Unit II
Procedure Invocation

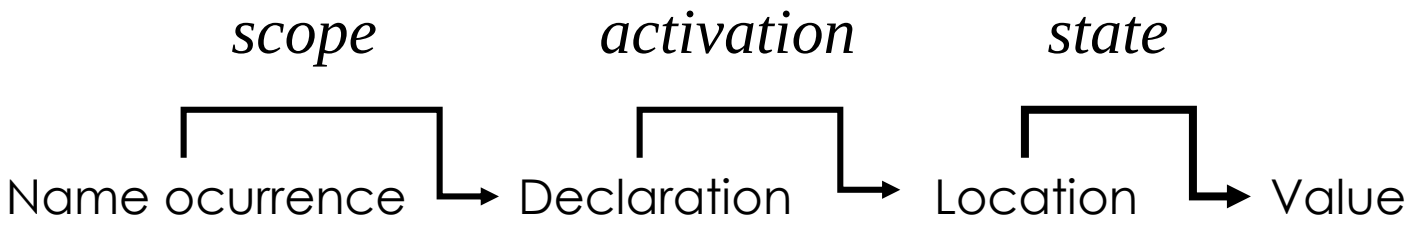
By B Lokesh Joel

UNIT-2:

Title and Content Layout with List

- **Structured Programming :**
  - Need for Structured programming
  - Design considerations
  - Handling special cases in loops,
  - Programming with invariants
  - Control flow in C.
- **Types –**
  - Role of Types
  - Basic Types
  - Arrays
  - Records
  - Unions
  - Sets
  - Pointers
  - Types and Error Checking.
- **Procedure Invocation:**
  - **Introduction to Procedures**
  - **parameter passing methods**
  - **Scope Rules for Names**
  - **Nested Scopes**
  - **Activation Records.**

Exploration



a. **From names to their declarations. The same name x can be used in different senses in different parts of program. When the same name x is used in different senses, which sense or declaration applies to a given occurrence of x?**

b. **From declarations to storage locations. Each time a procedure is executed or activated, the variables in the procedure are bound to locations. Which location does a name in a declaration denote?**

c. **From locations to values. A variable x in an imperative program can denote both a location and the value in that location. Does an occurrence of a variable name x refer to its value or its location?**

Introduction to Procedures

- Procedures
- Procedure Calls
- Elements of a procedure
- Recursion
- Benefits

Introduction to Procedures

- Procedures are constructs for giving a name to a piece of coding(body)
- When the name is called , the body is executed.
- Two Forms
  - **Function Procedures - Functions**
  - **Proper Procedures - Procedures**
- **Functions** Return a single value
- **Procedures** have only a side effect such as setting variables or performing
  - output and returns no value

Procedure Calls

- Use of a Procedure is referred to as a call of Procedure

Actual parameter

- **< Procedure - name > (< parameters>)**

- The parenthesis around parameters are a syntactic cue to a call
- Functions are called from within expressions

  example: **r * sin(angle)**

- Procedures are treated as Atomic statements

  example : **read(ch) ;**

ELEMENTS OF A PROCEDURE

- A name for the declared Procedure
- A body consisting of local declaration and statements
- The formal parameters which are place holders for actuals
- An optional result type

Example Pascal

```
function  square ( x : integer): integer
          begin
                    square :=  x *  x
          end ;
```

Example C Lang

```
int square ( int x)
{
          int sq;
          sq = x * x;
          return  sq;
}
```

