

# Runtime Environment

- Relationship between names and data objects (of target machine)
- Allocation & de-allocation is managed by run time support package
- Each execution of a procedure is an activation of the procedure. If procedure is recursive, several activations may be alive at the same time.
  - If a and b are activations of two procedures then their lifetime is either non overlapping or nested
  - A procedure is recursive if an activation can begin before an earlier activation of the same procedure has ended

# Procedure

- A procedure definition is a declaration that associates an identifier with a statement (procedure body)
- When a procedure name appears in an executable statement, it is called at that point
- Formal parameters are the one that appear in declaration. Actual Parameters are the one that appear in when a procedure is called

# Activation tree

- Control flows sequentially
- Execution of a procedure starts at the beginning of body
- It returns control to place where procedure was called from
- A tree can be used, called an activation tree, to depict the way control enters and leaves activations
  - The root represents the activation of main program
  - Each node represents an activation of procedure
  - The node **a** is parent of **b** if control flows from **a** to **b**
  - The node **a** is to the left of node **b** if lifetime of **a** occurs before **b**

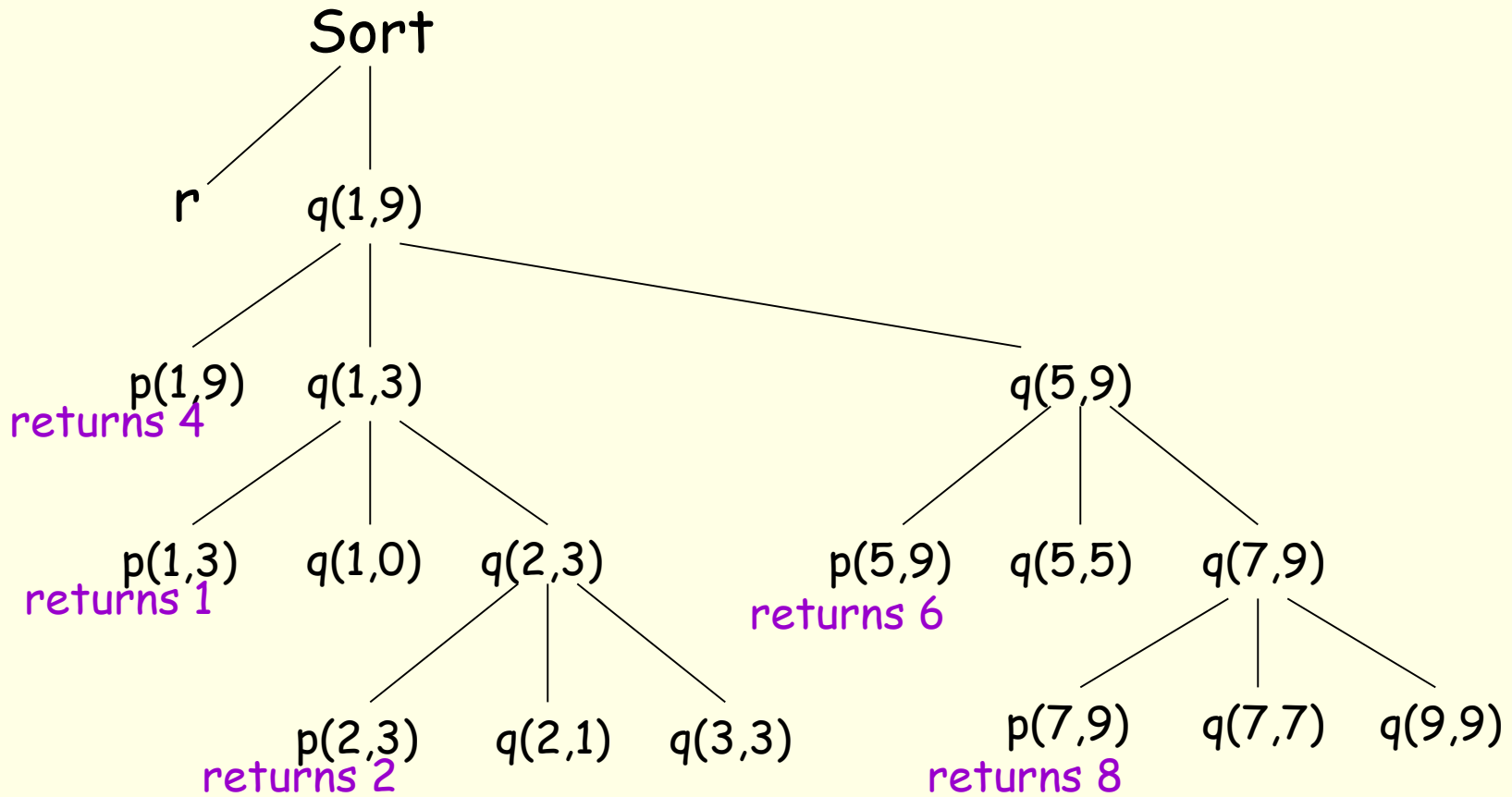
# Example

```
program sort;
  var a : array[0..10] of
    integer;

  procedure readarray;
    var i :integer;
        :
  function partition (y, z
                    :integer)
:integer;
    var i, j ,x, v :integer;
        :
```

```
  procedure quicksort (m, n
                    :integer);
    var i :integer;
        :
    i:= partition (m,n);
    quicksort (m,i-1);
    quicksort(i+1, n);
        :
begin{main}
  readarray;
  quicksort(1,9)
end.
```

# Activation Tree



# Control stack

- Flow of control in program corresponds to depth first traversal of activation tree
- Use a stack called control stack to keep track of live procedure activations
- Push the node when activation begins and pop the node when activation ends
- When the node  $n$  is at the top of the stack the stack contains the nodes along the path from  $n$  to the root

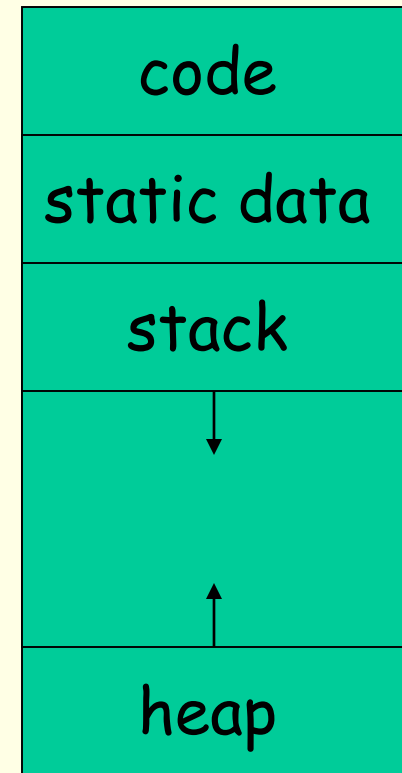
# Scope of declaration

- A declaration is a syntactic construct associating information with a name
  - Explicit declaration :Pascal (Algol class of languages)  
var i : integer
  - Implicit declaration: Fortran  
i is assumed to be integer
- There may be independent declarations of same name in a program.
- Scope rules determine which declaration applies to a name
- Name binding

name  $\xrightarrow{\text{environment}}$  storage  $\xrightarrow{\text{state}}$  value

