Chapter 10

Implementing Subprograms

10.1 The General Semantics of Calls and Returns
10.2 Implementing "Simple" Subprogram
10.3 Implementing Subprograms with Stack-Dynamic Local Variables
10.4 Nested Subprograms
10.5 Blocks
10.6 Implementing Dynamic Scoping

"The purpose of this chapter is to explore methods for implementing subprograms in the major imperative languages. The discussion will provide the reader with some insight into how such language "works"?

The increased difficulty of implementing subprograms is caused by the need to include ① support for recursion and for ② mechanisms to access nonlocal variables.

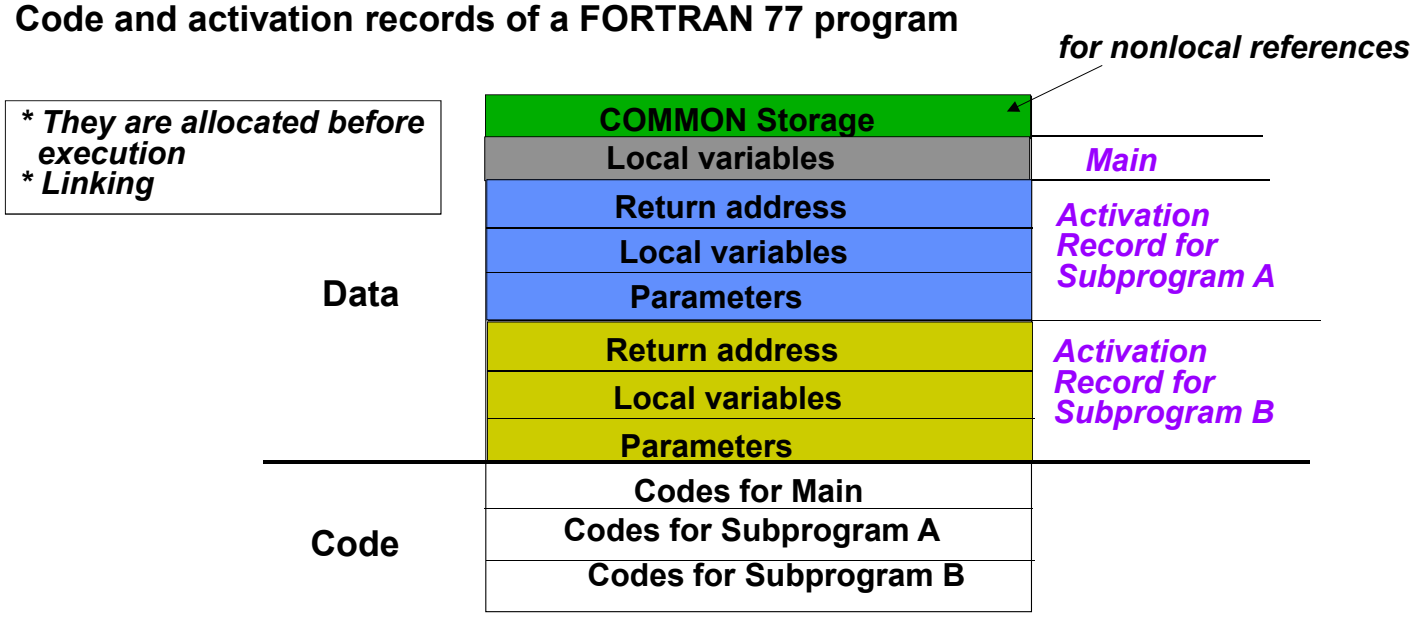
10.1 The General Semantics of Calls and Returns

- *Subprogram linkage*
  - the subprogram call and return operations of a language are together called its subprogram linkage
  - Any implementation method for subprograms must be based on the semantics of subprogram linkage
- *The actions associated with subprogram call*
  - includes parameter passing mechanism
  - causes storage to be allocated for local variables and bind those variables to that storage
  - save the execution status of calling program unit
  - arranges to transfer control to the code of subprogram and ensure that control can return to the proper place when the subprogram execution is completed
  - cause some mechanism to be created to provide access to nonlocal variables
- *The actions associated with subprogram return*
  - if the subprogram has parameters that are out mode and are implemented by copy, the first action of the return process is to move local values of the associated formal parameters to the actual parameters
    ⇔ how about by-reference or by-copy ?
  - deallocate the storage used for local variables
  - return the mechanism used for nonlocal references
  - control must be returned to the calling program unit

10.2 Implementing "Simple" Subprograms

- Subprograms in early version of FORTRAN
  - subprograms can not be recursive
  - All referencing of nonlocal variables in FORTRAN 77 is through COMMON
  - variables declared in subprograms are statically allocated
- The semantics of a FORTRAN 77 *subprogram call*
  1) Save the execution status of the current program unit
  2) Carry out the parameter-passing process
  3) Pass the return address to the callee
  4) Transfer control to the callee
- The semantics of a FORTRAN 77 *subprogram return*
  - If pass-by-value-result parameters are used, the current values of those parameters are moved to the corresponding actual parameters
  - If the subprogram is a function, the function value is moved to a place accessible to the caller
  - The execution status of caller is restored
  - Control is transferred back to the caller
- The call and return actions requires *storage* for the following:
  - Status information about the caller
  - Parameters
  - Return address
  - Function value for function subprograms

- A FORTRAN 77 subprogram consists of two parts each of which is fixed size:
  - the actual code of subprogram, which is static
  - the local variables and data areas for call/return actions
    - ⇔ the noncode part of a subprogram is associated with a particular execution, or activation, of subprogram, and is therefore called an *activation record*
    - ⇔ because FORTRAN 77 does not support recursion, there can be *only one active version of subprogram a given subprogram at a time*
      - ⇒ there can only *a single instance of the activation record* for a subprogram, and *can be statically allocated*
- Code and activation records of a FORTRAN 77 program



9.3 Implementing Subprograms with Stack-Dynamic Local Variables (with recursion)

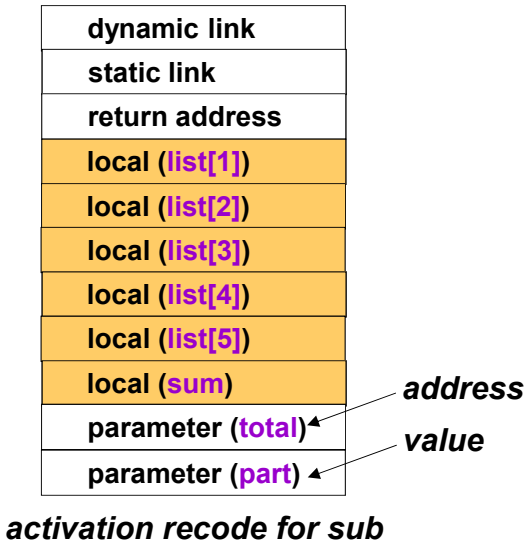
(1) More Complex Activation Records

- **Subprogram linkage in ALGOL-like languages** is **more complex** than the linkage of FORTRAN 77 subprograms for the following reasons :
  - Parameters are usually passed by two different methods
    - ⇔ For example, in Modular-2, they are passed by value or reference
  - Variables declared in subprograms are often dynamically allocated
  - *Recursion* adds the possibility of *multiple simultaneous activations of a subprogram*
    - ⇔ requires *multiple instances of activation records*
    - ⇔ each activation requires its own copy of the formal parameters and the dynamically allocated local variables, along with the return address
  - ALGOL-like languages use *static scoping* to provide access to nonlocal variables. Support for these nonlocal accesses must be part of the linkage mechanism
- **Creation of activation record**
  - Activating a procedure requires the dynamic creation of an instance of the activation record for the procedure
  - Because the call and return semantics specify that the subprogram last called is the first completed, it is reasonable to create instance of these activation records on *stack*
    - ⇔ Every procedure activation, whether recursive or nonrecursive, creates a new instance of an activation record on stack

- **Activation record**
  - ⇔ the format (and size) of an activation record for a given subprogram is known at compile time
  - ⇔ an activation record format is a template for instance of the activation record
  - *Local variables* are bound to storage within an activation record
  - *Static link* (also called *static scope pointer*)
    - ⇔ points to the activation record instance of an activation of the static parent
    - ⇔ used for accesses to nonlocal variables
  - *Dynamic link*
    - ⇔ a pointer to an instance of the activation record of caller (dynamic parent)
    - ⇔ in static-scoped languages, this link is used to in the destruction of current activation record instance when the procedure completes its execution
  - *Return address* : (code_segment, offset)
  - (Actual) Parameters
    - ⇔ the values or addresses provided by the caller

