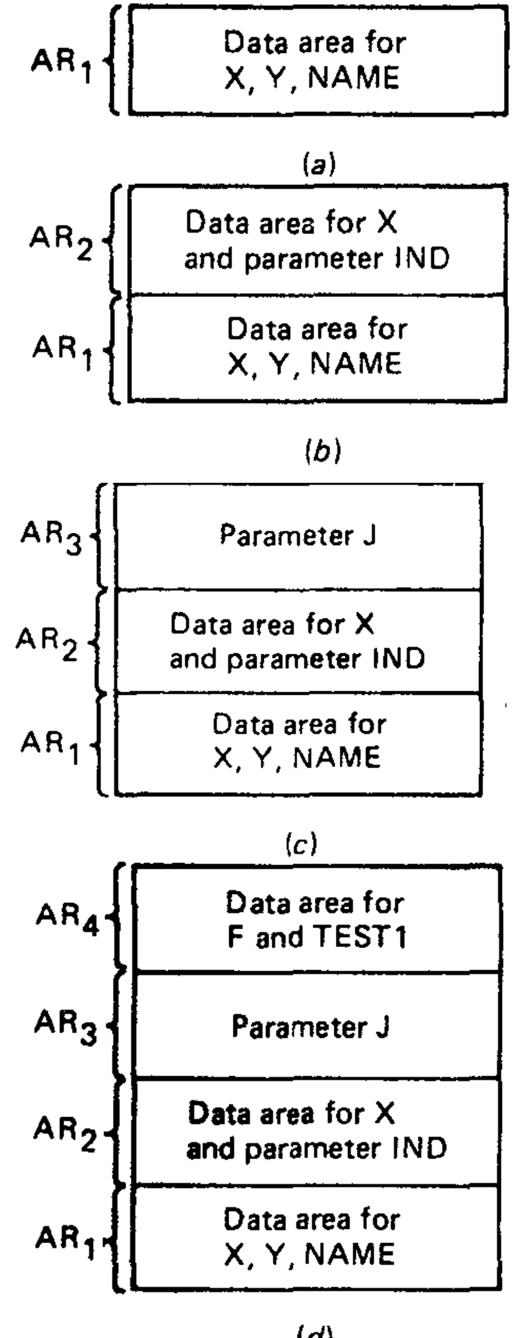
=30+max(320, (50+240))

=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

AR_3	Parameter J
AR_2	Data area for X and parameter IND
AR_1	Data area for X, Y, NAME

(*e*)

AR_2	Data area for X and parameter IND
AR_1	Data area for X, Y, NAME

(*f*)

AR_1	Data area for X, Y, NAME

(*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
```

- 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*

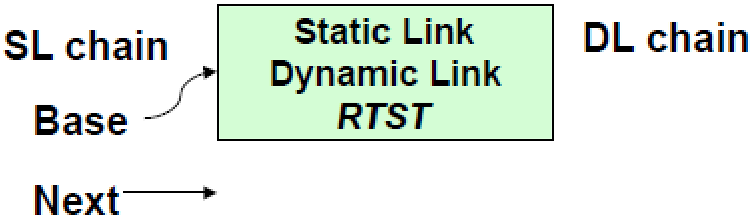