# Storage organization

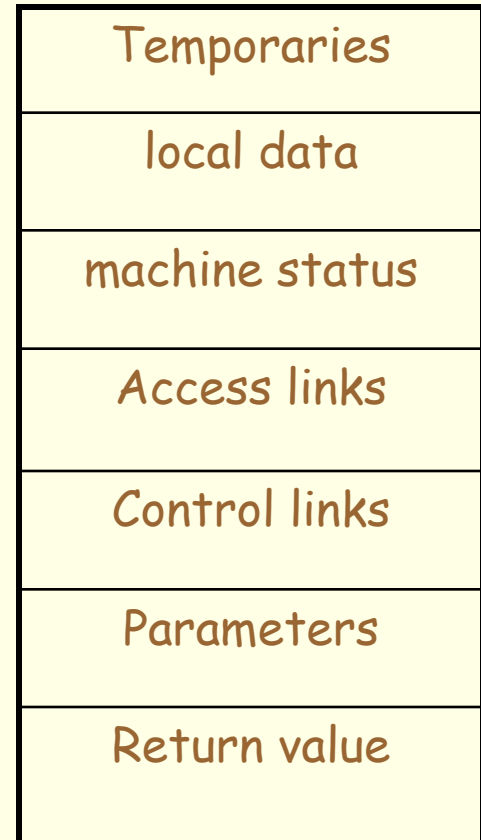
- The runtime storage might be subdivided into
  - Target code
  - Data objects
  - Stack to keep track of procedure activation
  - Heap to keep all other information





# Activation Record

- **temporaries:** used in expression evaluation
- **local data:** field for local data
- **saved machine status:** holds info about machine status before procedure call
- **access link :** to access non local data
- **control link :** points to activation record of caller
- **actual parameters:** field to hold actual parameters
- **returned value:** field for holding value to be returned



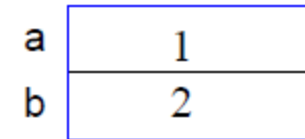
# Activation Records: Examples

- Examples on the next few slides by Prof Amitabha Sanyal, IIT Bombay
- C/C++ programs with gcc extensions
- Compiled on x86\_64

# Example 1 – Vanilla Program in C

```
int a=1,b=2;
main ()
{
    a = a + b;
}
```

Global  
Memory



a and b have both been  
given absolute addresses

compilation of `a = a + b`

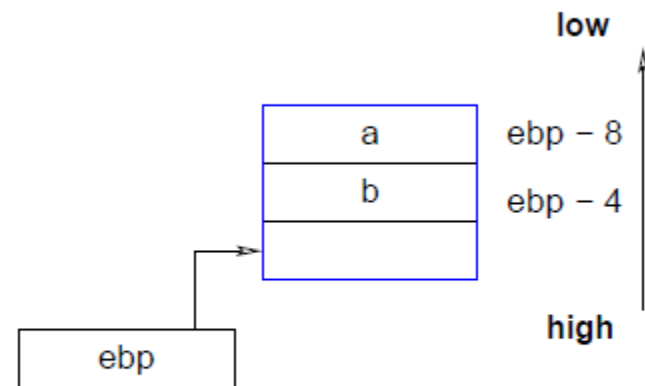
```
...
movl a, %edx
movl b, %eax
addl %edx, %eax
movl %eax, a
...
```

# Example 2 – Function with Local Variables

```
void f()
{
    int a, b;
    a = a + b;
}
```

... compilation of  $a = a + b$

```
...
movl -4(%ebp), %eax
addl %eax, -8(%ebp)
...
```



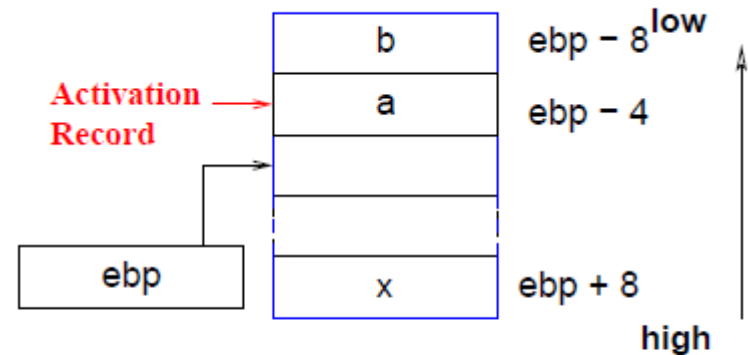
a and b have been given relative address on stack

# Example 3 – Function with Parameters

```
void f(int x)
{
    int a, b;
    a = x + b;
}
...
```

## Compilation of $a = x + b$

```
...
movl -8(%ebp), %eax
movl 8(%ebp), %edx
addl %edx, %eax
movl %eax, -4(%ebp)
...
```



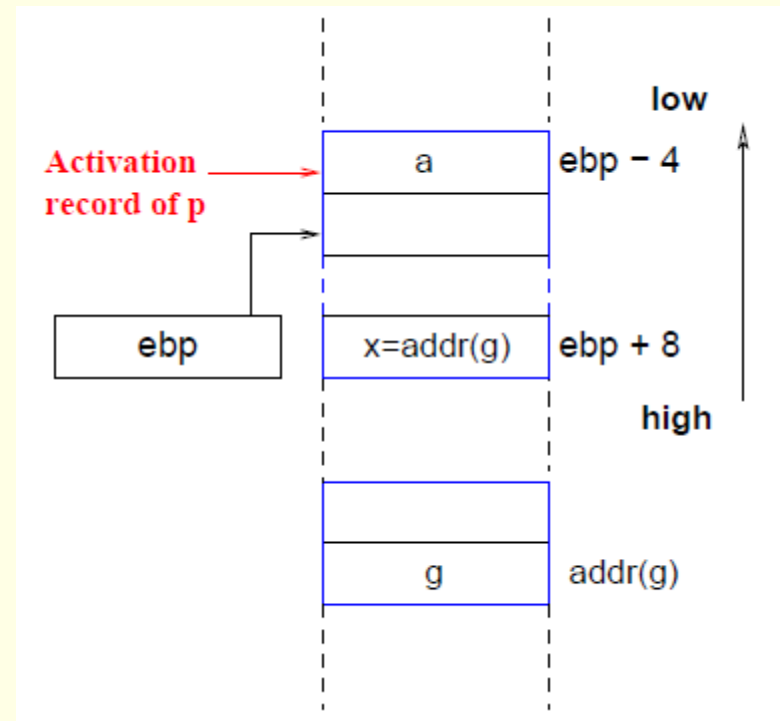
Relative to fp:  
a and b – negative address  
x – positive address

# Example 4 – Reference Parameters

```
int g;
void p(int& x)
{
    int a;
    a = x + 1;
}
main()
{
    p(g);
}
```

... compilation of  $a := x + 1$

```
movl 8(%ebp), %eax
movl (%eax), %eax
addl $1, %eax
movl %eax, -4(%ebp)
```