– **Example**

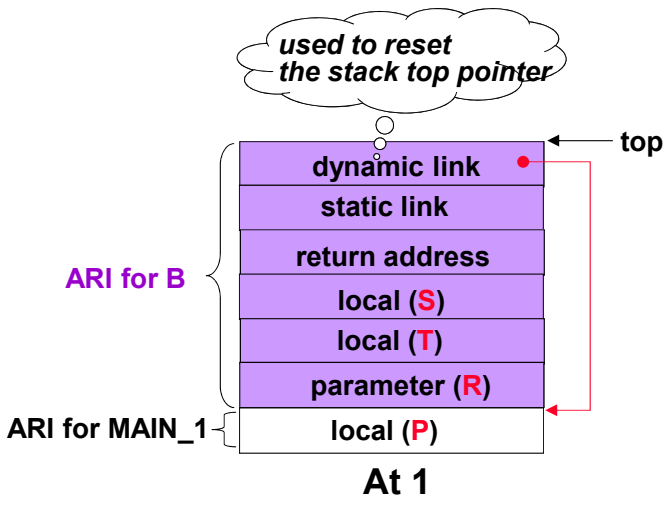
```
procedure sub (var total : real ;
                   part : integer) ;
   var list : array [1..5] of integer ;
   sum : real ;
   begin
      sum = total + sum ;
   end ;
```



activation recode for sub

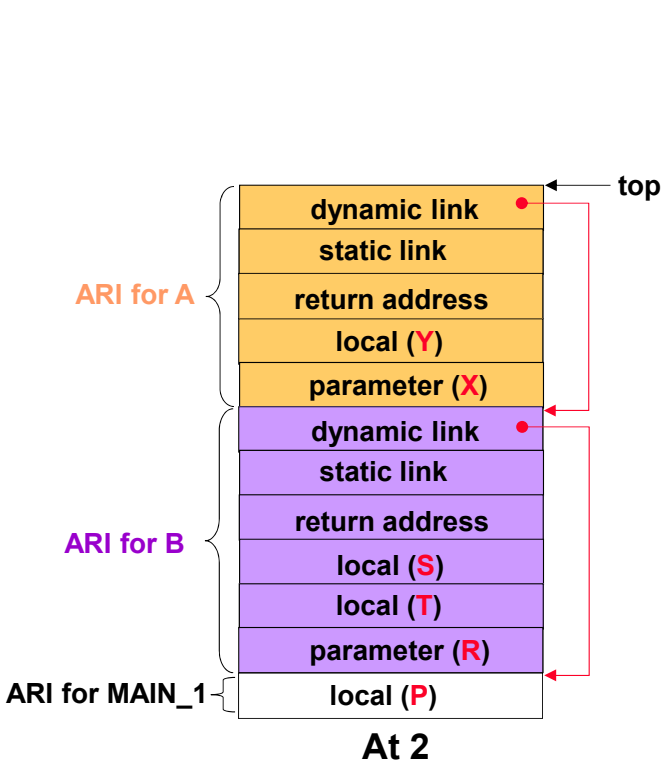
(2) Example Without Recursion and Nonlocal References

```
program MAIN_1
  var P : real;
  procedure A(X:integer);
    var Y:boolean;
    procedure C(Q:boolean);
      begin
        P=P+1;                    <-- 3
      end
    begin
      …                           <-- 2
      C(Y);
      …
    end {end of procedure A}
  procedure B(R:real);
    var S,T;
    begin
      …                           <-- 1
      A(S);
      …
    end ({end of procedure B}
  begin {Main_1}
    …
    B(P);
    …
  end
```

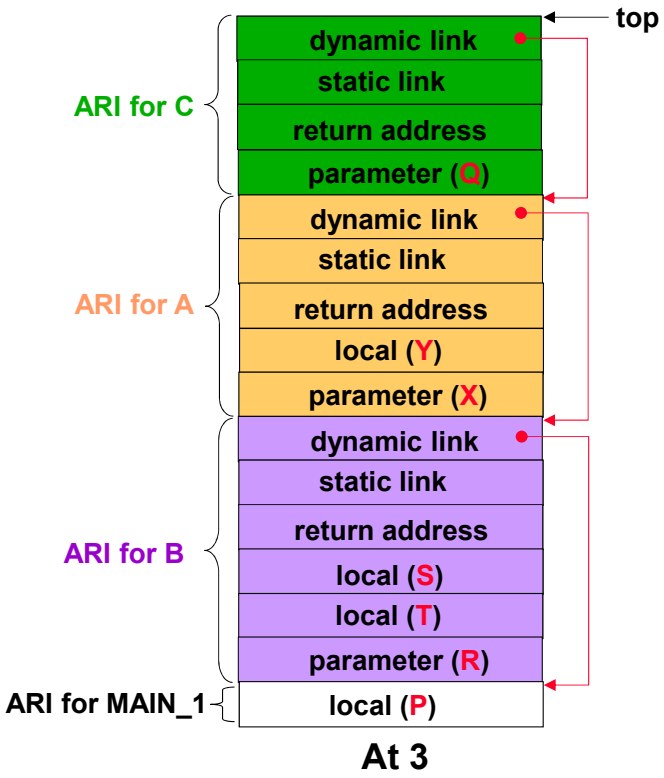


At 1

MAIN_1 -> B(P) -> A(S) -> C(Y)



At 2



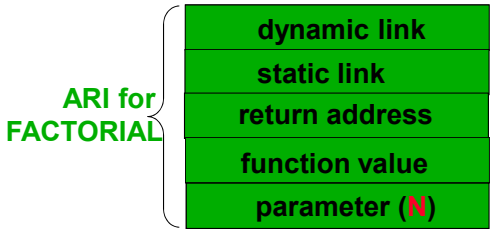
At 3

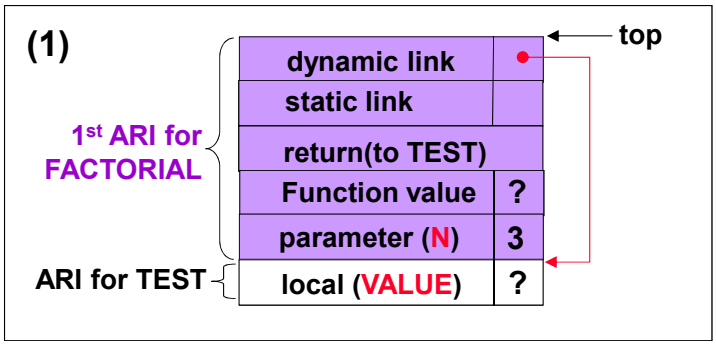
- **Dynamic Chain** (or *call chain*)
  - the collection of dynamic links present in the stack at a given time
  - represents the dynamic history of how execution got to its current position
- **Local_Offset**
  - references to local variables can be represented in the code as *offsets from the beginning of the activation record* of the local scope -> local_offset
  - the local_offset of a variable in an activation record can be *determined at compile time*, using the order, types, and sizes of variables declared in the procedure associated with the activation record