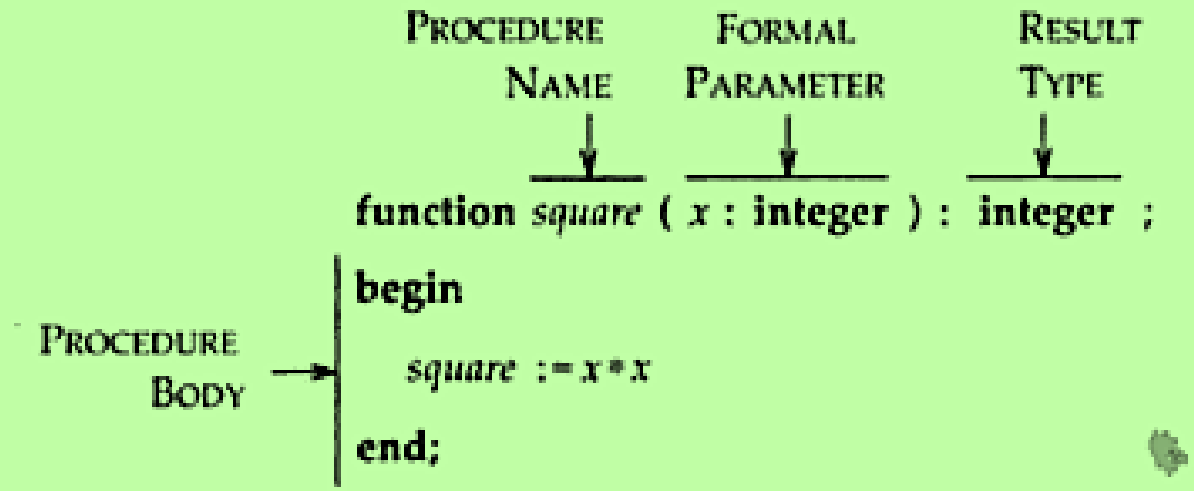


Figure 5.2 The elements of a function procedure *square*, in Pascal.

Example 5.1 A procedure declaration begins with keyword procedure in Pascal:

```
procedure getch;
        begin
            while eoln do readln;
            read (ch)
        end;
```

RECURSION : MULTIPLE ACTIVATION

- **Activation -** Each execution of a procedure body is referred to as an activation of the procedure

- **Recursion -** A procedure is recursive if it can be activated from within its own procedure body

Example- Factorial function

```
function f( n : integer) : integer;
begin
        if  n = 0 then f := 1  else
        f :=  n * f ( n - 1 )
end ;
```

- f(n) is computed in terms of f(n-1), f(n-1) in terms of f(n-2) and so on
- for n = 3 the sequence of activation is a s follows

```
f(3) = 3 * f(2)
        f(2) = 2 * f(1)
                f(1) = 1 * f(0)
                        f(0) = 1
                f(1) = 1
        f(2) = 2
f(3) = 6
```

Benefits of Procedures

1. Procedure abstraction
2. Implementation hiding
3. Modular programs
4. Libraries

Parameter-Passing Methods

- match actuals with formals when a procedure call occurs

```
function square (x: integer): integer
begin
        square := x * x
end;
```

- call *square*(2): match 2 with x

1. **Call-by-value**
   - Pass the value of A[j]
2. **Call-by-Reference**
   - Pass the location of A[j]
3. **Call-by-value-Result**
   - Pass the text A[j] itself, avoiding "Name clashes."

1.Call-by-value

- a formal parameter corresponds to the value of an actual parameter
- the primary parameter-passing method in C and Pascal
- ***swap(a, b)* does nothing to *a* and *b***

```
procedure swap(x, y: T)
var z: T;
begin
   z := x; x:= y; y:= z;
end
```

CALL-by-Value

```
Square(5);
```

```
Function square(x:integer):integer;
Begin
   square := x*x;
End;
```

```
X := 5;
```

Execute the body:

```
Square := x * x := 5 * 5 = 25
```

Return square

2.Call-by-Reference

- a formal parameter becomes a synonym for the location of an actual parameter
- an actual reference parameter must have a location
- the location of an actual parameter is computed and passed-by-reference before the procedure body is executed

```
procedure swap(var x: integer; var y: integer);
var z:integer;
begin
  z := x; x := y; y := z;
end;
```

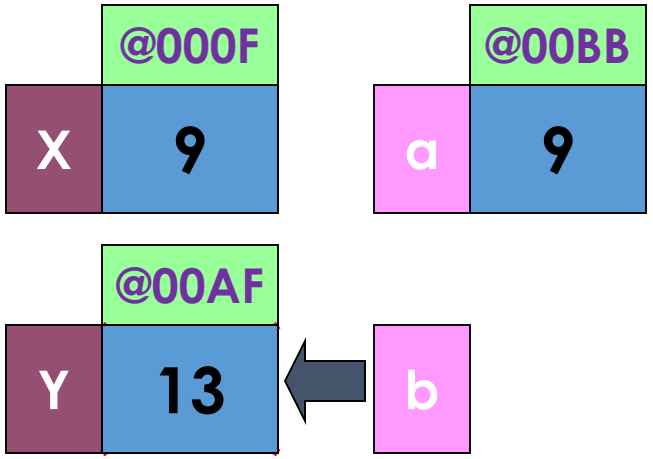
- if *i* =2 and *A*[2] = 99, what are the values of *i* and *A*[*i*] after calling *swap*(*i*, *A*[*i*])?