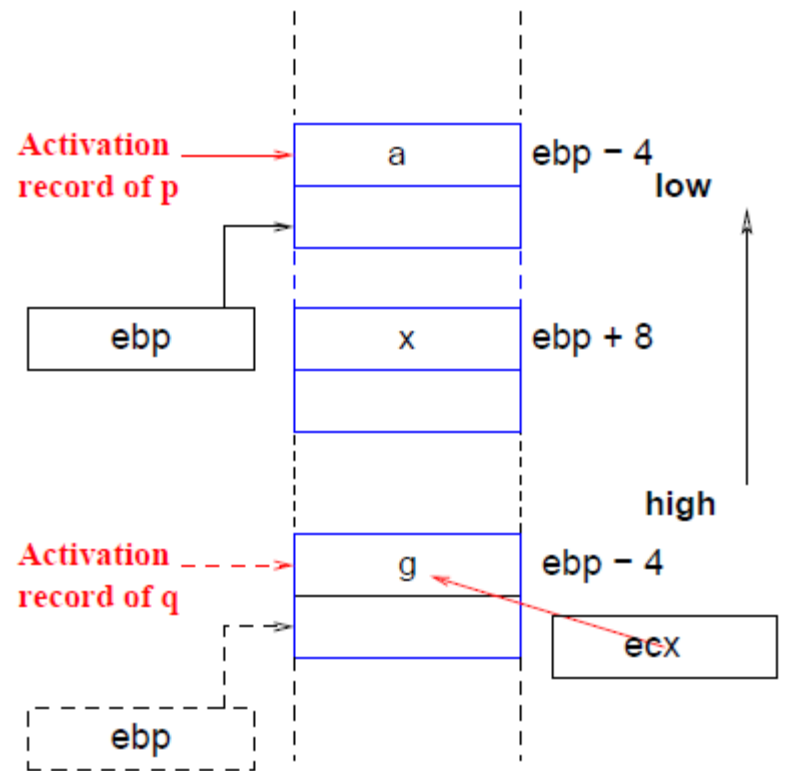


# Example 5 – Global Variables

```
void q()  
{  
    int g;  
    void p(int x)  
    {  
        int a;  
        a = x + g;  
    };  
    p(1);  
};
```

## Compilation of $a = x + g$

```
...  
movl %ecx, %eax ;static link  
movl (%eax), %edx  
movl 8(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)
```



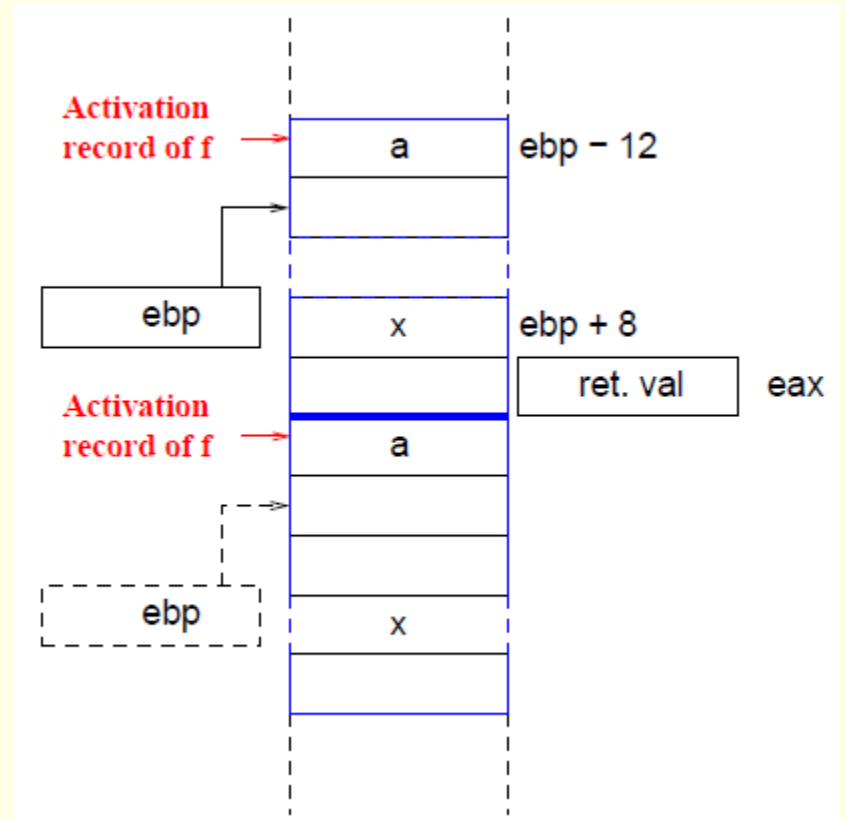
# Example 6 – Recursive Functions

```
int f(int x)
{
    int a;
    if (x==0) return 1;
    {
        a = f(x-1);
        return(x * a);
    }
}
```

... compilation of `a = f(x-1);`  
`return(x * a)`

...

```
movl %eax, -12(%ebp)
movl 8(%ebp), %eax
imull -12(%ebp), %eax
```



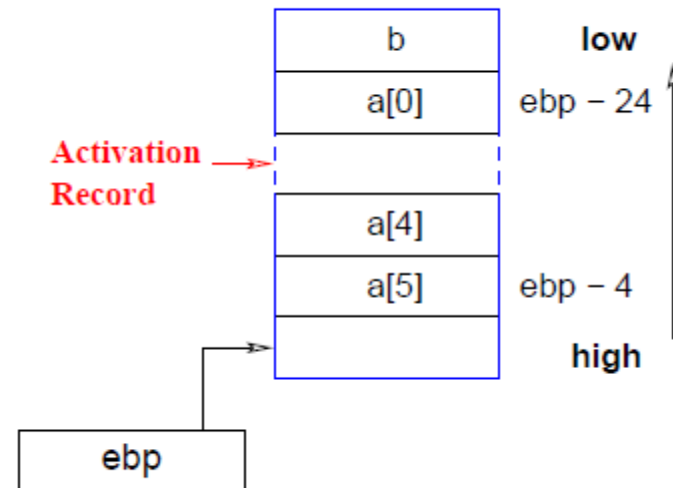


# Example 7 – Array Access

```
void p()
{
    int a[6], b;
    b = a[5];
}
```

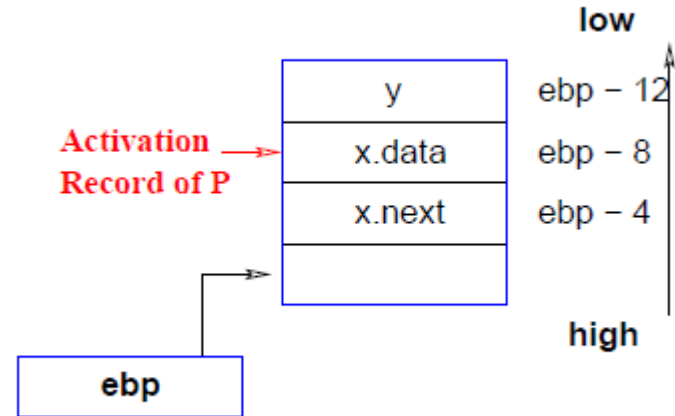
... compilation of `b = a[5]`

```
...
movl -4(%ebp), %eax
movl %eax, -28(%ebp)
...
```