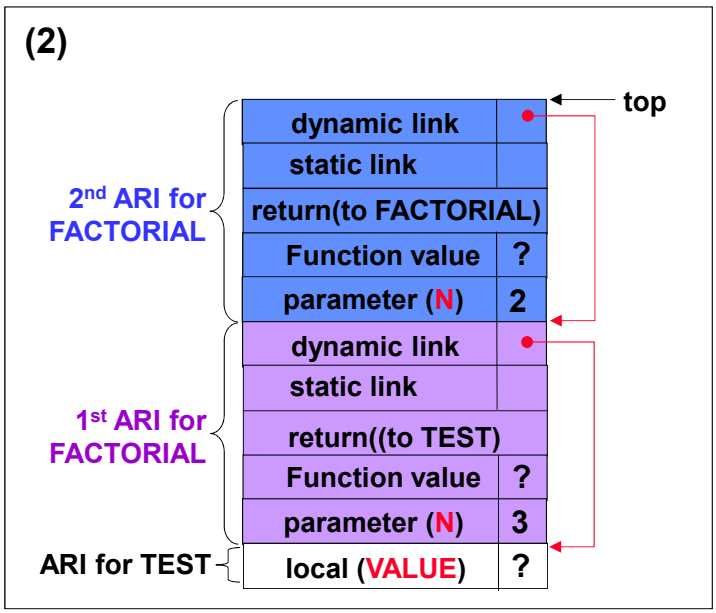
(3) Recursion

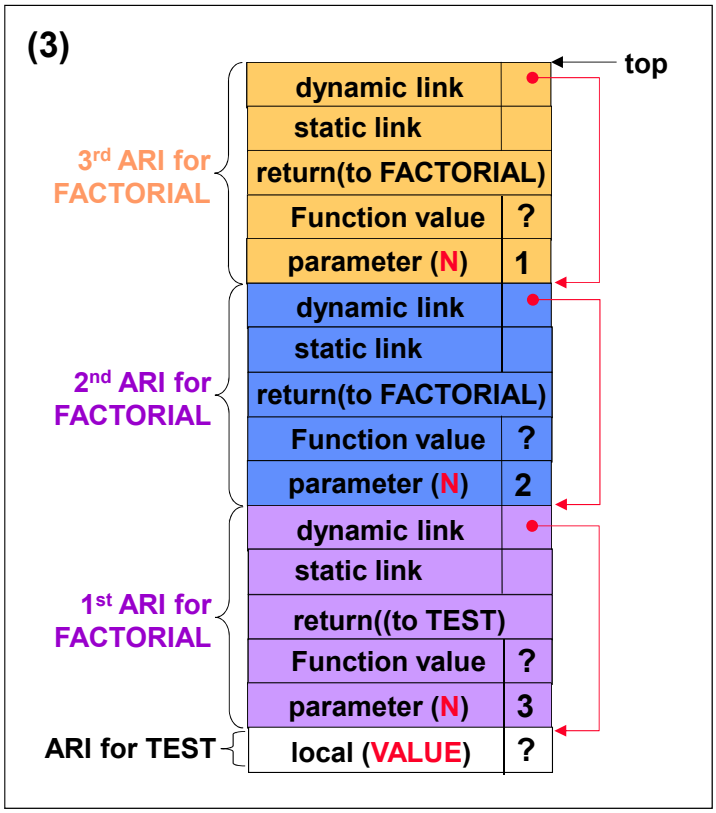
- Example : Using recursion to Compute the factorial

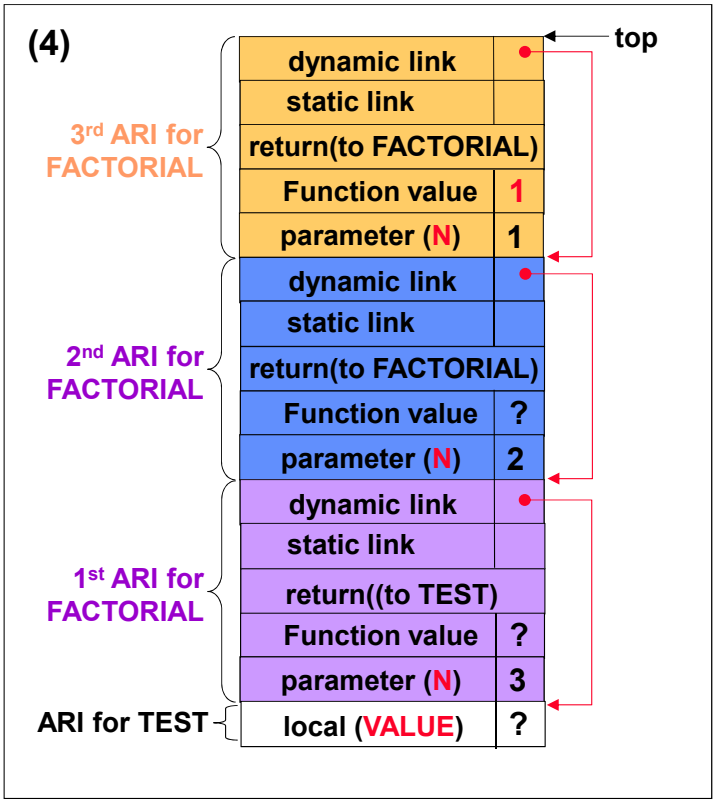
```
Program TEST
  var VALUE:integer;

  function FACTORIAL(N:integer);
    begin                                  1
      if N<=1
         then FACTORIAL:=1
         else FACTORIAL:= N*FACTORIAL(N-1);
    end                                    2

  begin
    VALUE:=FACTORIAL(3);
    writeln("factorial 3 is:", VALUE)      3
  end.
```

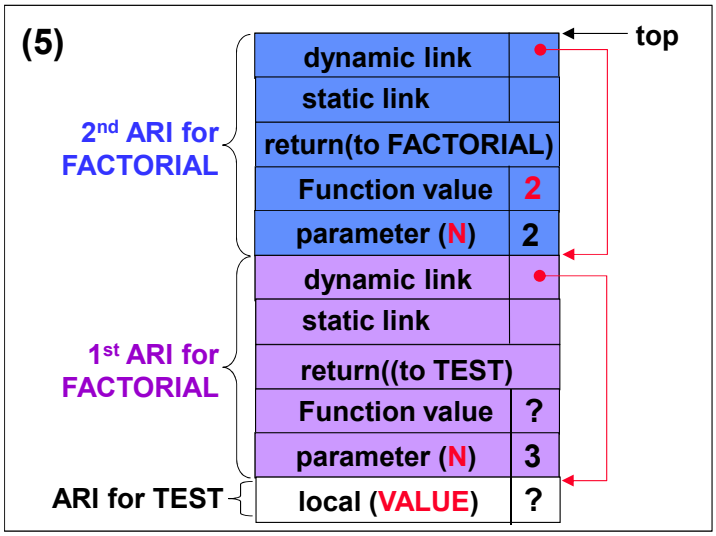


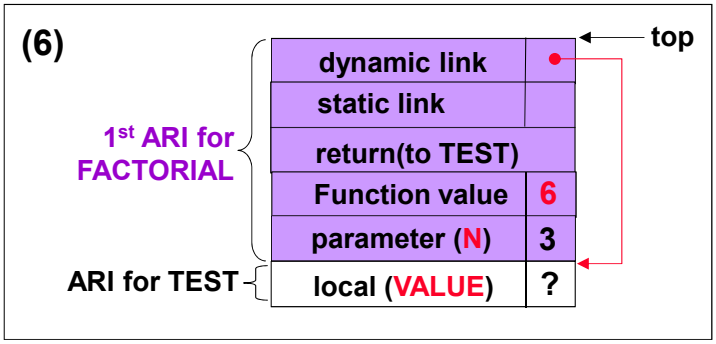

(1)
top
dynamic link
static link
1st ARI for FACTORIAL
return(to TEST)
Function value ?
parameter (N) 3
ARI for TEST
local (VALUE) ?

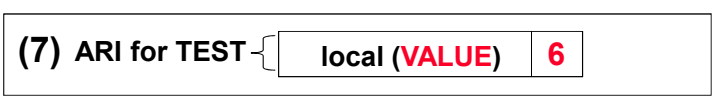

(2)
top
dynamic link
static link
2nd ARI for FACTORIAL
return(to FACTORIAL)
Function value ?
parameter (N) 2
dynamic link
static link
1st ARI for FACTORIAL
return((to TEST)
Function value ?
parameter (N) 3
ARI for TEST
local (VALUE) ?


(3)
top
dynamic link
static link
3rd ARI for FACTORIAL
return(to FACTORIAL)
Function value ?
parameter (N) 1
dynamic link
static link
2nd ARI for FACTORIAL
return(to FACTORIAL)
Function value ?
parameter (N) 2
dynamic link
static link
1st ARI for FACTORIAL
return((to TEST)
Function value ?
parameter (N) 3
ARI for TEST
local (VALUE) ?


(4)
top
dynamic link
static link
3rd ARI for FACTORIAL
return(to FACTORIAL)
Function value 1
parameter (N) 1
dynamic link
static link
2nd ARI for FACTORIAL
return(to FACTORIAL)
Function value ?
parameter (N) 2
dynamic link
static link
1st ARI for FACTORIAL
return((to TEST)
Function value ?
parameter (N) 3
ARI for TEST
local (VALUE) ?