CALL-by-Reference

- A formal parameter becomes a synonym for the location of an actual parameter.

```
X := 9;
Y := 4;
Add(x,y);
```

```
Procedure add(a:int;var b:int);
Begin
  b := b + a;
End;
```




```
A := X = 9;
B is locate as location of y
B := 9 + 4 = 13;
Then
B&Y := 13;
```

Cont..

- **In C, the effect of call-by-reference can be achieved using pointers**

```
void swapc(int *px, int *py) {
   int z;
   z = *px; *px = *py; *py = z;
}
```

- A prefix '*' is the pointer dereferencing operator

3.Call-by-value-Result

- also known as copy-in/copy-out
- the actuals are initially copied into the formals
- the formals are eventually copied back out to the actuals
- **Copy-in phase**
  - Both values and locations for actual parameters are computed.
  - Values are assigned to corresponding formals; locations are saved for Copy-out phase
- **Copy-out phase**
  - After procedure body is executed, the final value of formals are copied back out to the locations computed in Copy-in phase

```
program
...
procedure foo(x,y); begin i := y end;
...
begin
  i:= 2; j:=3;
foo(i,j);
end
```

- The call *foo*(*i, j*) leaves both *i* and *j* unchanged because that copy-out phase restores their values
- what is the value of *i* if call-by-reference is used instead?

In C Language

- In C has only call-by-value parameter-passing method.

```
void swapc(int *px,int *py) {
   int z;
   z   = *px;
   *px = *py;
   *py = z;
}
```

```
swapc(&a,&b);
```

```
px = &a;    {pass the address of a to px}
py = &b;     {pass the address of b to py}
z   = *px; {assign z the initial value of a}
*px = *py; {assign a the value of b}
*py = z;    {assign b in z}
```

Scope rules for names

- Lexical or Static Scope Rules
- Dynamic Scope Rules
- Renaming of local variables
- Macro Expansion
- Parameter Passing: Textual Substitution
- Call-by-Name

Scope rules for names

- Determines which declaration of a *name* x applies to an *occurrence of* x in a Program
- Names are able to denote anything in PLs such as procedures, types, constants and variables
- **Two kinds**:
  1. **lexical or Static Scope Rules**
     - name occurrence must be bound to a declaration in terms of the source text at compile time
  2. **dynamic scope rules**
     - name occurrence is bound to a declaration at run time

Lexical and Dynamic Scope

```
program L;
var n : char;    { n declared in L }
  procedure W;
  begin
    writeln(n); { occurrence of n in W }
  end;
  procedure D;
  var n : char; { n redeclared in D }
  begin
    n := 'D';    { W called within D }
    W;
  end;
Begin {program L}
  n := 'L';
  W;  { W called from the main program L}
  D;
End.
```

Lexical

L
L

Dynamic

L
D

Lexical Scope and the Renaming of Locals

- Consider procedure D; rename *n* to *r*.

```
procedure D;
var r:char;
begin
  r:= 'D';   W;
end;
```

- Only one declaration of n in program, then every time procedure W is called, it write out the value n in L.

Renaming of local variables. **Consistent renaming of local names in the source text has no effect on the computation set up by a program.**

Macro Expansion and Dynamic Scope

- If a procedure body is simply copied or substituted at the point of call, we get dynamic scope. A macro processor does:
  - Actual parameters are textually substituted for the formals.
  - The resulting procedure body is textually substituted for the call.

Textual substitution

- Example : C used a macro preprocessor

```
#define MAXBUF 4
```

Every occurrence of MAXBUF is replaced by 4 before the program is compiled.