begin *P; end*

- **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;*

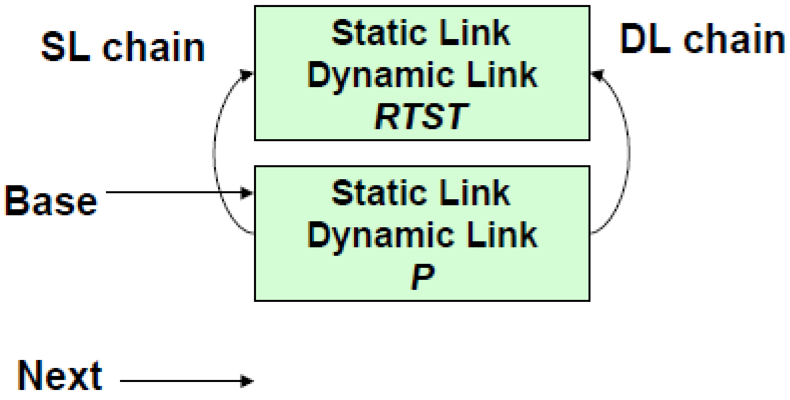
begin *R; end*

procedure *R;*

begin *Q; end*

begin *R; end*

begin *P; end*



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

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

- **Call sequence**

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

Allocation of nested procedure

program *RTST;*

    procedure *P;*

        procedure *Q;*