(5)
top
dynamic link
static link
2nd ARI for FACTORIAL
return(to FACTORIAL)
Function value 2
parameter (N) 2
dynamic link
static link
1st ARI for FACTORIAL
return((to TEST)
Function value ?
parameter (N) 3
ARI for TEST
local (VALUE) ?


(6)
top
dynamic link
static link
1st ARI for FACTORIAL
return(to TEST)
Function value 6
parameter (N) 3
ARI for TEST
local (VALUE) ?


(7) ARI for TEST
local (VALUE) 6

10.4 Nested Subprogram

- all variables that can be nonlocally accessed are in activation record instances, and therefore are somewhere in the stack

- A reference to a nonlocal variables : two-step
  1) to find the instance of the activation record in the stack where the variable was allocated
  2) to use the local_offset of the variable (within the activation record instance) to actually access it

- Semantic rules of static-scoped language
  - in a given subprogram, only variables that are declared in static ancestor scopes cab be nonlocally accessed
  - activation record instances of all the static ancestors are guaranteed to exist on the stack when variables in them are referenced by a nested procedure
    ⇔ A procedure is callable only when all of its static ancestor program units are active
  - the correct declaration is the first one found when looking through the enclosing scopes, most closely nested first
    ⇔ So to support nonlocal references, it must be possible *to find all of the instances of activation records in the stack that correspond to those static ancestors*
    ⇒ Using *Static chain*
    ⇒ Using *Display*

- Static Chains
  - a chain of static links that connect certain activation record instances in the stack
    ⇔ It links all the static ancestors of an existing subprogram, in order of static parent first
  - when a reference is made to a nonlocal variable, the activation record instance containing the variable can be found *by searching the static chain until a static ancestor activation instance is found that contains the variable*
  - because the nesting of scope is *known at compile time*, the compiler can determine not only that a reference is nonlocal, but also the length of the static chain needed to reach the activation record instance that actually contains the nonlocal object
  - Static_depth
    ⇔ an integer associated with a static scope that indicates how deeply it is nested in the outermost scope
  - Nesting_depth (or chain_offset) of reference
    ⇔ the length of the static chain needed to reach the correct activation record instance for a nonlocal reference
    ⇔ the difference between the static_depth of the procedure containing the reference to X and the static_depth of the procedure containing the declaration for X
  - Actual reference : (*chain_offset, local_offset*)

```
program A ;
    procedure B ;
        procedure C ;
        end; { of procedure C }
    end; { of procedure B)
end ;
```

- static_depth
  A : 0, B : 1, C : 2

-chain_offset when C refers the variable in A : 2

```
MAIN_2
var X : integer
  BIGSUB
  var A, B, C : integer ;
    SUB1
    var A, D : integer ;
    A : = B + C ;

    SUB2
    var B, E : integer ;

      SUB3
      var C, E : integer ;
      SUB1 ;
      E := B + A ;

    ....
    SUB3 ;
    ....
    A := D + E ;

  SUB2 ;

BIGSUB ;
```

- Calling Sequence
  - MAIN_2 calls BIGSUB
  - BIGSUB calls SUB2
  - SUB2 calls SUB3
  - SUB3 calls SUB1

A : (0, 3)
B : (1, 4)
C : (1, 5)

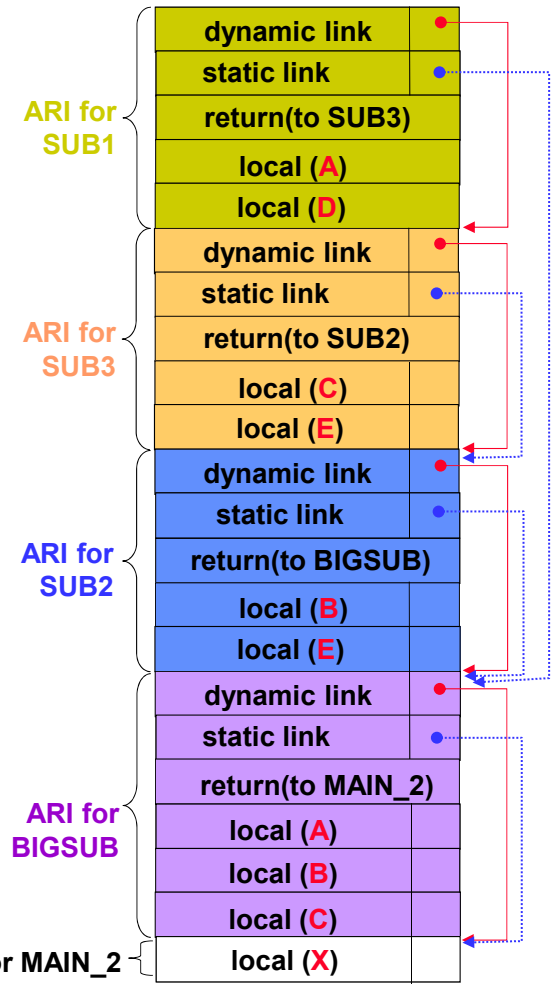
E : (0, 4)
B : (1, 3)
A : (2, 3)

A : (1, 3)
D : ? (Error)
E : (0, 4)

0	dynamic link
1	static link
2	return address
3	local 1
4	local 2

activation record format

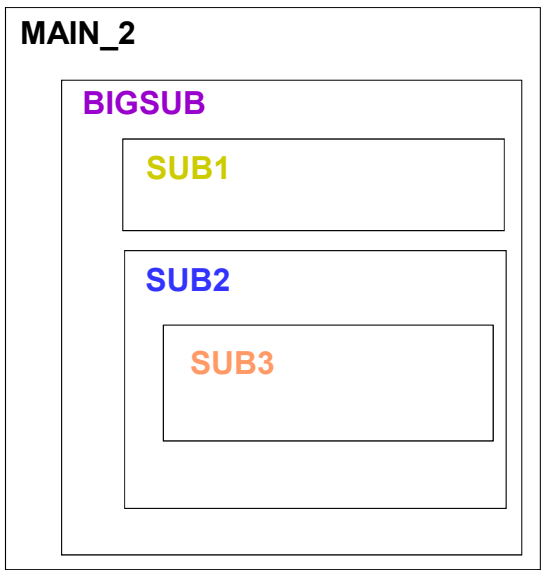
(chain offset, local-offset)



Calling Sequence

MAIN_2 -> *BIGSUB* -> *SUB2* -> *SUB3* -> *SUB1*

Program Structure



- How the static chain is maintained during program execution ?
  - ⇔ actions required at subprogram return
    - ⇒ trivially, nothing to do because its activation record is removed from the stack
  - ⇔ actions required at subroutine call : the most recent activation record instance of the parent scope must be found at the time of the call,
    - 방법 1)
      - → looking at activation record instance on the dynamic chain until the first one of parent scope is found, at run-time
    - 방법 2)
      - → at compiler time : *compiler compute the nesting_depth between caller and the procedure that declared the called program.*
      - → at the time of the call : the static link of the called procedure's activation record instance is determined by *moving down the static chain of the caller the number of links equal to the nesting depth computed at compiler time*
- Problems of static chain method
  - ⇔ references to variables in scope beyond the static parent are costly
    - ⇒ the static chain must be followed, one link per enclosing scope from the reference to the declaration, to accomplish the access
  - ⇔ it is difficult for a programmer working on time-critical program to estimate the costs of nonlocal references, since the cost of each reference depends on the depth of nesting