Macro expanding

- Naming Conflicts

```
procedure W;
begin  writeln(n); end;
```

```
procedure D;
var n:char;
begin
  n := 'D';
  W;
end;
```

(a) Call of W

```
procedure D;
var n:char;
begin
  n := 'D';
  begin writeln(n); end;
end;
```

(b) After macro expansion

- Parameter Passing

Nested Scope

- Binding occurrence : x,y
- Bound occurrence : z

```
procedure swap(var x,y:T);
var z : T;
begin
   z:=x; x:=y; y:=z;
end;
```

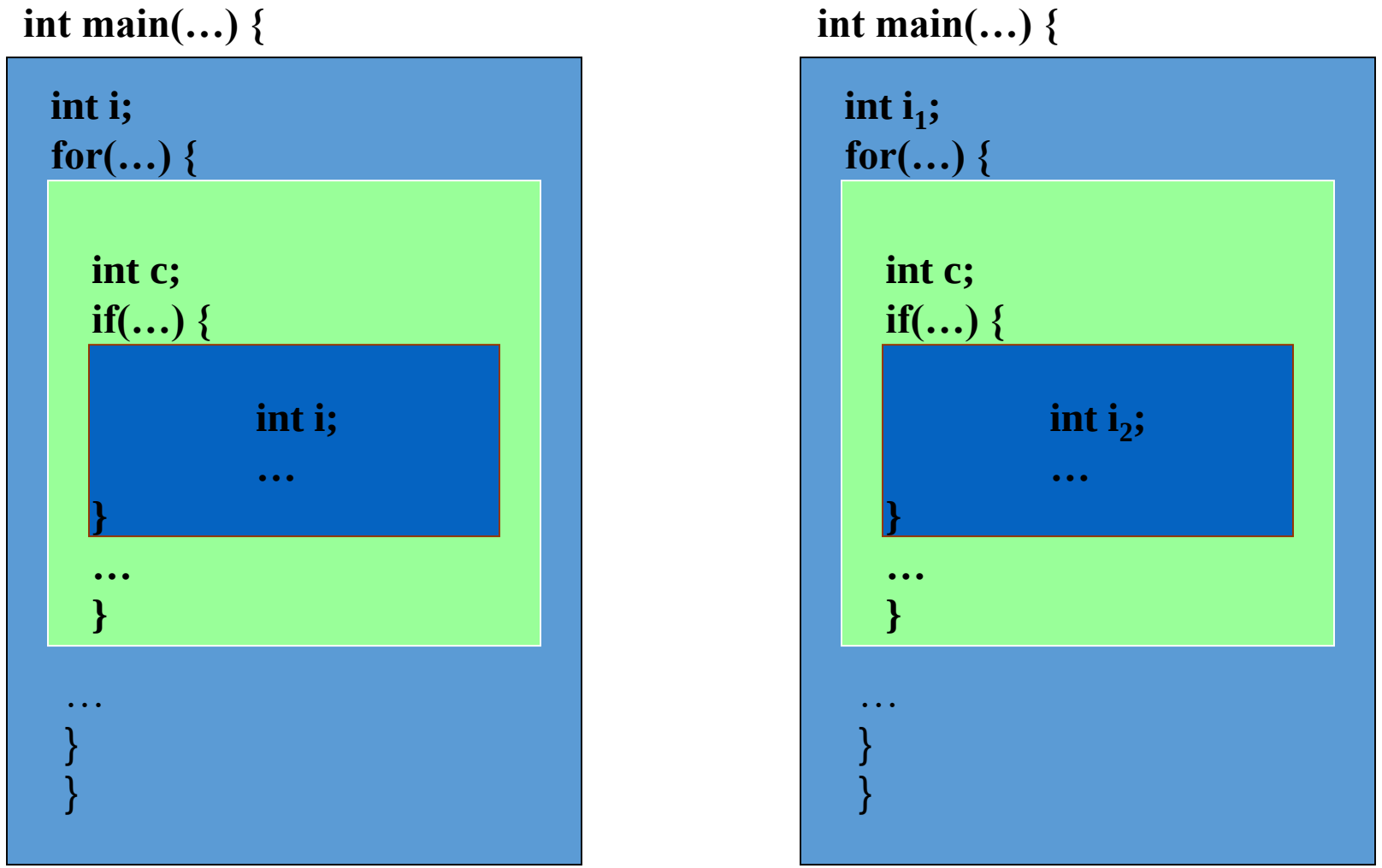
Nested Scopes : Variable Declarations in C

- Variable declarations can appear within any grouping of statements in C. Compound statements are grouped within braces { and }