# Example 8 – Records and Pointers

```
typedef struct rec
{
    int data;
    struct rec* next;
} rec;
void p ()
{
    rec x; rec *y;
    x.next = y;
}
```



```
... compilation of x.data = 5;
                    x.next = y;
```

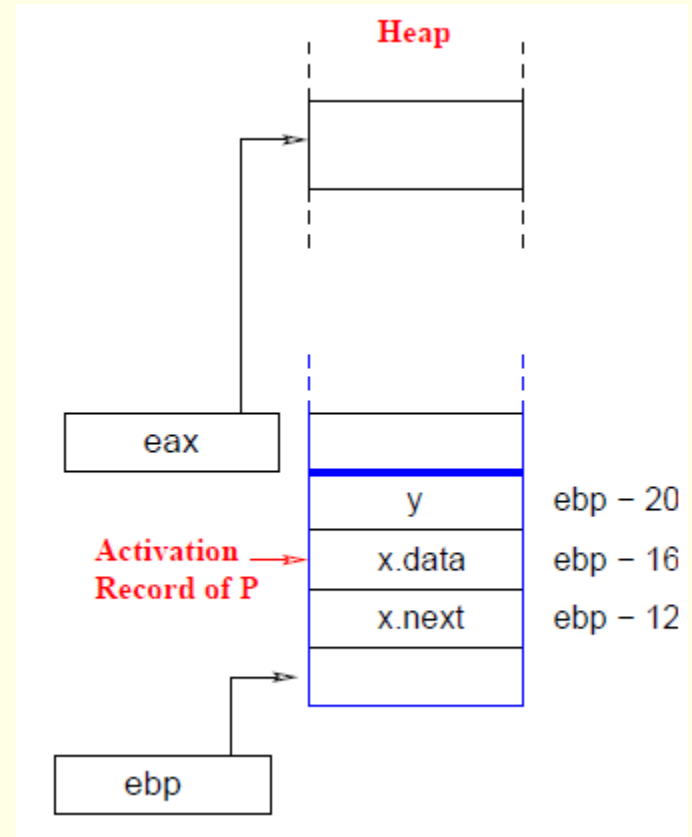
```
...
movl -12(%ebp), %eax
movl %eax, -4(%ebp)
...
```

# Example 9 – Dynamically Created Data

```
typedef struct rec
{
    int data;
    struct rec* next;
} rec;
void p ()
{
    rec x; rec *y;
    y = malloc(4); x.next = y;
}
```

## Compilation of `y = malloc...; x.next = y;`

```
call malloc
movl %eax, -20(%ebp)
movl -20(%ebp), %eax
movl %eax, -12(%ebp)
```



# Issues to be addressed

- Can procedures be recursive?
- What happens to locals when procedures return from an activation?
- Can procedure refer to non local names?
- How to pass parameters?
- Can procedure be parameter?
- Can procedure be returned?
- Can storage be dynamically allocated?
- Can storage be de-allocated?

# Layout of local data

- Assume byte is the smallest unit
- Multi-byte objects are stored in consecutive bytes and given address of first byte
- The amount of storage needed is determined by its type
- Memory allocation is done as the declarations are processed
  - Keep a count of memory locations allocated for previous declarations
  - From the count *relative* address of the storage for a local can be determined
  - As an *offset* from some fixed position

# Layout of local data

- Data may have to be aligned (in a word) padding is done to have alignment.
- When space is important
  - Compiler may pack the data so no padding is left
  - Additional instructions may be required to execute packed data
  - Tradeoff between space and execution time

# Storage Allocation Strategies

- Static allocation: lays out storage at compile time for all data objects
- Stack allocation: manages the runtime storage as a stack
- Heap allocation :allocates and de-allocates storage as needed at runtime from heap

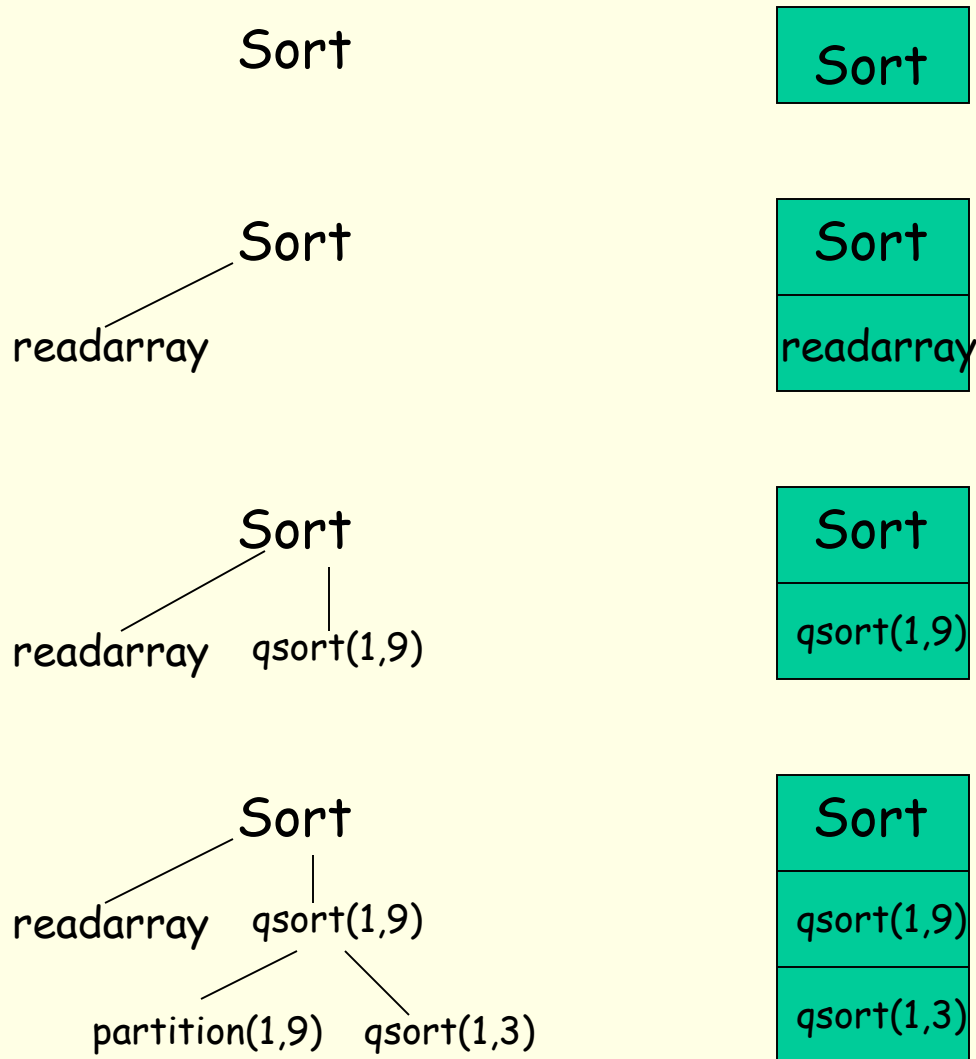
# Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time
- On every invocation of procedure names are bound to the same storage
- Values of local names are retained across activations of a procedure