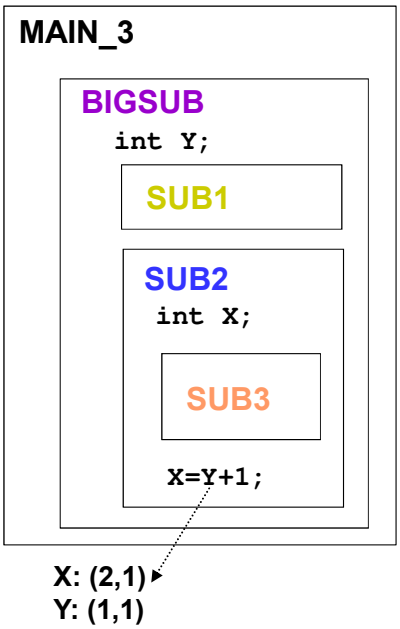
- Display
  - the static links are collected in a single array called a display, rather than being stored in the activation records
  - the contents of the display at any specific time are *a list of addresses of the accessible activation record instances* - one for each active scope - in the order in which they are nested
  - nonlocal reference : (display_offset, local_offset)
  - access to nonlocals using a display
    - ⇔ the link to correct activation record, which resides in the display, is found using a statically computed value called the *display_offset*
    - ⇔ the local offset within the activation record instance is computed and used exactly as with static chain implementations
  - In general, the pointer at position k of the display points to an activation record instance for a procedure with a static depth of k
  - How to modify the display to reflect the new scope situation ?
    - ⇔ the display modification required for a call to procedure P, which has a static_depth of k, is
      - ⇒ Save, in the new activation record instance, a copy of the pointer at position k in the display
      - ⇒ Place the link to the activation record instance for P at position k in the display
    - ⇔ at termination, the saved pointer in the activation record instance of the terminating subprogram to be placed back in the display

– Example : a call to procedure P by procedure Q (Q -> P)

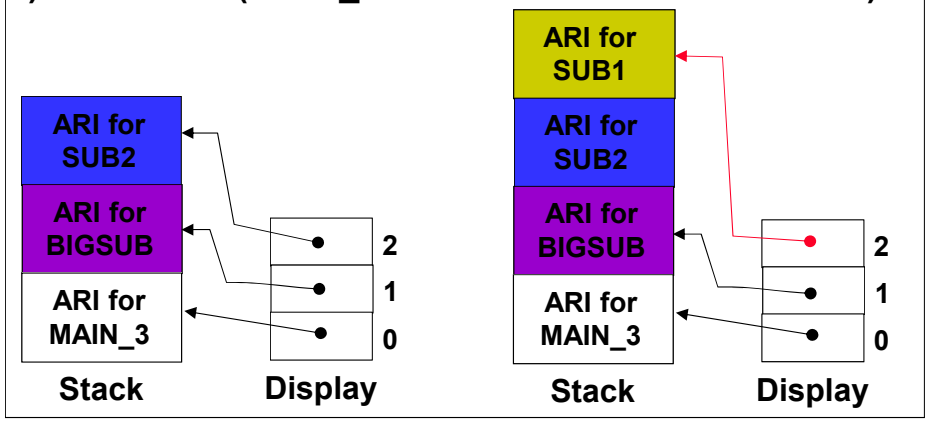
⇔ Qsd = Psd
⇔ Qsd < Psd
⇔ Qsd > Psd

Psd : static_depth of P
Qsd : static_depth of Q

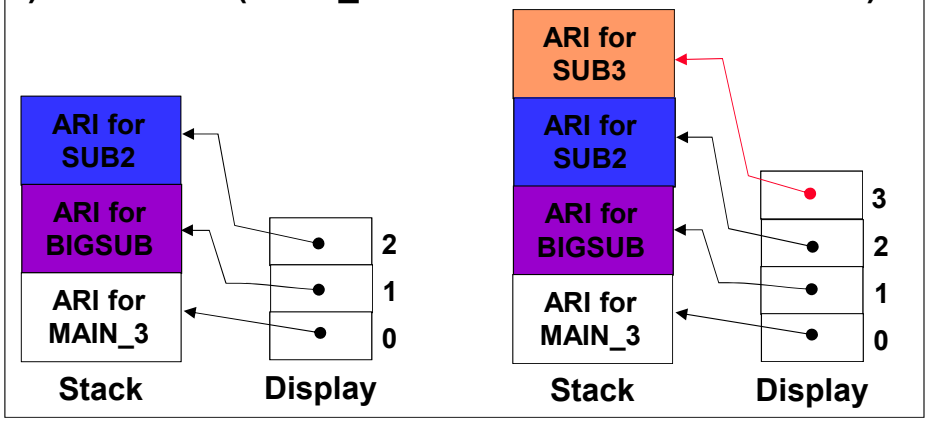
Program Structure



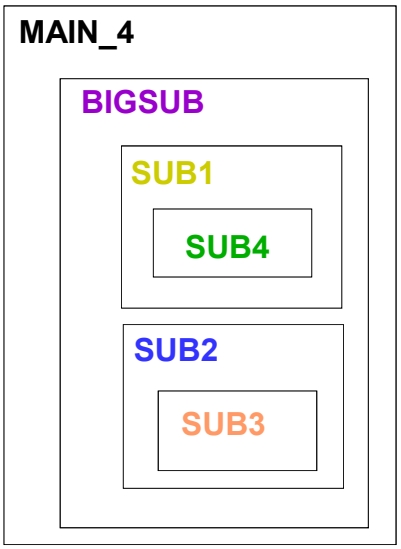
1) Qsd = Psd (MAIN_3 -> BIGSUB -> *SUB2 -> SUB1*)



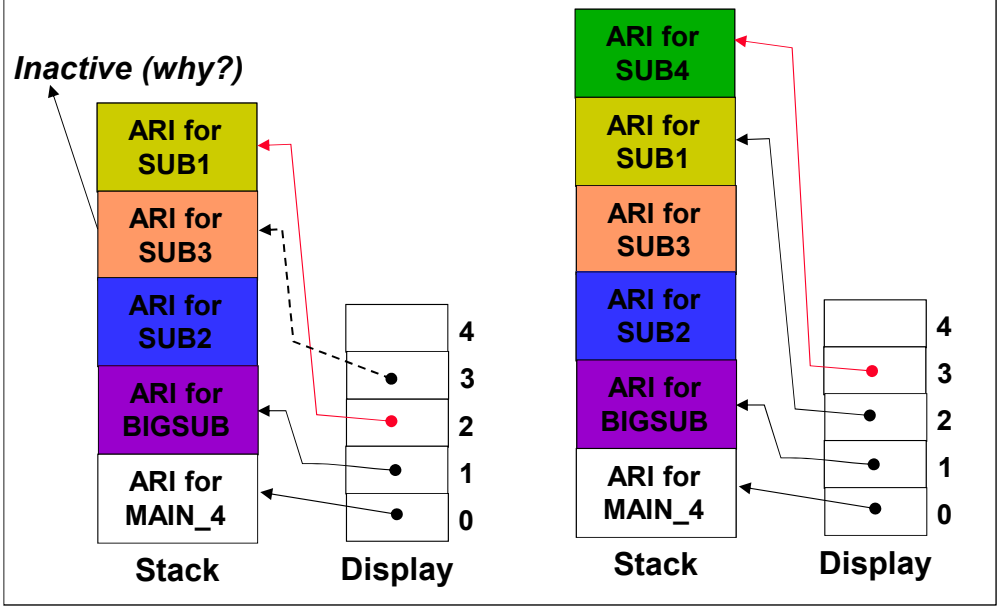
2) Qsd < Psd (MAIN_3 -> BIGSUB -> *SUB2 -> SUB3*)