            begin *R; end*

        procedure *R;*

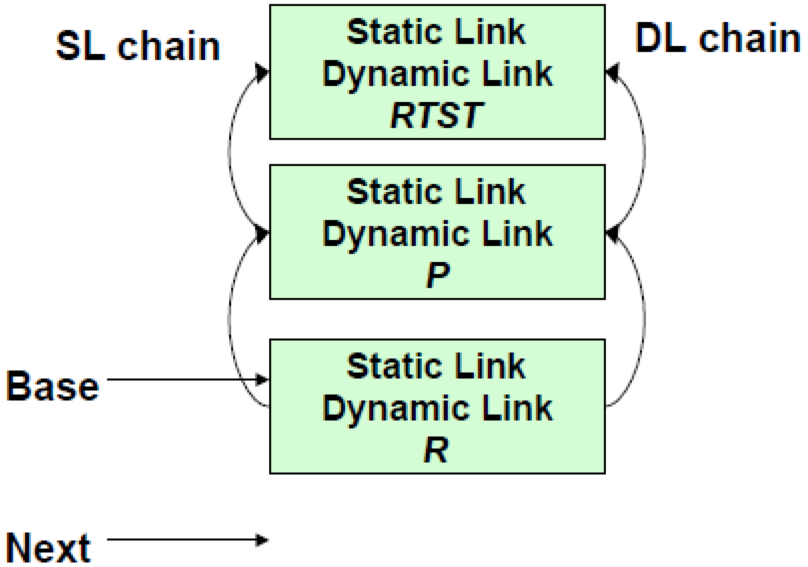
            begin *Q; end*

    begin *R; end*

begin *P; end*

- **Call sequence**

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

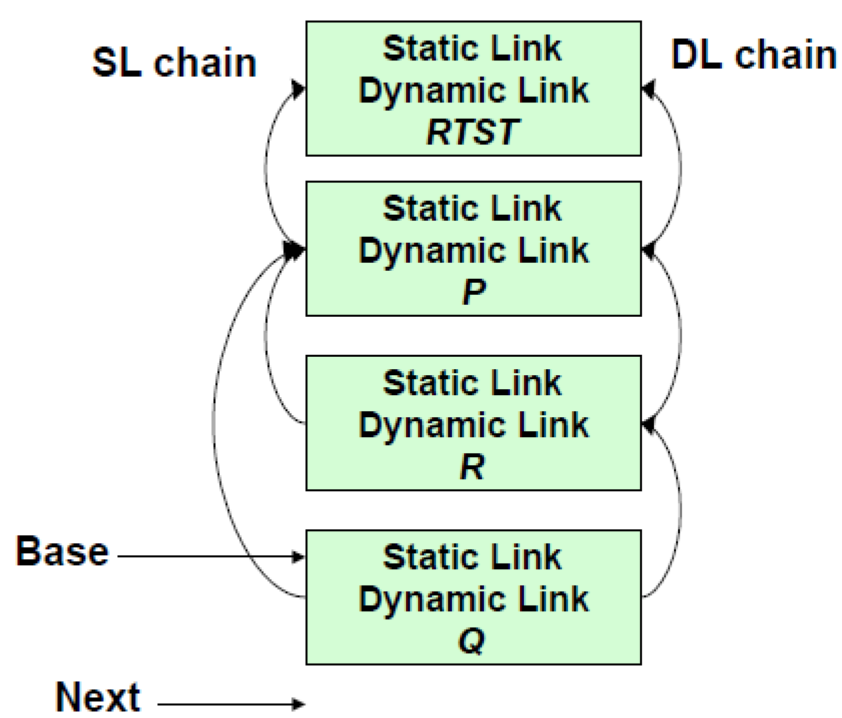
begin *R; end*

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;*

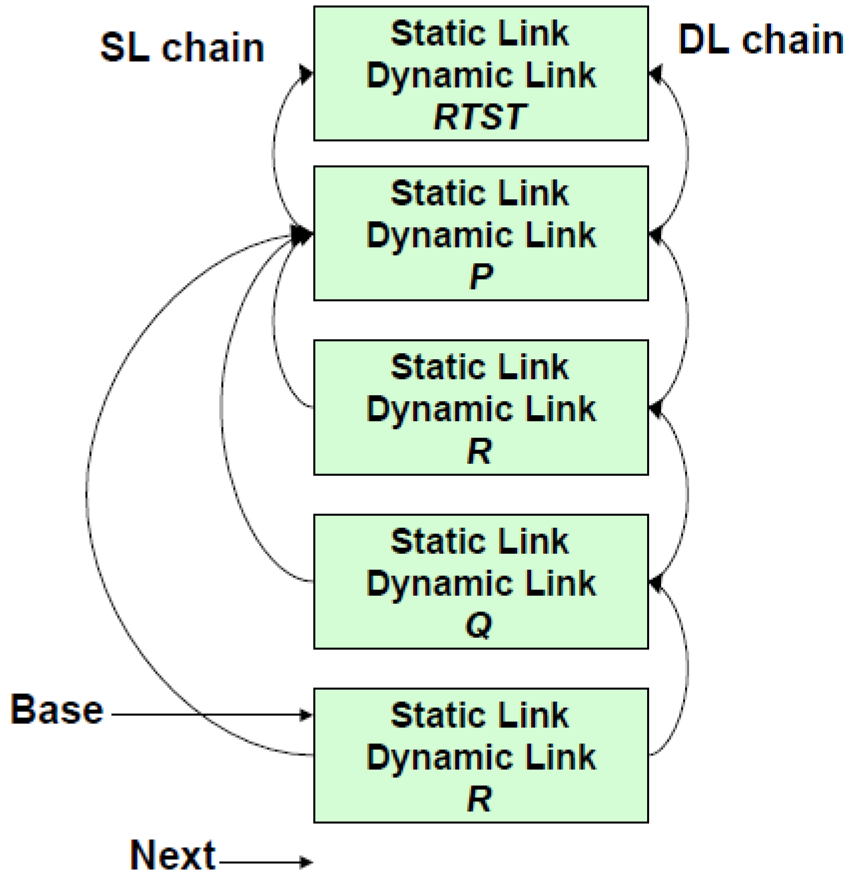
begin *Q; end*

begin *R; end*

begin *P; end*

- **Call sequence**

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