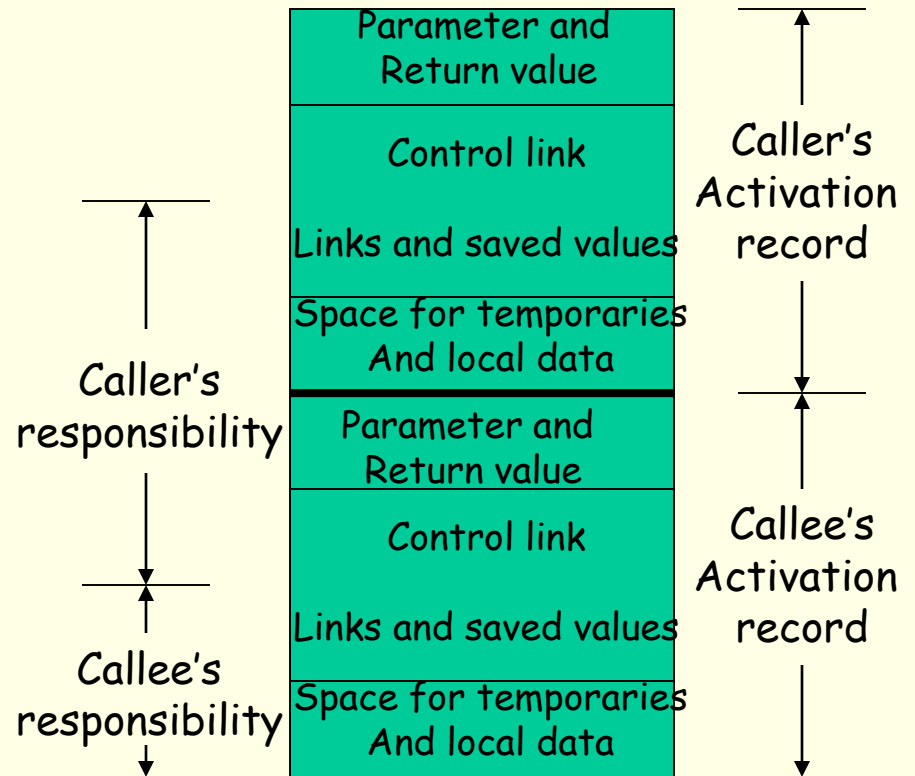
- Type of a name determines the amount of storage to be set aside
- Address of a storage consists of an offset from the end of an activation record
- Compiler decides location of each activation
- All the addresses can be filled at compile time
- Constraints
  - Size of all data objects must be known at compile time
  - Recursive procedures are not allowed
  - Data structures cannot be created dynamically

# Stack Allocation



# Calling Sequence

- A call sequence allocates an activation record and enters information into its field
- A return sequence restores the state of the machine so that calling procedure can continue execution



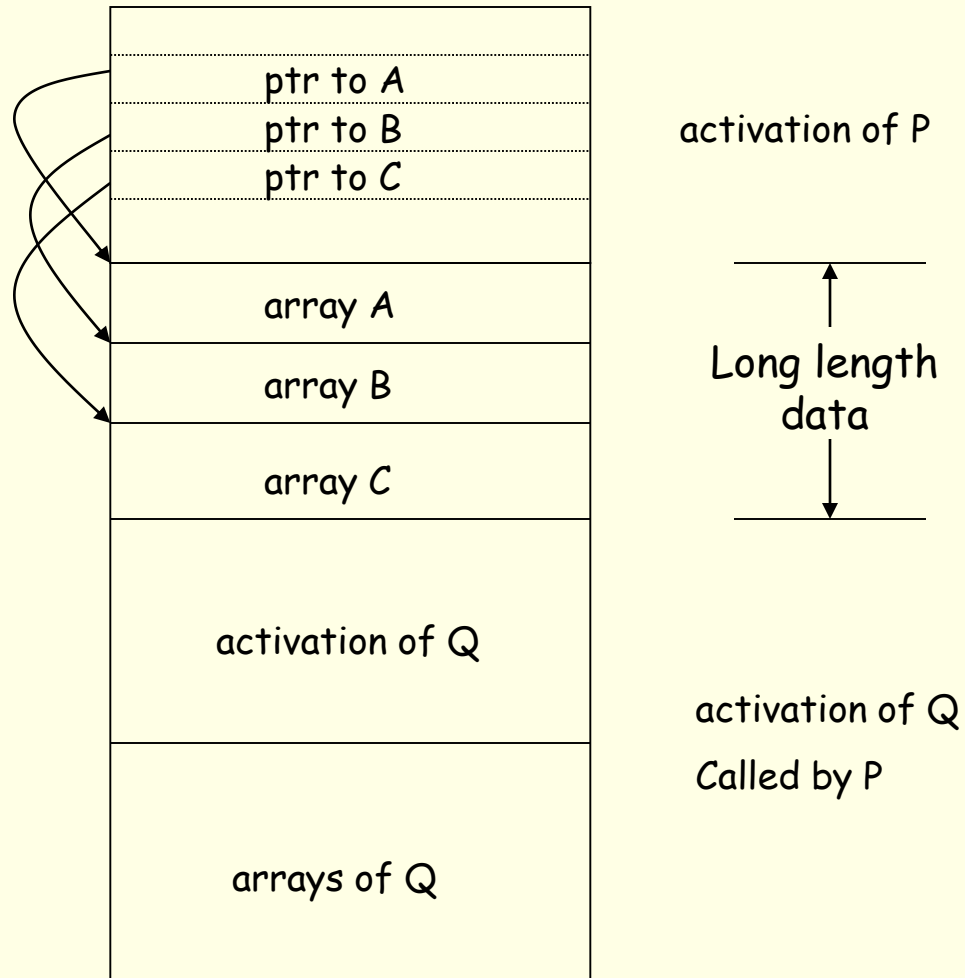
# Call Sequence

- Caller evaluates the actual parameters
- Caller stores return address and other values (control link) into callee's activation record
- Callee saves register values and other status information
- Callee initializes its local data and begins execution

# Return Sequence

- Callee places a return value next to activation record of caller
- Restores registers using information in status field
- Branch to return address
- Caller copies return value into its own activation record

# Long/Unknown Length Data



# Dangling references

Referring to locations which have been deallocated

```
main() {  
    int *p;  
    p = dangle(); /* dangling reference */  
}
```

```
int *dangle() {  
    int i=23;  
    return &i;  
}
```

# Heap Allocation

- Stack allocation cannot be used if:
  - The values of the local variables must be retained when an activation ends
  - A called activation outlives the caller
- In such a case de-allocation of activation record cannot occur in last-in first-out fashion
- Heap allocation gives out pieces of contiguous storage for activation records



# Heap Allocation ...

- Pieces may be de-allocated in any order
- Over time the heap will consist of alternate areas that are free and in use
- Heap manager is supposed to make use of the free space
- For efficiency reasons it may be helpful to handle small activations as a special case

# Heap Allocation ...

- For each size of interest keep a linked list of free blocks of that size
- Fill a request of size  $s$  with block of size  $s'$  where  $s'$  is the smallest size greater than or equal to  $s$ .
- When the block is deallocated, return it to the corresponding list