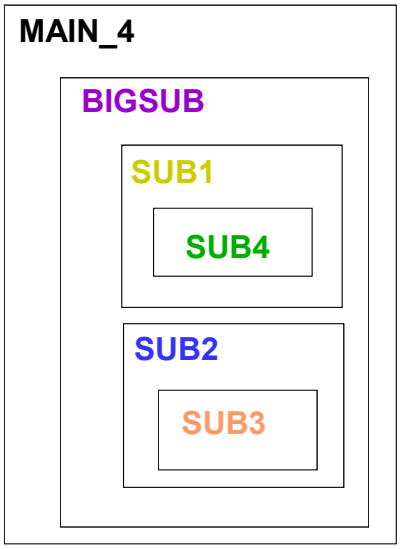


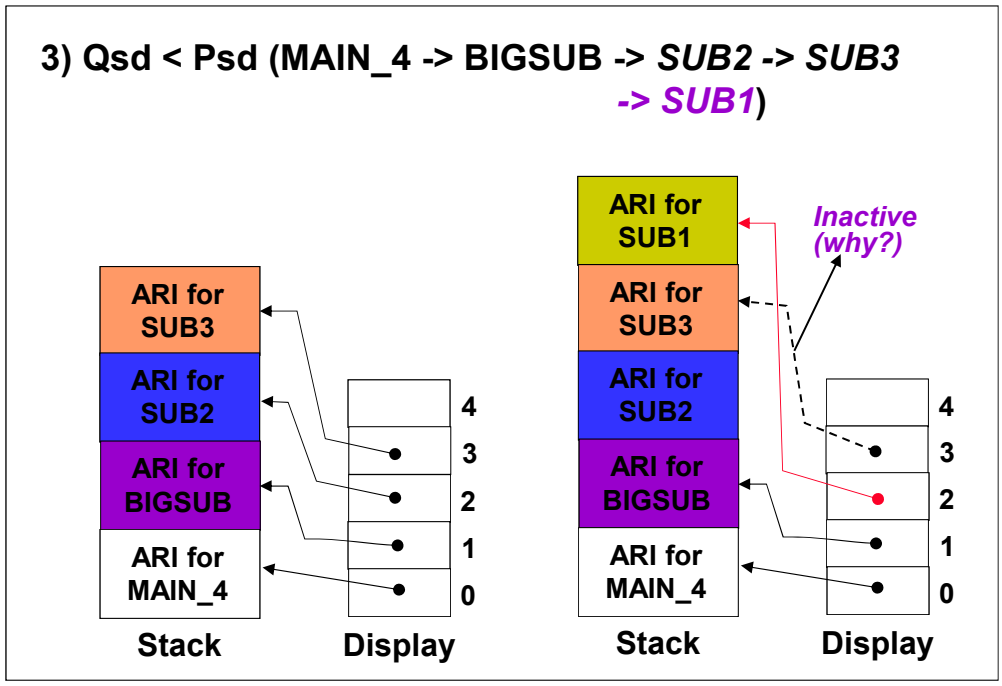
Program Structure



2') Qsd < Psd (MAIN_4 -> BIGSUB -> *SUB2 -> SUB3 -> SUB1 -> SUB4*)







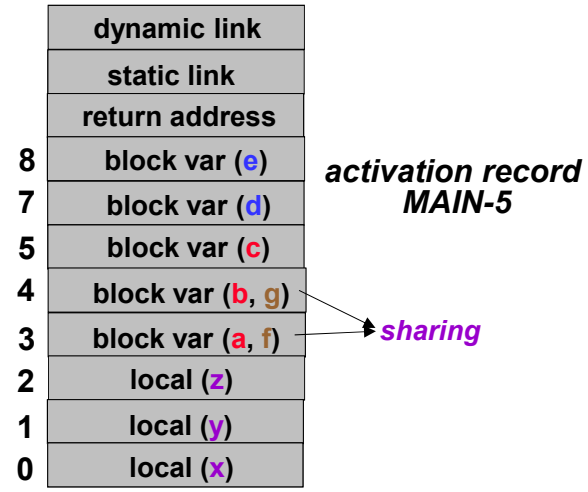
- Implementation of display
  - ⇔ the maximum size of display, which is the maximum static_depth of any subprogram in the program, can be determined by the compiler
  - ⇔ can be stored as a run-time static array in memory
    - ⇒ nonlocal accesses cost one more memory cycle than local accesses, if the machine has indirect addressing through memory location
  - ⇔ to place the display in registers
    - ⇒ do not require the extra memory cycle

- Static chaining *vs.* display method
  - references to local variables would be slower with a display than with static chains if the display is not stored in registers (adds a level of indirection)
  - references to nonlocal variables that are more than one static level away will be faster with display than static chain
  - the time is equal for all nonlocal references when a display is used
  - the maintenance at a procedure call is faster with static chains, unless the called program is more a few static levels away
  - Overall comparison
    - ⇔ *displays* are better if there is deep static nesting and many references to distant nonlocal variables
    - ⇔ *static chaining* is better if there are few nesting levels and few references to distant nonlocal variables, which is the more common situation
      - ⇒ usual nesting level is less than three

10.5 Blocks

- **Block** = **compound statement** + **data declaration**
- **Implementation** of Block
  - 방법 1) treated as **parameterless procedures** that are always called from the same place in the program
    - ⇔ maximum nesting grows -> display size grows
  - 방법 2) the amount of space required for block variables can be **allocated next to local variables** in the activation record
    - ⇒ **Offsets** for all block variables can be **statically computed**, so block variables can be addressed exactly as if they were local variables

```
MAIN-5() {
  int x, y, z ;
  while (… …) {
    int a, b, c ;
    …
    while (… …) {
      int d, e ;
      …
    }
  }
  while (… …) {
    int f, g ;
    …
  }
}
```



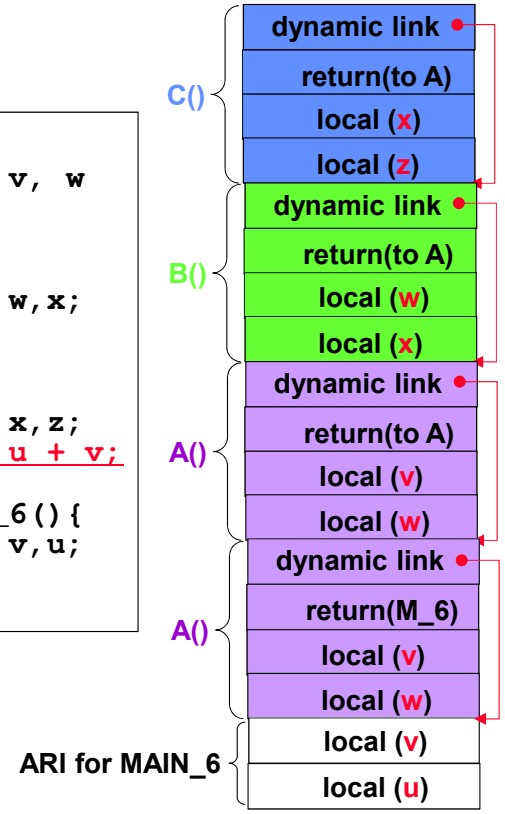
10.6 Implementing Dynamic Scoping

- there are at least two distinct ways in which nonlocal references in a dynamic-scoped language can be implemented : *deep access* and *shallow access*

(1) Deep Access

- the nonlocal reference can be resolved by **searching through the declaration in the other subprograms that are currently active**, beginning with the one most recently activated
  - dynamic-chain is followed
  - this method is called deep access because access may require searching deep in the stack
- In a dynamic-scoped language, there is **no way to determine at compile time the length of the chain** that must be searched
  - typically slow than static scoped language
- Activation record must **store the names of variables** for the search process
- Example : Calling sequence : MAIN_6 -> A -> A -> B -> C

```
A() {
  int v, w
  …
}
B() {
  int w,x;
  …
}
C() {
  int x,z;
  x:= u + v;
}
Main_6() {
  int v,u;
  …
}
```



(2) Shallow Access

– Variables declared in subprograms are not stored in the activation records of those subprograms

- Implementations

방법1) to have a separate stack for each variable name in a complete program

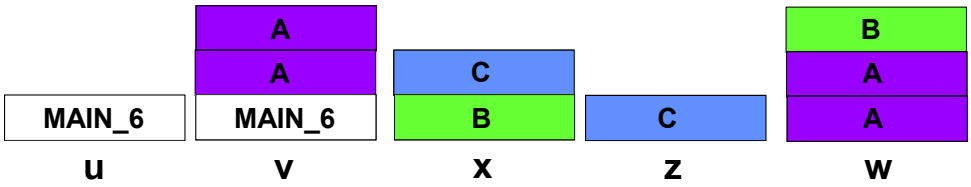
⇔ every time a new variable with a particular name is created by a declaration at the beginning of a subprogram activation, it is given a cell on the stack for its name

⇔ every reference to the name is to the variable on top of the stack

⇔ fast references to variables, but maintaining the stacks at the entrances and exits of subprogram is expensive

```
A(){
 int v, w
 …
}
B(){
 int w,x;
 …
}
C(){
 int x,z;
 x:= u + v;
}
Main_6(){
 int v,u;
 …
}
```

MAIN_6 -> A -> A -> B -> C



<Homework> #1 ~ #6

<Homework>

```
procedure Bigsub is
  MySum : Float;
  procedure A is
    X : Integer;
    procedure B(Sum : Float) is
      Y, Z : Float;
      begin -- of B
        . . .
        C(Z)
        . . .
      end; -- of B
    begin -- of A
      . . .
      B(X);
      . . .
    end; -- of A
  procedure C(Plums : Float) is
    begin -- of C
      . . . ①
  end; -- of C
  L : Float;
  begin -- of Bigsub
    . . .
    A;
    . . .
  end; -- of Bigsub
```

1. Show the stack with all activation record instances, including static and dynamic chains, when execution reaches position ① in the following skeletal program. Assume **Bigsub** is at level 1.