No SL: R→Q and R→R

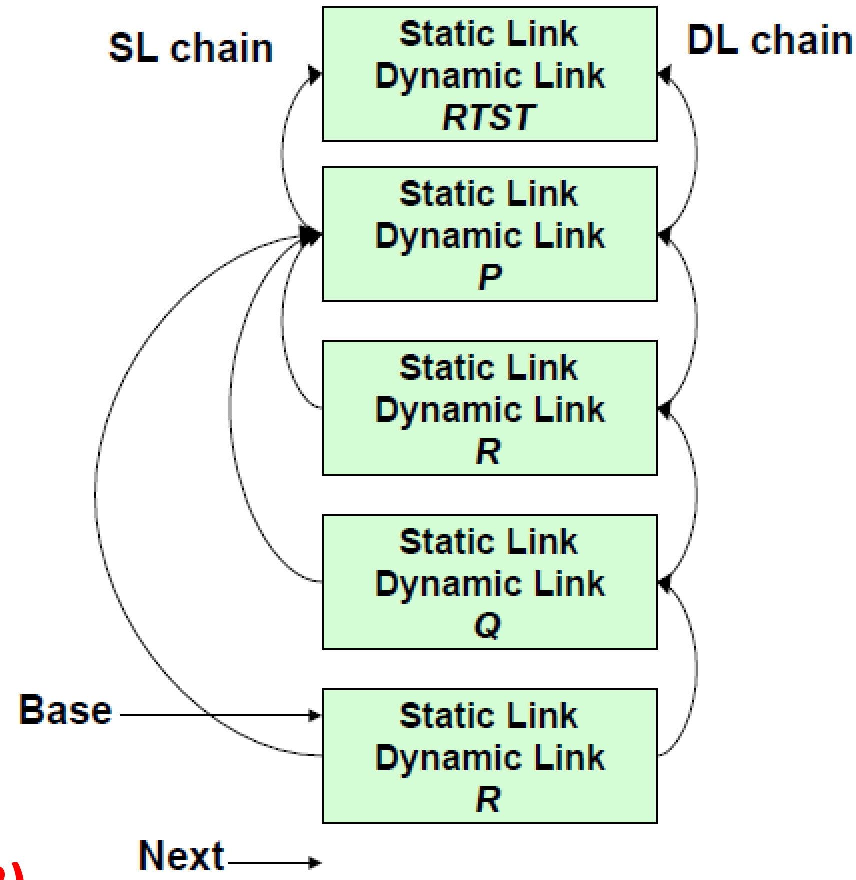
Allocation of nested procedure

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)

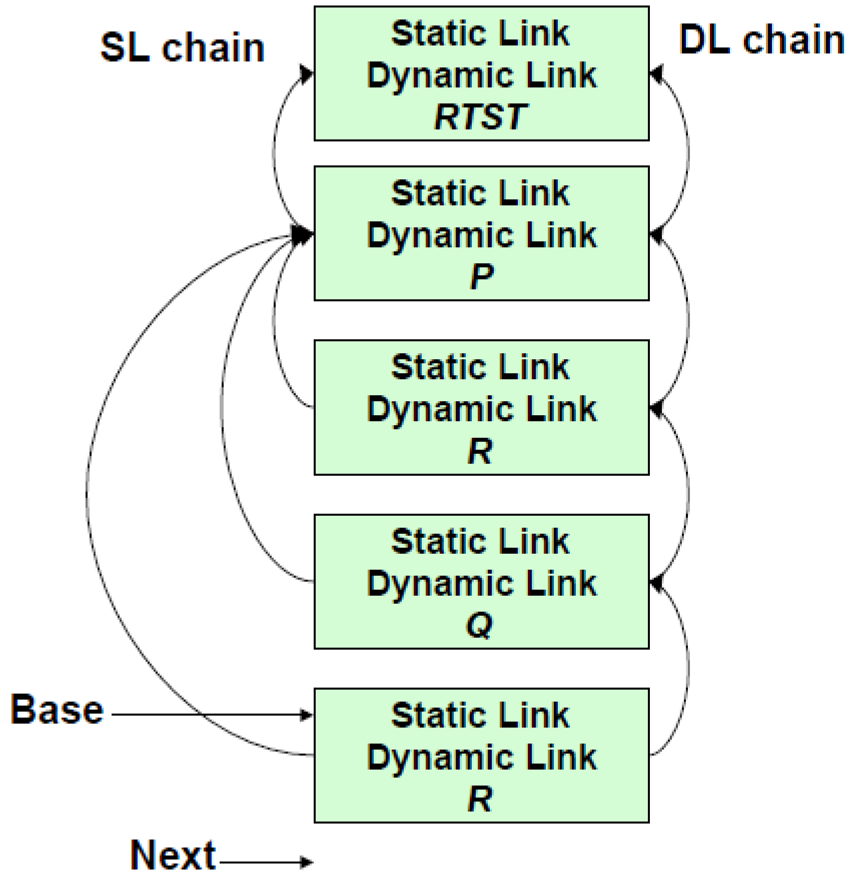
- for **P(2)→R(3),** 2 – 3 + 1 = 0;

  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;*

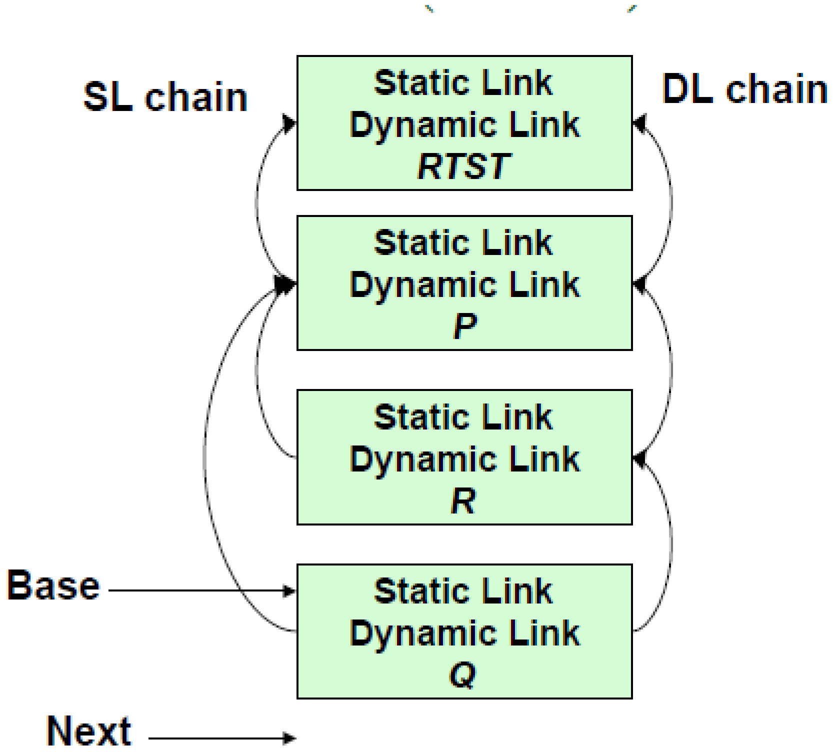
begin *R; end*

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;*

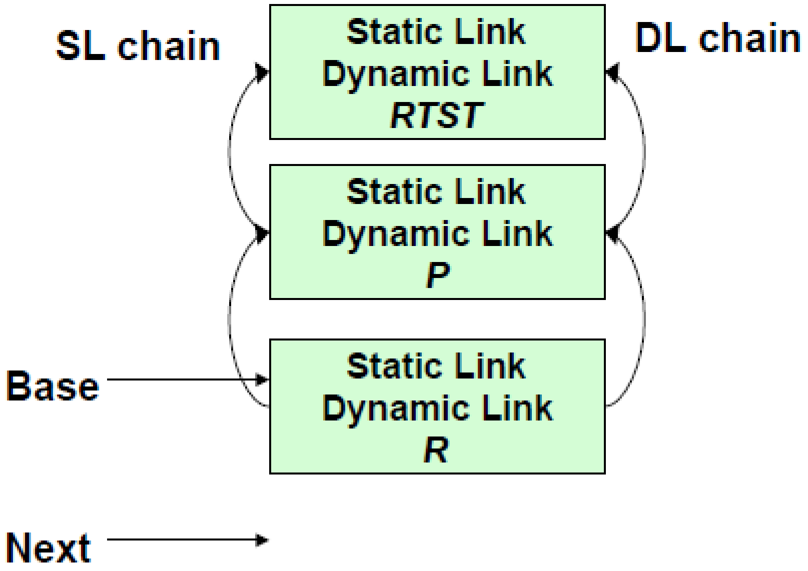
begin *R; end*