# Heap Allocation ...

- For large blocks of storage use heap manager
- For large amount of storage computation may take some time to use up memory
  - time taken by the manager may be negligible compared to the computation time

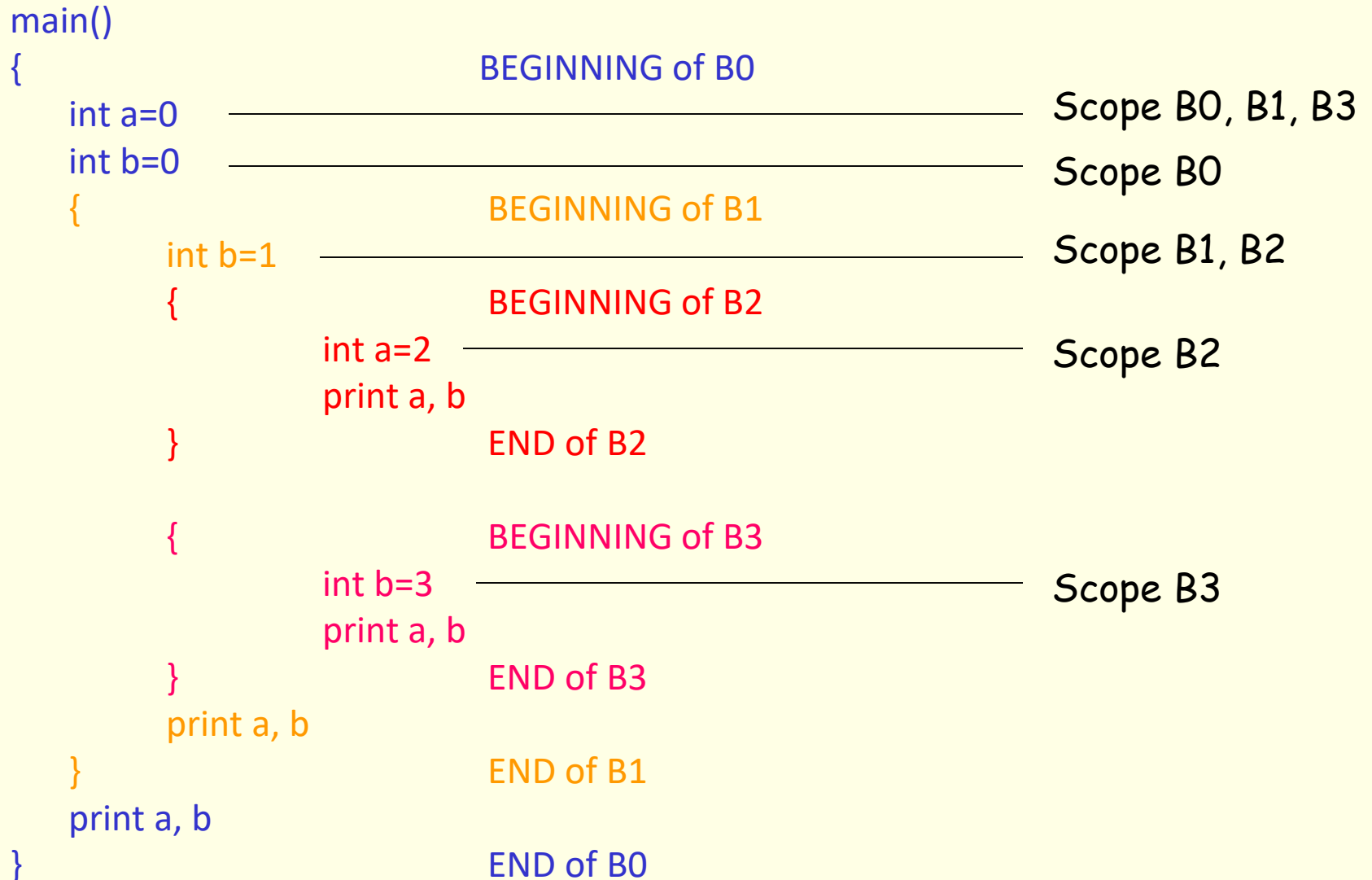
# Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is *lexical scoping* or *static scoping* (most languages use lexical scoping)
  - Most closely nested declaration
- Alternative is *dynamic scoping*
  - Most closely nested activation

# Block

- Statement containing its own data declarations
- Blocks can be nested
  - also referred to as *block structured*
- Scope of the declaration is given by *most closely nested* rule
  - The scope of a declaration in block B includes B
  - If X is not declared in B then an occurrence of X in B is in the scope of declaration of X in B' such that
    - B' has a declaration of X
    - B' is most closely nested around B

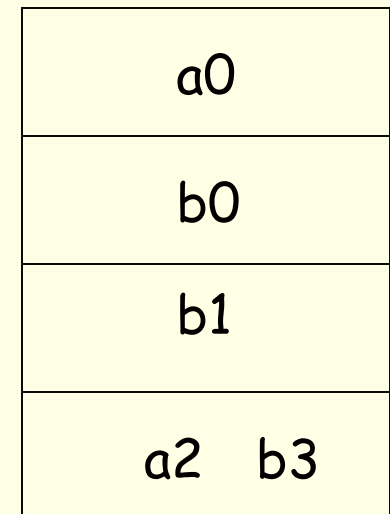
# Example



# Blocks ...

- Blocks are simpler to handle than procedures
- Blocks can be treated as parameter less procedures
- Either use stack for memory allocation
- OR allocate space for complete procedure body at one time

```
{ // a0
  { // b0
    { // b1
      { // a2
      }
      { //b3
      }
    }
  }
}
```



## Lexical scope without nested procedures

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated at compile time
- Any name non-local to one procedure is non-local to all procedures
- In absence of nested procedures use stack allocation
  - Storage for non locals is allocated statically
    - Any other name must be local to the top of the stack
- Static allocation of non local has advantage:
  - Procedures can be passed/returned as parameters



# Scope with nested procedures

```
Program sort;  
  var a: array[1..n] of integer;  
      x: integer;  
  procedure readarray;  
    var i: integer;  
  begin  
  
  end;  
  procedure exchange(i,j:integer)  
  begin  
  
  end;
```

```
procedure quicksort(m,n:integer);  
  var k,v : integer;  
  
  function partition(y,z:integer): integer;  
    var i,j: integer;  
  begin  
  
  end;  
  begin  
  
  end;  
begin  
  
end.
```