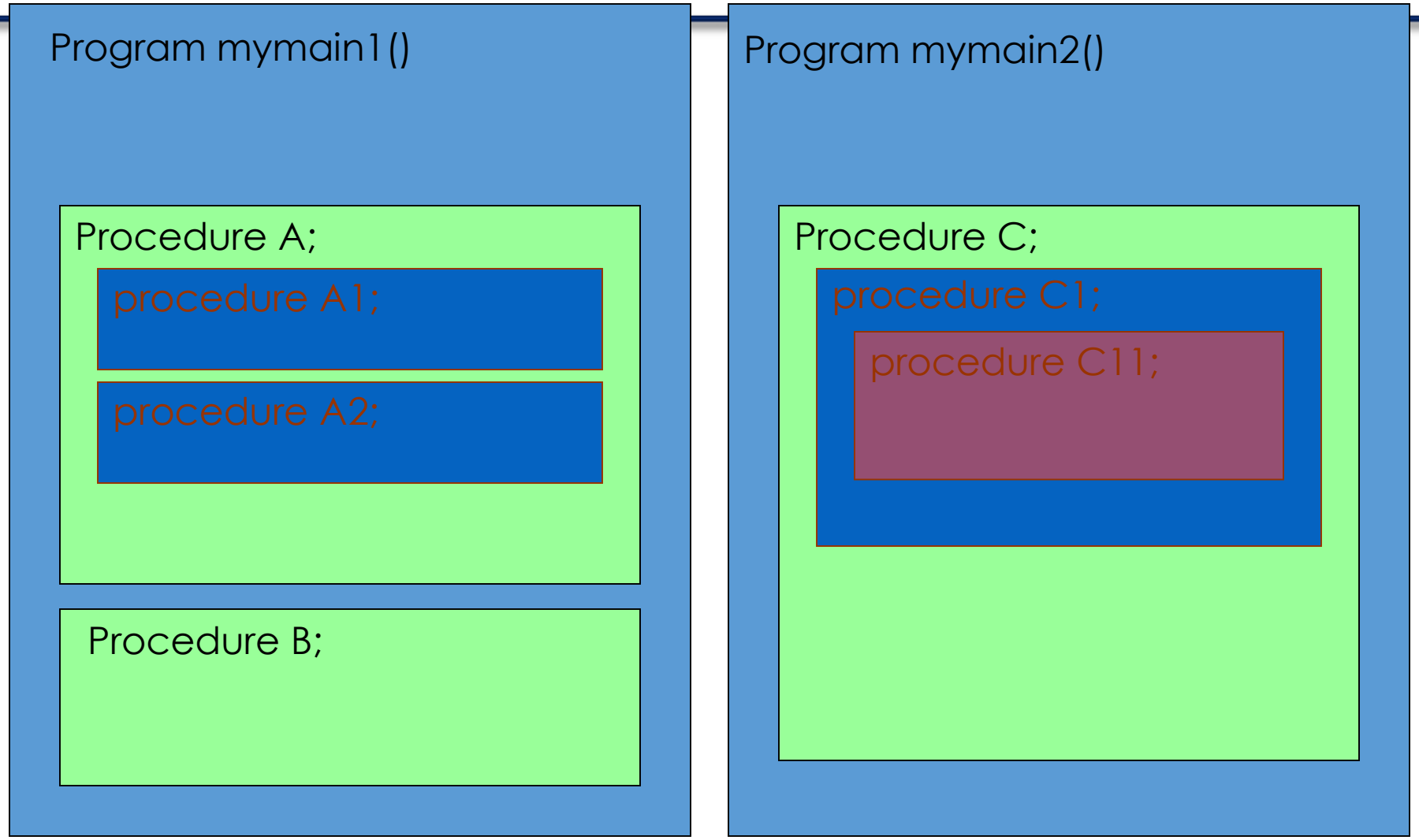
{ <declaration-list> <statement-list>}

```
{
int I=0;
While (I<=limit )
    {…
    I++;
    }
}
```