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;*

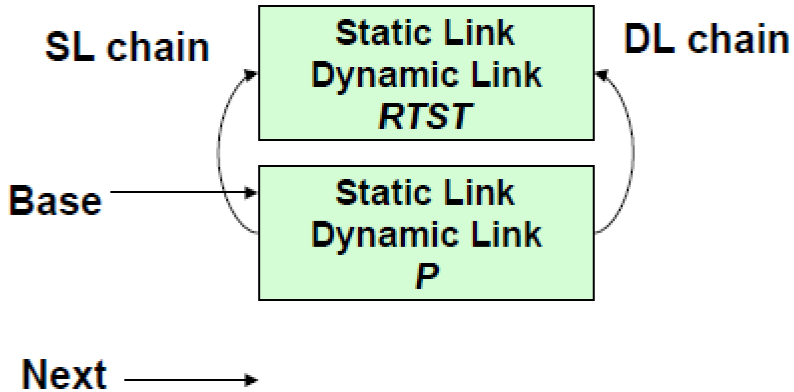
begin *R; end*

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;*

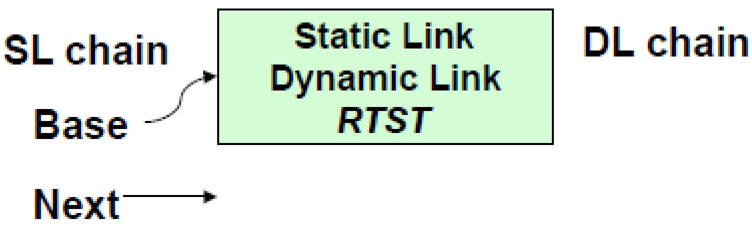
begin *R; end*

procedure *R;*

begin *Q; end*

begin *R; end*

begin *P; end*



- **Call sequence**

RTST ← P

Return from P

Static vs. Dynamic link

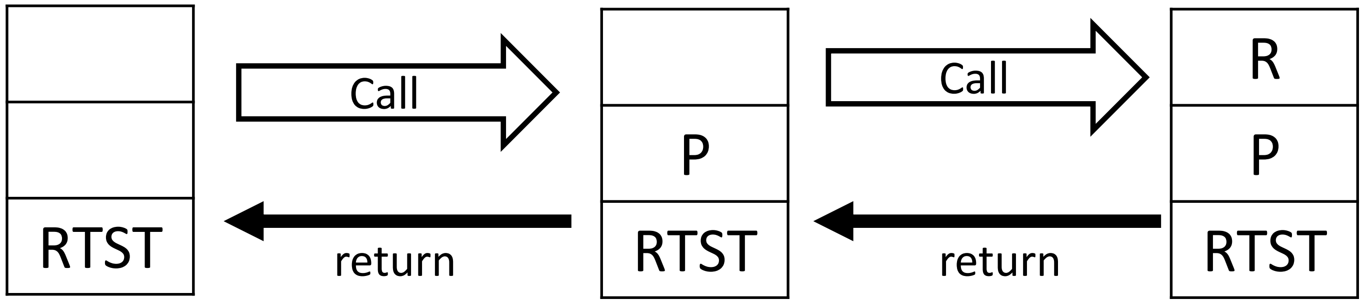
- **Static link:**
  - to access the global variables in the various activation records.