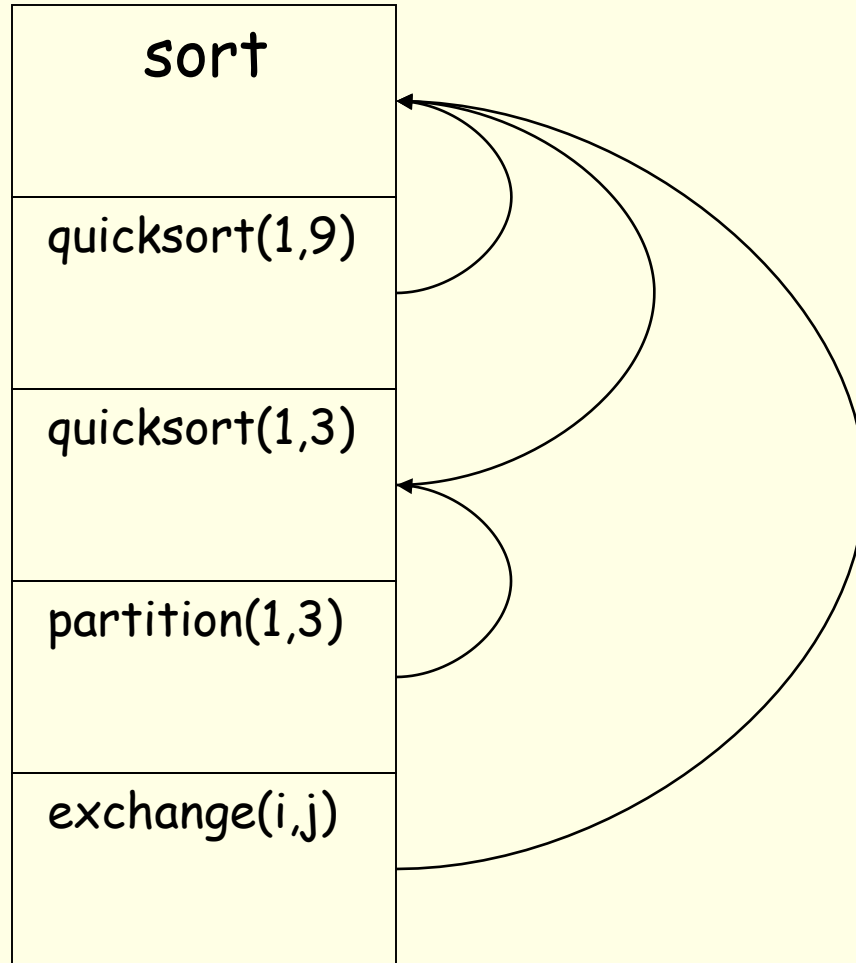
# Nesting Depth

- Main procedure is at depth 1
- Add 1 to depth as we go from enclosing to enclosed procedure

## Access to non-local names

- Include a field 'access link' in the activation record
- If  $p$  is nested in  $q$  then access link of  $p$  points to the access link in most recent activation of  $q$

↓  
Stack



# Access to non local names ...

- Suppose procedure  $p$  at depth  $n_p$  refers to a non-local  $a$  at depth  $n_a$  ( $n_a \leq n_p$ ), then storage for  $a$  can be found as
  - follow  $(n_p - n_a)$  access links from the record at the top of the stack
  - after following  $(n_p - n_a)$  links we reach procedure for which  $a$  is local
- Therefore, address of a non local  $a$  in  $p$  can be stored in symbol table as
  - $(n_p - n_a, \text{offset of } a \text{ in record of activation having } a)$

# How to setup access links?

- Code to setup access links is part of the calling sequence.
- suppose procedure  $p$  at depth  $n_p$  calls procedure  $x$  at depth  $n_x$ .
- The code for setting up access links depends upon *whether or not* the called procedure is nested within the caller.

# How to setup access links?

$$np < nx$$

- Called procedure  $x$  is nested more deeply than  $p$ .
- Therefore,  $x$  must be declared in  $p$ .
- The access link in  $x$  must point to the access link of the activation record of the caller just below it in the stack

# How to setup access links?

$$np \geq nx$$

- From scoping rules enclosing procedure at the depth  $1, 2, \dots, nx-1$  must be same.
- Follow  $np-(nx-1)$  links from the caller.
- We reach the most recent activation of the procedure that statically encloses both  $p$  and  $x$  most closely.
- The access link reached is the one to which access link in  $x$  must point.
- $np-(nx-1)$  can be computed at compile time.

# Procedure Parameters

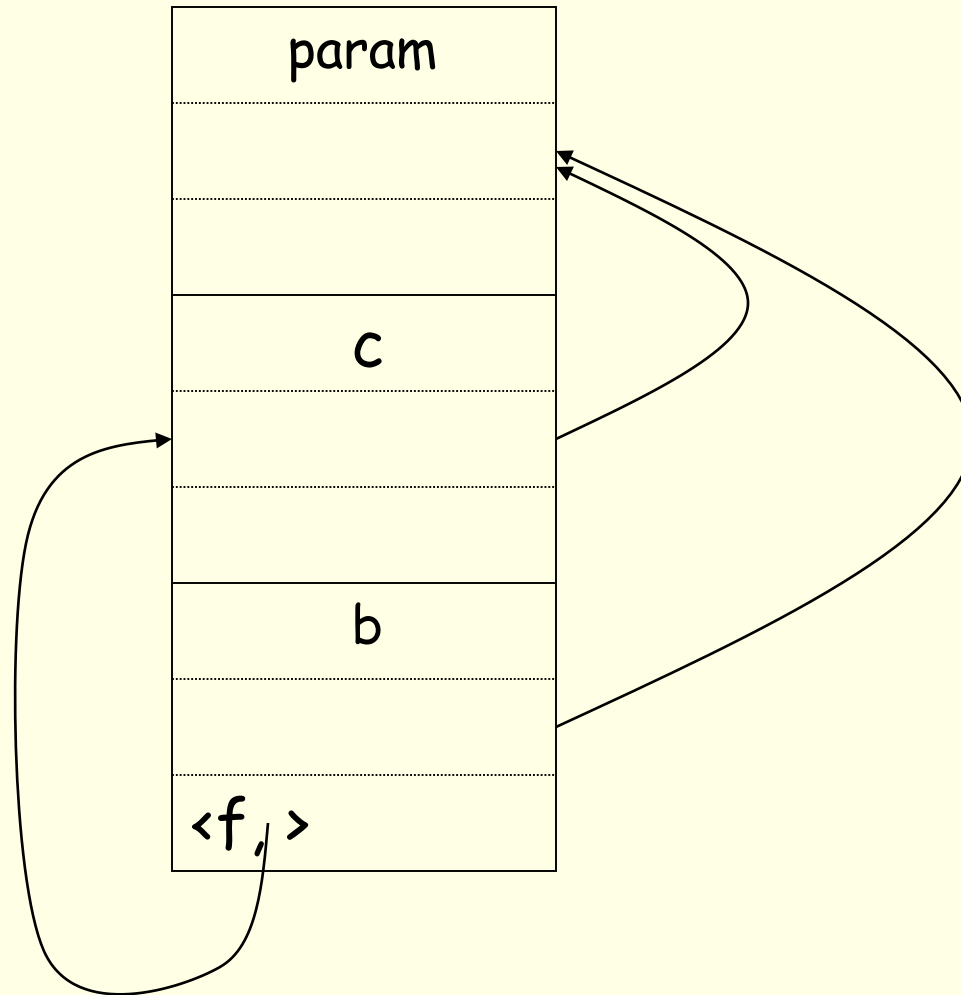
```
program param (input,output);  
  procedure b( function h(n:integer): integer);  
    begin  
      print (h(2))  
    end;  
  procedure c;  
    var m: integer;  
    function f(n: integer): integer;  
      begin  
        return m + n  
      end;  
    begin  
      m :=0; b(f)  
    end;  
begin  
  c  
end.
```



