PUNE VIDYARTHI GRIHA’s

COLLEGE OF ENGINEERING, NASHIK.

“ RUN TIME STORAGE ORGANIZATION ”

PREPARED BY :

PROF. ANAND N. GHARU

ASSISTANT PROFESSOR

COMPUTER DEPARTMENT

SUBJECT – COMPILER (BE COMPUTER SPPU-2019)

CONTENTS :-

1. Storage Management – Static, Stack and Heap
2. Activation Record, static and control links, parameter passing, return value, passing array and variable number of arguments
3. Static and Dynamic scope, Dangling Pointers, translation of control structures – if, if-else statement, Switch-case, while, do -while statements, for, nested blocks,
4. display mechanism, array assignment, pointers, function call and return. Translation of OO constructs: Class, members and Methods.

Run-time Storage

- Run-Time Environment
- Storage Organization
- Storage Allocation Strategies
- Dynamic Storage Allocation

Run-Time Environments

- The compiler must implement various abstractions in the source language definition such as
  - Names used in a program
  - Define the scope of variables
  - Data types
  - Operators
  - Procedures
  - Parameters and
  - Flow of control constructs.
- The compiler must co-operate with operating system and other systems software to support the implementation of these abstractions on the target machine. This can be done by the compiler by creating run-time environment.

Run-Time Environments

What is run-time environment in compiler design?

- A run-time environment in compiler design deals variety of issues such as:

1. Managing the processor stack.
2. Layout and allocation of memory for various variables used in the source program.
3. Instructions to copy the actual parameters on top of the stack when a function is called.
4. Allocating and de-allocating the memory dynamically with the help of operating system.
5. The mechanism used by the target program to access variables.
6. The mechanism to pass parameters.
7. The interfaces to the operating system, input/output devices and other programs.

Run-Time Environments

- A lot has to happen at run time to get your program running.
- At run time, we need a system to map NAMES (in the source program) to STORAGE on the machine.
- Allocation and deallocation of memory is handled by a RUN-TIME SUPPORT SYSTEM typically linked and loaded along with the compiled target code.
- One of the primary responsibilities of the run-time system is to manage ACTIVATIONS of procedures.

Run-Time Environments

- For allocating memory to data items, the following information is required:

1. Size of data item
2. The type of data item
3. Dimensions of the data item
4. Scope of the data item

Run-Time Environments

- The organization of data objects in memory depends on the source language features:

1. Recursion
2. Parameter passing mechanism
3. Local names
4. Non-local names
5. Dynamic data structures

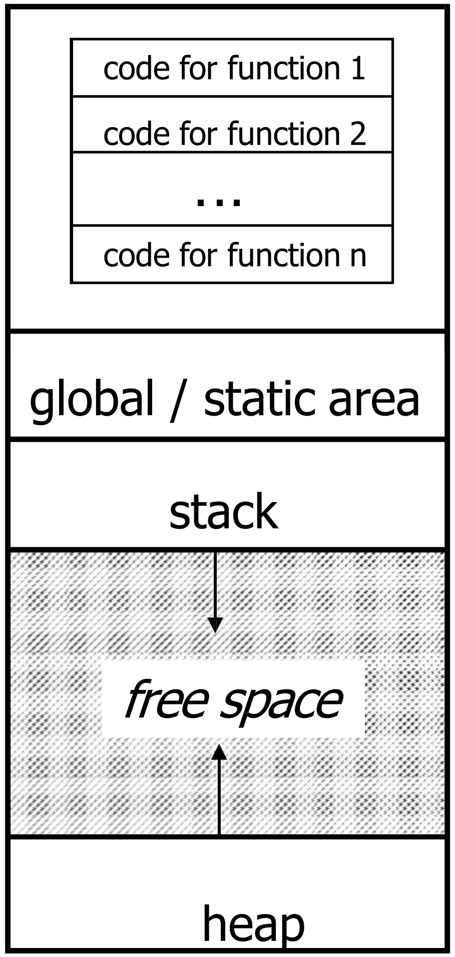
Run-time Storage

- Run-Time Environment
- Storage Organization
- Storage Allocation Strategies
- Dynamic Storage Allocation