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

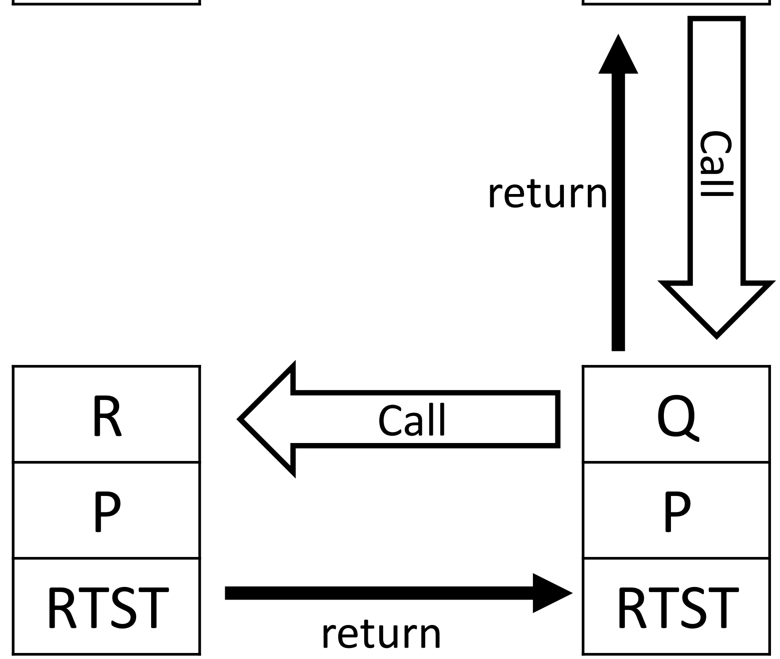
Display Stack

- 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**

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*

Example 2 (C-like structure is used)

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