2. Show the stack with all activation record instances, including static and dynamic chains, when execution reaches position ① in the following skeletal program. Assume **Bigsub** is at level 1.

```
procedure Bigsub is
  procedure A is
    procedure B is
      begin -- of B
        . . . ①
      end; -- of B
    procedure C is
      begin -- of C
        . . .
        B;
        . . .
      end; -- of C
    begin -- of A
      . . .
      C;
      . . .
    end; -- of A
  begin -- of Bigsub
    . . .
    A;
    . . .
end; -- of Bigsub
```

```
procedure Bigsub is
  procedure A(Flag:Boolean) is
    procedure B is
      . . .
      A(false);
    end; -- of B
  begin -- of A
    if flag
       then B;
       else C;
    . . .
  end; -- of A
  procedure C is
    procedure D is
      . . . ①
    end; -- of D
    . . .
    D;
  end; -- of C
  begin -- of Bigsub
    . . .
    A(true);
    . . .
  end; -- of Bigsub
```

Bigsub calls A
A calls B
B calls A
A calls C
C calls D

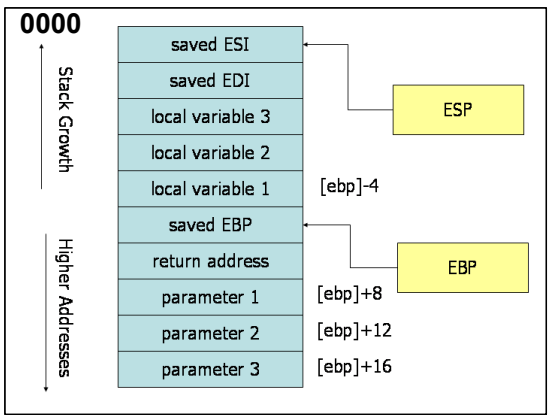
3. Show the stack with all activation record instances, including static and dynamic chains, when execution reaches position ① in the following skeletal program. Assume Bigsub is at level 1.
4. Show the stack with all activation record instances, including dynamic chain, when execution reaches position ① in the following skeletal program. This program uses the deep-access method to implement dynamic scoping.

```
void fun1() {
  float a;
  . . .
}
void fun2() {
  int b, c;
  . . .
}
void fun3() {
  float d;
  . . . ①
}
void main() {
  char e, f, g;
  . . .
}

main calls fun2
fun2 calls fun1
fun1 calls fun1
fun1 calls fun3
```

5. Assume that the program of Problem 4 is implemented using the shallow-access method using a stack for each variable name. Show the stacks for the time of the execution of fun3, assuming execution found its way to that point through the sequence of calls shown in Problem 4.
6. Although local variables in Java methods are dynamically allocated at the beginning of each activation, under what circumstances could the value of a local variable in a particular activation retain the value of the previous activation?

- **Cdecl (C declaration)**
  - **a calling convention that originates from the C programming language and is used by many C compilers for the x86 architecture**
  - **subroutine arguments are passed on the stack.**
  - **Integer values and memory addresses are returned in the EAX register, floating point values in the ST0 x87 register.**
  - **Registers EAX, ECX, and EDX are caller-saved, and the rest are callee-saved.**

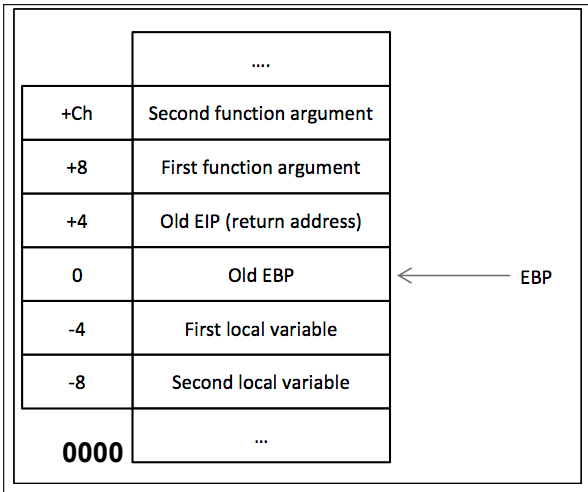
```
int callee(int, int, int);

int caller(void) {
  int ret;
  ret = callee(1, 2, 3);
  ret += 5;
  return ret;
}
```

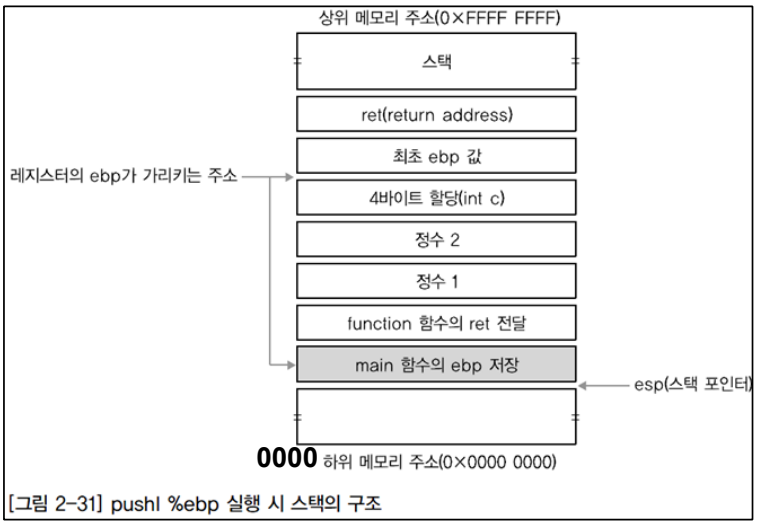



```
caller:
    ; make new call frame
    push        ebp  /* 이전 스택의 base 주소를 저장 */
    /*현재 스택의 꼭대기를 새로운 스택의 base로 설정 */
    mov         ebp, esp
    ; push call arguments
    push        3
    push        2
    push        1
    ; call subroutine 'callee'
    call        callee
    ; remove arguments from frame
    add         esp, 12
    ; use subroutine result
    add         eax, 5
    ; restore old call frame
    pop         ebp
    ; return
    ret
```