# Procedure Parameters ...

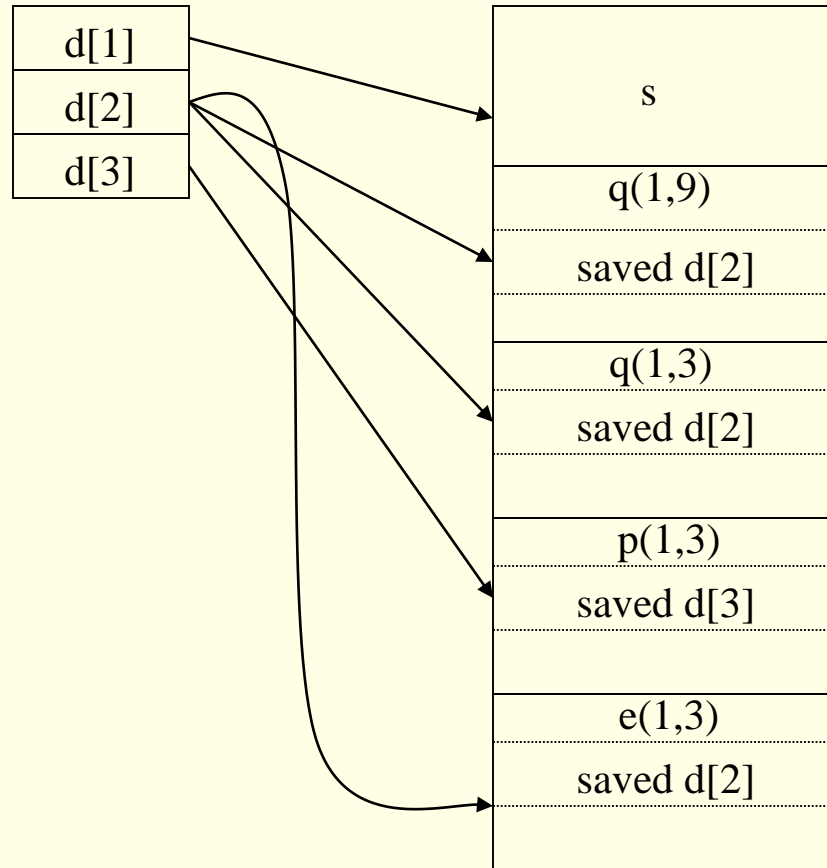
- Scope of  $m$  does not include procedure  $b$
- within  $b$ , call  $h(2)$  activates  $f$
- how is access link for activation of  $f$  is set up?
- a nested procedure must take its access link along with it
- when  $c$  passes  $f$ :
  - it determines access link for  $f$  as if it were calling  $f$
  - this link is passed along with  $f$  to  $b$
- When  $f$  is activated, this passed access link is used to set up the activation record of  $f$

# Procedure Parameters ...



# Displays

- Faster access to non locals
- Uses an array of pointers to activation records
- Non locals at depth  $i$  are in the activation record pointed to by  $d[i]$



# Setting up Displays

- When a new activation record for a procedure at nesting depth  $i$  is set up:
- Save the value of  $d[i]$  in the new activation record
- Set  $d[i]$  to point to the new activation record
- Just before an activation ends,  $d[i]$  is reset to the saved value

# Justification for Displays

- Suppose procedure at depth  $j$  calls procedure at depth  $i$
- Case  $j < i$  then  $i = j + 1$ 
  - called procedure is nested within the caller
  - first  $j$  elements of display need not be changed
  - old value of  $d[i]$  is saved and  $d[i]$  set to the new activation record
- Case  $j \geq i$ 
  - enclosing procedure at depths  $1 \dots i-1$  are same and are left un-disturbed
  - old value of  $d[i]$  is saved and  $d[i]$  points to the new record
  - display is correct as first  $i-1$  records are not disturbed

# Dynamic Scoping: Example

- Consider the following program

```
program dynamic (input, output);
```

```
  var r: real;
```

```
  procedure show;
```

```
    begin write(r) end;
```

```
  procedure small;
```

```
    var r: real;
```

```
    begin r := 0.125; show end;
```

```
begin
```

```
  r := 0.25;
```

```
  show; small; writeln;
```

```
  show; small; writeln;
```

```
end.
```

```
// writeln prints a newline character
```

## Example ...

- Output under lexical scoping

0.250      0.250

0.250      0.250

- Output under dynamic scoping

0.250      0.125

0.250      0.125

# Dynamic Scope

- Binding of non local names to storage do not change when new activation is set up
- A non local name  $x$  in the called activation refers to same storage that it did in the calling activation



# Implementing Dynamic Scope

- Deep Access
  - Dispense with access links
  - use control links to search into the stack
  - term deep access comes from the fact that search may go deep into the stack
- Shallow Access
  - hold current value of each name in static memory
  - when a new activation of  $p$  occurs a local name  $n$  in  $p$  takes over the storage for  $n$
  - previous value of  $n$  is saved in the activation record of  $p$

# Parameter Passing

- Call by value
  - actual parameters are evaluated and their r-values are passed to the called procedure
  - used in Pascal and C
  - formal is treated just like a local name
  - caller evaluates the actual parameters and places rvalue in the storage for formals
  - call has no effect on the activation record of caller

# Parameter Passing ...

- Call by reference (call by address)
  - the caller passes a pointer to each location of actual parameters
  - if actual parameter is a name then l-value is passed
  - if actual parameter is an expression then it is evaluated in a new location and the address of that location is passed

# Parameter Passing ...

- Copy restore (copy-in copy-out, call by value result)
  - actual parameters are evaluated, rvalues are passed by call by value, lvalues are determined before the call
  - when control returns, the current rvalues of the formals are copied into lvalues of the locals

# Parameter Passing ...

- Call by name (used in Algol)
  - names are copied
  - local names are different from names of calling procedure
  - Issue:

```
swap(x, y) {  
    temp = x  
    x = y  
    y = temp  
}
```

```
swap(i,a[i]):  
    temp = i  
    i = a[i]  
    a[i] = temp
```

# 3AC for Procedure Calls

$S \rightarrow \text{call id ( Elist )}$

$\text{Elist} \rightarrow \text{Elist , E}$

$\text{Elist} \rightarrow \text{E}$

- Calling sequence
  - allocate space for activation record
  - evaluate arguments
  - establish environment pointers
  - save status and return address
  - jump to the beginning of the procedure

# Procedure Calls ...

## Example

- parameters are passed by reference
- storage is statically allocated
- use param statement as place holder for the arguments
- called procedure is passed a pointer to the first parameter
- pointers to any argument can be obtained by using proper offsets

# Procedure Calls

- Generate three address code needed to evaluate arguments which are expressions
- Generate a list of param three address statements
- Store arguments in a list

$S \rightarrow \text{call id ( Elist )}$

for each item p on queue do emit('param' p)  
emit('call' id.place)

$\text{Elist} \rightarrow \text{Elist , E}$

append E.place to the end of queue

$\text{Elist} \rightarrow \text{E}$

initialize queue to contain E.place



# Procedure Calls

- Practice Exercise:

How to generate intermediate code for parameters passed by value? Passed by reference?