Implementing Dynamic scope

1. Deep access method

2. Shallow access method

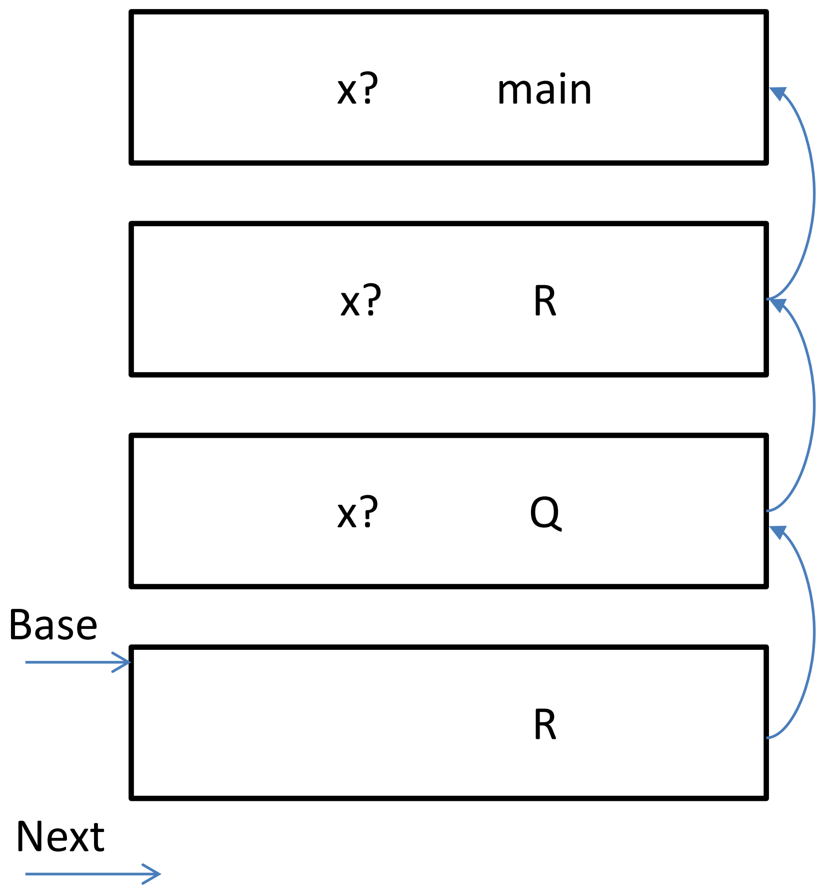
Deep Access Method

- 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

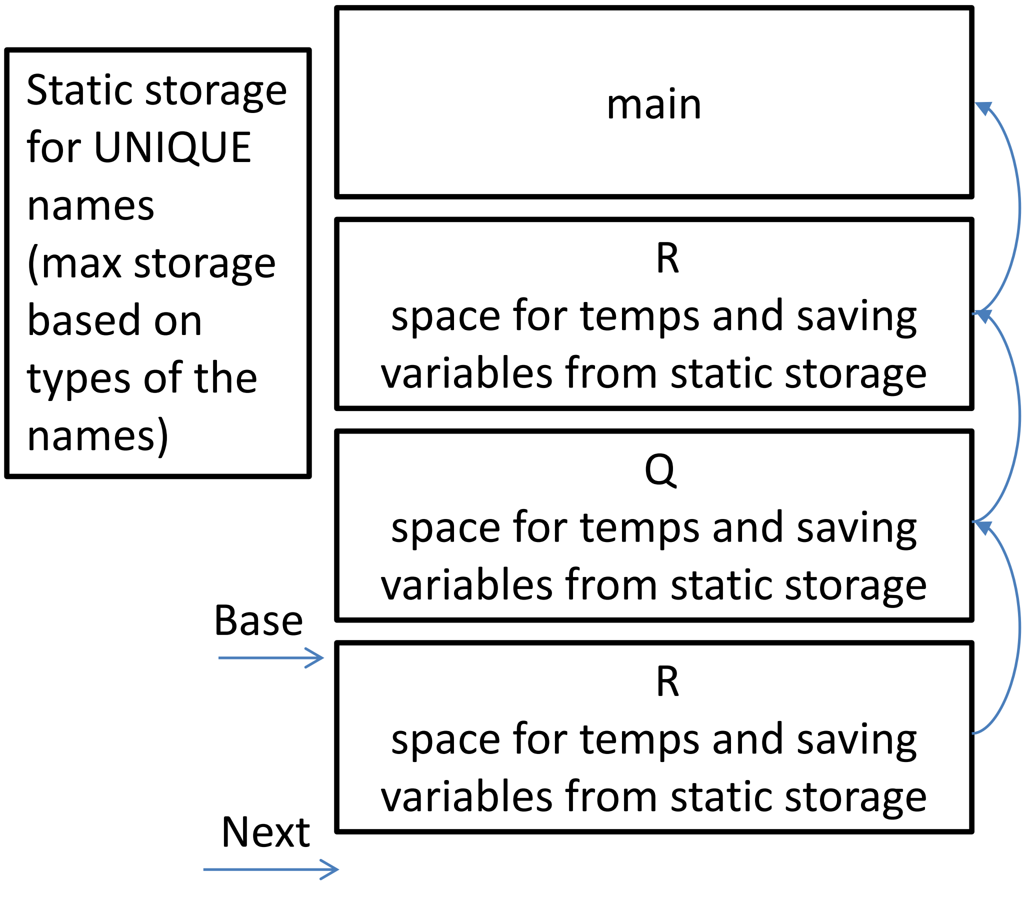
Currently, R is being accessed. (Base)

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.