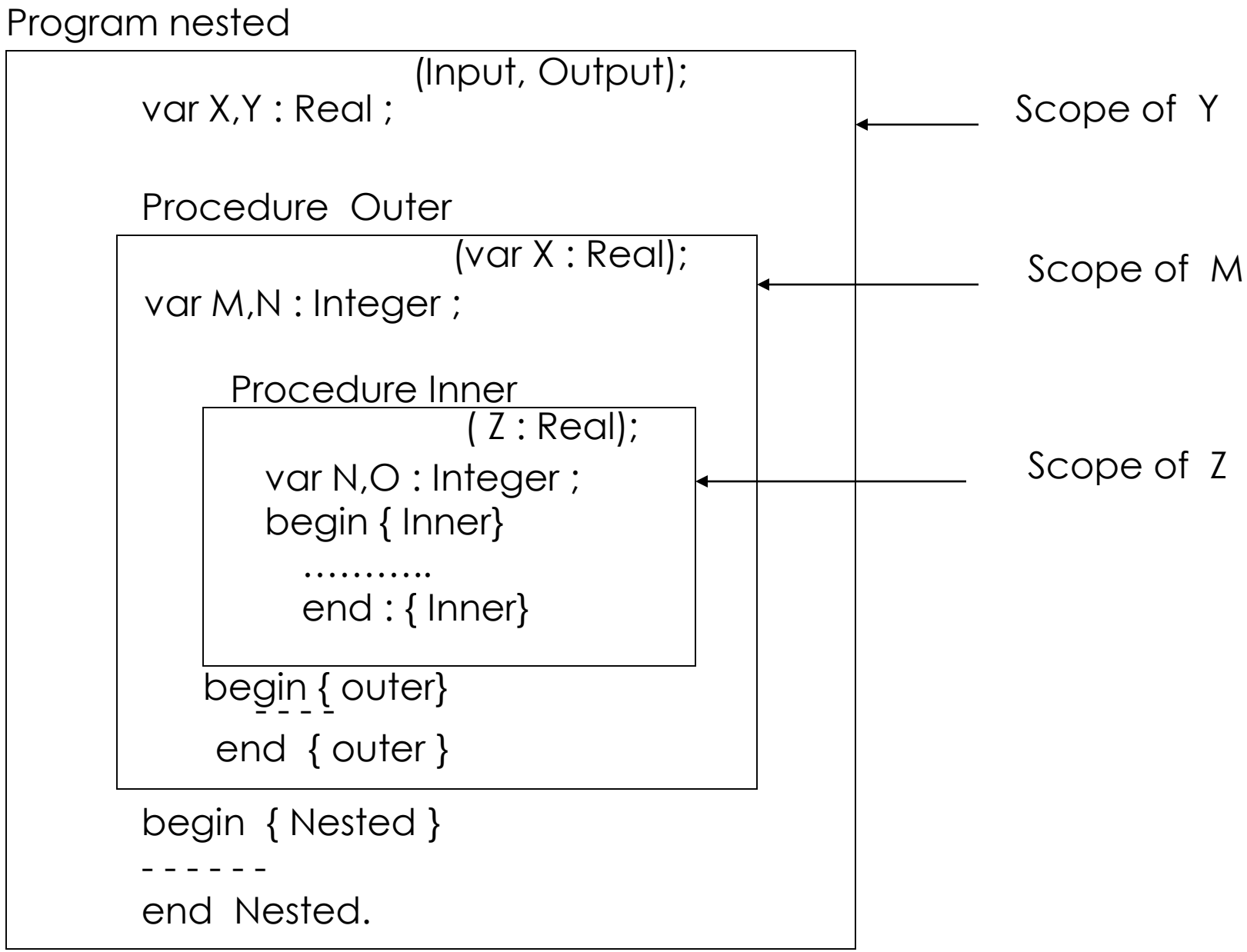
Nested compound statements



Procedure declaration



NESTED SCOPES- PROCEDURE DECLARATION IN PASCAL



Activition Records

- Activation of a procedure is storage for the variables declared in the procedure.
- The storage associated with a activation is called an activation record.

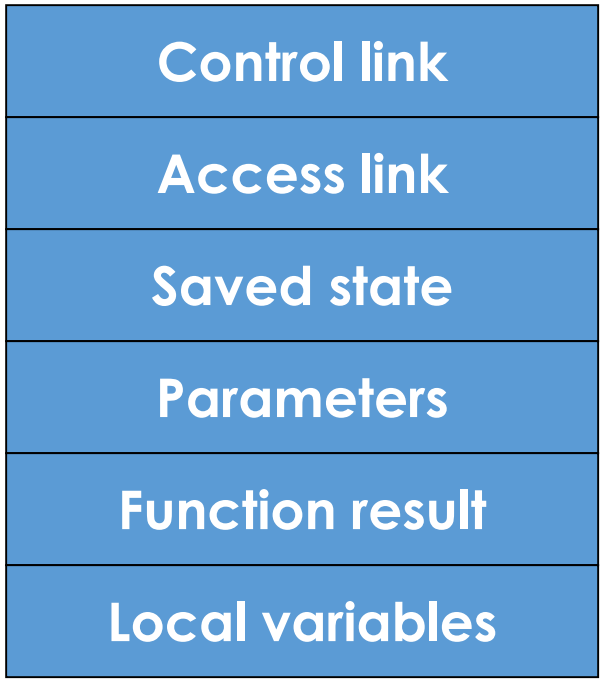
Activation binding

- In a lexically scoped language, the 3 mappings or bindings are done at three different times:

1. **Compile time.** The binding of name occurrences to declarations is defined in terms of the source.
2. **Activation time** . The binding of declarations to locations in done at activation time
3. **Run time.** The binding of locations to values is done dynamically at run time

Elements of an Activation Record [frames]

- Data needed for an activation of a procedure is collected in a record called an activation record or frame.
- The record contains



Cont..

- In a language with recursive procedures, each activation has its own activation record
- declaration in a procedure result in storage being allocated within the activation records for the procedure
- ***Control link***, also called ***dynamic link***, points to the activation record of the runtime caller
- ***Access link,*** also called a ***static link***, is used to implement lexically scoped language

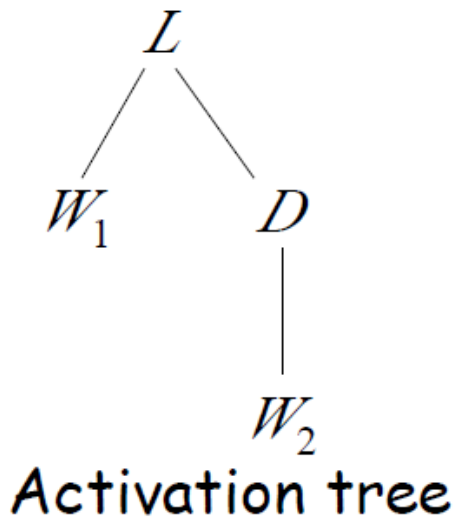
Under dynamic scope

- the storage for some nonlocal variable can be found by following control links

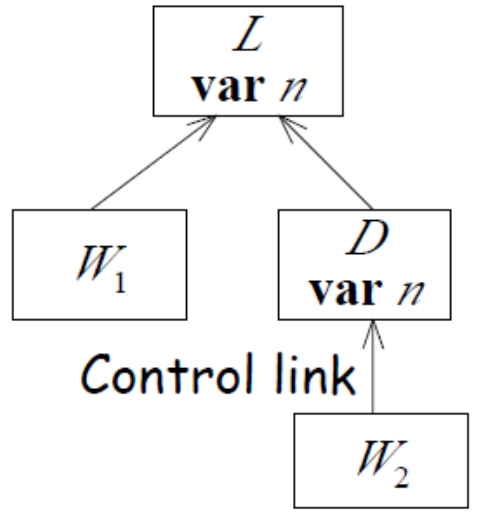
```
program L
  var n: char;
begin
  n:= 'L';
  W;
  D
end

procedure W
  writeln(n);

procedure D
  var n: char
begin
  n := 'D';
  W;
end;
```