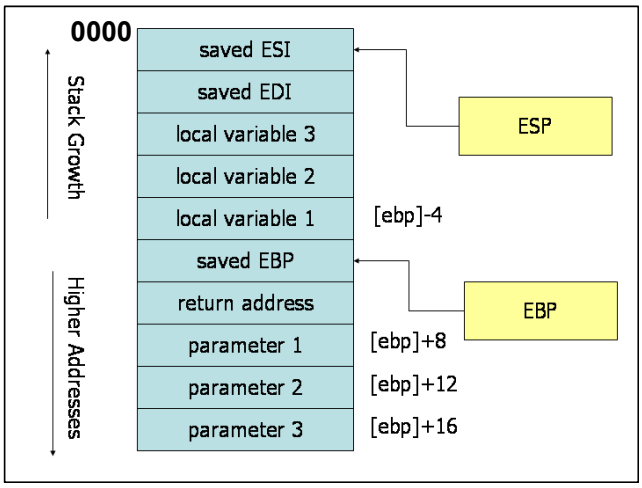
return value of function call is stored

ESP : Extented Stack Pointer
EBP : Extented Base Pointer





[그림 2-31] pushl %ebp 실행 시 스택의 구조



- Register
  - CPU안에서 사용하는 메모리.
  - 할당할수 있는 크기는 굉장히 작으나, 속도는 굉장히 빠름.
  - 레지스터는 종류에 따라 2가지로 나뉠수 있음.
  - 1. 범용 레지스터(Genaral Register)
    - EAX(Accmulator) / EBX(Base Register) / ECX(Count) / EDX(Data)로 나뉨.
    - 일반적인 변수로 생각하여 사용하면 됨.(Ex - int a)
  - 2. 포인터 레지스터(Pointer Register)
    - EBP(Base Pointer) / ESP(Stack Pointer) / EIP(Instruction Pointer)로 나뉨.
    - EBP - 스택에 가장 밑부분(바닥)
    - ESP - 스택에 가장 윗부분(Top)
    - 포인터 변수로 생각하여 사용.(Ex - int *a)
  - 레지스터의 길이는 CPU에 따라 바뀌지만, 레지스터에 대해서 나누어서 사용할 수 있음.
- Stack
  - LIFO(Last In First Out)구조로 가지고 있음(후입선출)
  - 지역변수 저장 / 매개변수 전달 / 임시데이터 백업 / 함수호출 / 복귀 정보 저장시 사용함.
- 함수를 실행했을경우 변화
  - EBP(가장 아랫부분)과 ESP(가장 윗주소) 레지스터를 이용해서 스택의 위치를 가르킴.
  - 변수를 추가 / 함수를 호출할 경우 EBP와 ESP를 이용하여 스택 위치를 변경함.
  - Prolog / Epilog
    - 1.Prolog
      - 함수를 호출했을 경우 호출 받은 함수가 Stack에서 독립적으로 사용하기 위해서 스택에 할당하는 작업.
      - 호출되기전 EBP에 ESP의 주소를 복사함.
      - EBP를 Push(스택에 삽입)함. --> 호출받은 함수의 EBP가 완성
      - SFP(Stack Frame Pointer)를 스택에 쌓음.(호출된 함수의 시작 위치)
    - 2.Epilog
      - 함수를 모두 실행한 뒤 복귀할 경우 생성한 스택을 해제하는 작업.
      - 해제하기전 ESP에 EBP를 복사함.
      - EBP를 Pop(스택에서 꺼냄) --> 꺼낸 EBP로 이동(이전 함수로 되돌아감)
      - ESP를 이전 함수위치로 되돌아감(이전 함수 Stack의 Top으로 돌아감)
  - 매개변수를 삽입하여 함수를 호출했을경우, 마지막에 들어간 인자값에 대해서 나중에 호출함.
  - Ex) func(int a, int b) --> b가 먼저 쌓이고, a가 나중에 들어감.

Wikipedia

List of x86 calling conventions [edit]

This is a list of x86 calling conventions.[1] These are conventions primarily intended for C/C++ compilers (especially the 64-bit part below), and thus largely special cases. Other languages may use other formats and conventions in their implementations.

Architecture	Calling convention name	Operating system, compiler	Parameters in registers	Parameter order on stack	Stack cleanup by	Notes
8086	cdecl			RTL (C)	Caller	
	Pascal			LTR (Pascal)	Callee	
	fastcall	Microsoft (non-member)	AX, DX, BX	LTR (Pascal)	Callee	Return pointer in BX.
	fastcall	Microsoft (member function)	AX, DX	LTR (Pascal)	Callee	"this" on stack low address. Return pointer in AX.
	fastcall	Turbo C[17]	AX, DX, CX	LTR (Pascal)	Callee	"this" on stack low address. Return pointer on stack high address.
		Watcom	AX, DX, BX, CX	RTL (C)	Callee	Return pointer in SI.
IA-32	cdecl	GCC		RTL (C)	Caller	When returning struct/class, the calling code allocates space and passes a pointer to this space via a hidden parameter on the stack. The called function writes the return value to this address.
	cdecl	Microsoft		RTL (C)	Caller	When returning struct/class, • POD return values 32 bits or smaller are in the EAX register • POD return values 33-64 bits in size are returned via the EAX:EDX registers. • Non-POD return values or values larger than 64-bits, the calling code will allocate space and passes a pointer to this space via a hidden parameter on the stack. The called function writes the return value to this address.
	stdcall	Microsoft		RTL (C)	Callee	
		GCC		RTL (C)	Hybrid	Stack aligned on 16 bytes boundary.
	fastcall	Microsoft	ECX, EDX	RTL (C)	Callee	Return pointer on stack if not member function.
	fastcall	GCC	ECX, EDX	RTL (C)	Callee	
	register	Delphi and Free Pascal	EAX, EDX, ECX	LTR (Pascal)	Callee	
	thiscall	Windows (Microsoft Visual C++)	ECX	RTL (C)	Callee	Default for member functions.
	vectorcall	Windows (Microsoft Visual C++)		RTL (C)		
		Watcom compiler	EAX, EDX, EBX, ECX	RTL (C)	Callee	Return pointer in ESI.
x86-64	Microsoft x64 calling convention [11]	Windows (Microsoft Visual C++, GCC, Intel C++ Compiler, Delphi), UEFI	RCX/XMM0, RDX/XMM1, R8/XMM2, R9/XMM3	RTL (C) [18]	Caller	Stack aligned on 16 bytes. 32 bytes shadow space on stack. The specified 8 registers can only be used for parameters 1 through 4.
	vectorcall	Windows (Microsoft Visual C++)	RCX/XMM0, RDX/XMM1, R8/XMM2, R9/XMM3 + XMM0-XMM5/YMM0-YMM5	RTL (C)	Caller	[19]
	System V AMD64 ABI[16]	Solaris, Linux, BSD, OS X (GCC, Intel C++ Compiler)	RDI, RSI, RDX, RCX, R8, R9, XMM0–7	RTL (C)	Caller	Stack aligned on 16 bytes boundary. 128 bytes red zone below stack.

- **Gcc on x86**
  - **%ebp is the "base pointer" for your stack frame. It's the pointer used by the C runtime to access local variables and parameters on the stack.**

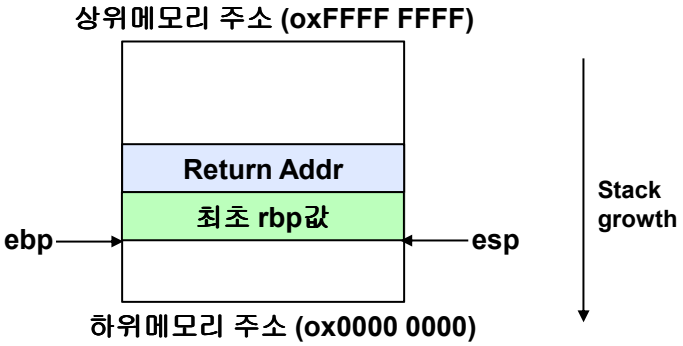
```
// junk.c++
int addtwo(int a)
{
    int x = 2;

    return a + x;
}
```

```
.file   "junk.c++"
    .text
.globl _Z6addtwoi
    .type   _Z6addtwoi, @function
_Z6addtwoi:
.LFB2:
    pushl   %ebp
.LCFI0:
    movl    %esp, %ebp
.LCFI1:
    subl    $16, %esp
.LCFI2:
    movl    $2, -4(%ebp)
    movl    -4(%ebp), %edx
    movl    8(%ebp), %eax
    addl    %edx, %eax
    leave
    ret
.LFE2:
    .size   _Z6addtwoi, .-_Z6addtwoi
    .ident  "GCC: (Ubuntu 4.3.3-5ubuntu4)
4.3.3"
    .section    .note.GNU-stack,"",@progbits
```

1. `pushl %ebp` stores the stack frame of the calling function on the stack.
2. `movl %esp, %ebp` takes the current stack pointer and uses it as the frame for the called function.
3. `subl $16, %esp` leaves room for local variables.
4. `leave` instruction which is an x86 assembler instruction which does the work of restoring the calling function's stack frame.

https://ejrtmtm2.wordpress.com/2013/04/10/%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%A8-%EC%8B%A4%ED%96%89-%EA%B3%BC%EC%A0%95%EC%97%90-%EB%94%B0%EB%A5%B8-%EC%8A%A4%ED%83%9D%EC%9D%98-%EB%8F%99%EC%9E%91-%EC%9D%B4%ED%95%B4%ED%95%98%EA%B8%B0-13%EB%85%84/

```c
main() {

  int i, j, sum ;

  i = 1 ; j = 2 ;
  sum = sub(i,j) ;

}

int sub(int x, int y) {
    int temp ;
    temp = x + y ;
    return (temp) ;
}
```




```
        .file   "activation.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $1, -12(%rbp)
        movl    $2, -8(%rbp)
        movl    -8(%rbp), %edx
        movl    -12(%rbp), %eax
        movl    %edx, %esi
        movl    %eax, %edi
        call    sub
        movl    %eax, -4(%rbp)
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .globl  sub
        .type   sub, @function
sub:
.LFB1:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movl    -24(%rbp), %eax
        movl    -20(%rbp), %edx
        addl    %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   sub, .-sub
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```