Activation tree



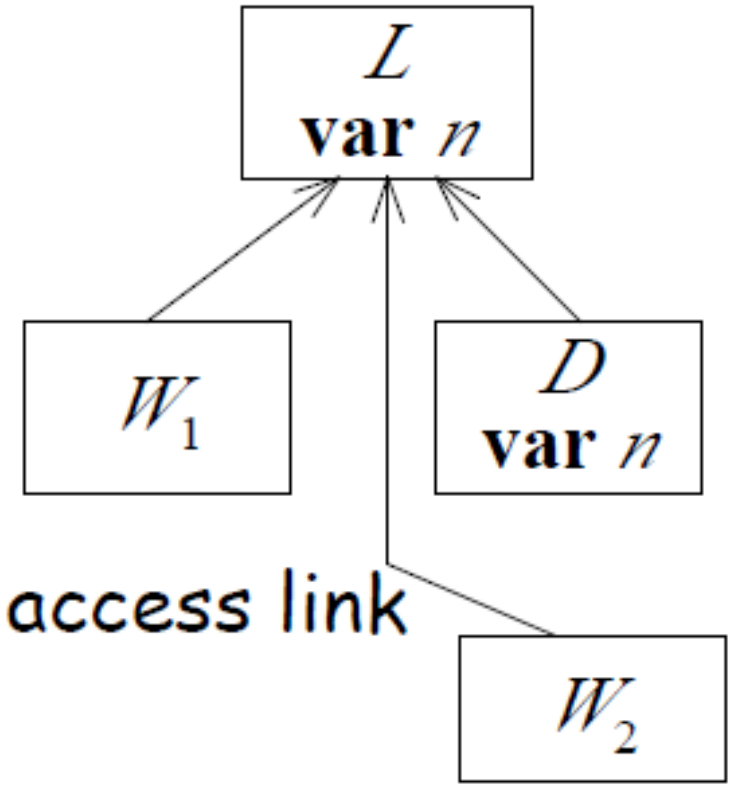
Under lexical scope

- the access link points from an activation record of *W* to that of *L*

```
program L
  var n: char;
begin
  n:= 'L';
  W;
  D
end

procedure W
  writeln(n);

procedure D
  var n: char
begin
  n := 'D';
  W;
end;
```

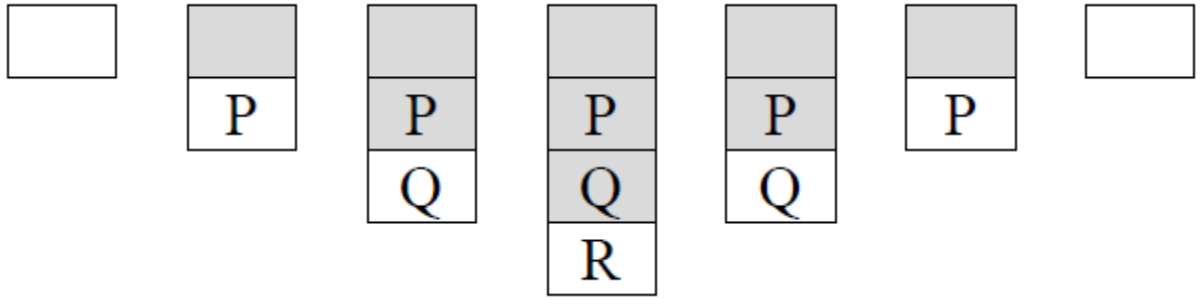


Heap Allocation and Deallocation

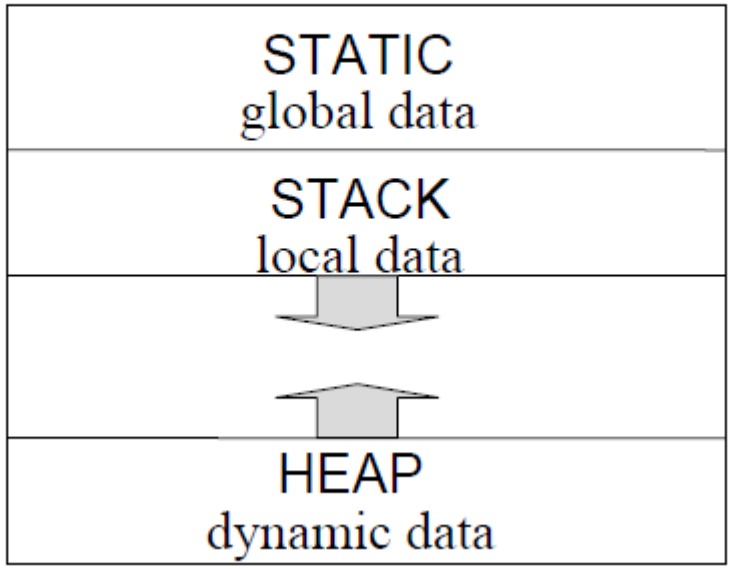
- A general technique for managing activation record is to allocate storage for them in an area called the *heap*
- use *garbage collection* to automatically reclaim storage that is no longer needed
- the lifetimes of activation records need not be tied to last-in/first-out flow of control between activation

Stack Allocation and Deallocation

- Pascal and C use stack allocation for activations



- Memory layout for C program



Allocating Static Variables

- *static variables* within a procedure retain their values between activations
- storage for it is allocated at compile time

Unit III
Object-Oriented Programming

By B Lokesh Joel