Storage Organization

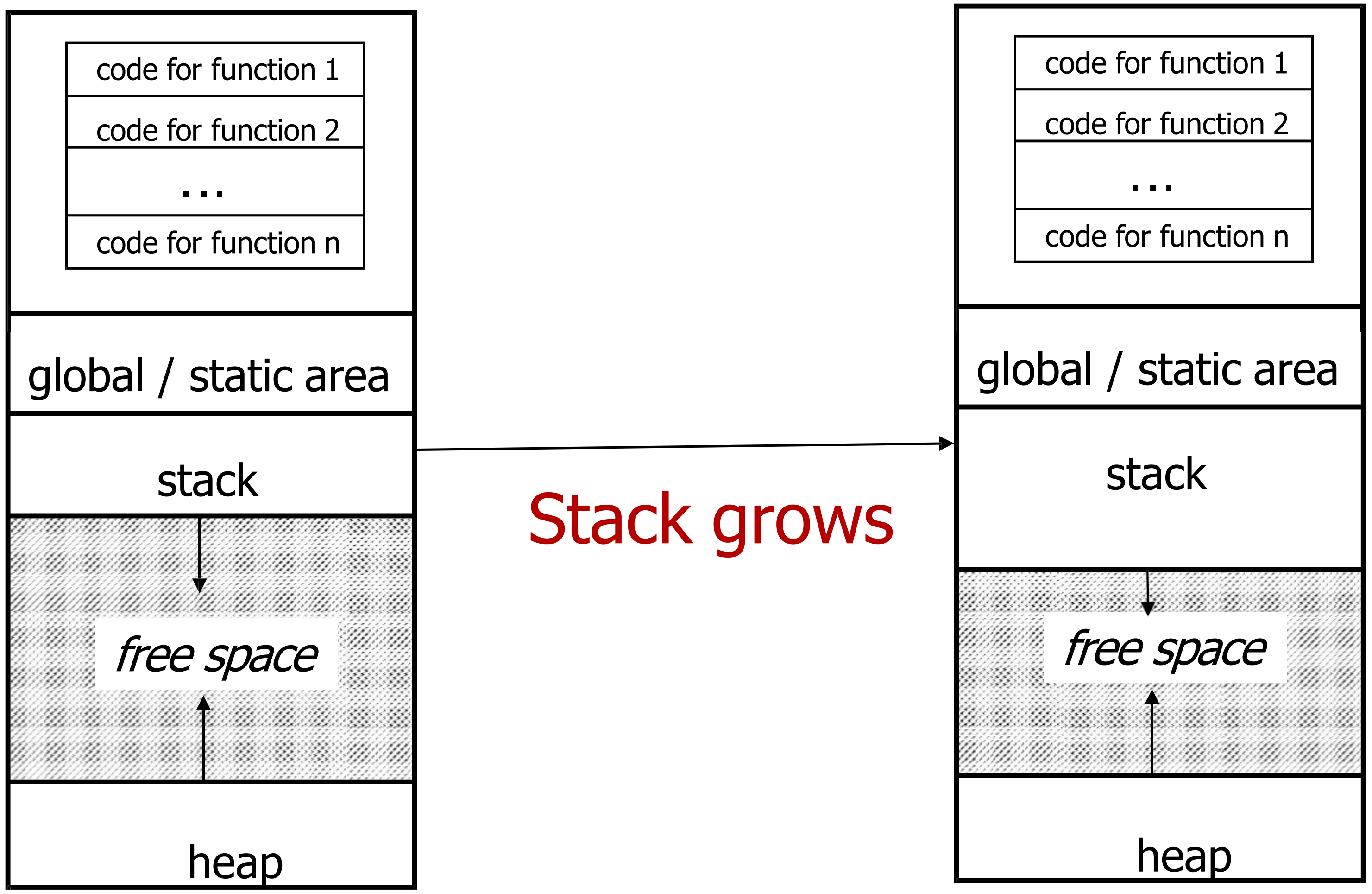
- Suppose that the compiler obtains memory from the OS so that it can execute the compiled program
  - Program gets loaded on a newly created process
- This runtime storage must hold
  - Generated target code
  - Data objects
  - A counterpart of the control stack to keep track of procedure activations

Runtime Memory



- PASCAL and C use extensions of the control stack to manage activations of procedures
- Stack contains information about register values, value of program counter and data objects whose lifetimes are contained in that of an activation
- Heap holds all other information. For example, activations that cannot be represented as a tree.
- By convention, stack grows down and the top of the stack is drawn towards the bottom of this slide (value of top is usually kept in a register)

Runtime Memory



Activation Record

- Information needed by a single execution of a procedure is managed using an activation record or frame
  - Not all compilers use all of the fields
  - Pascal and C push activation record on the runtime stack when procedure is called and pop the activation record off the stack when control returns to the caller

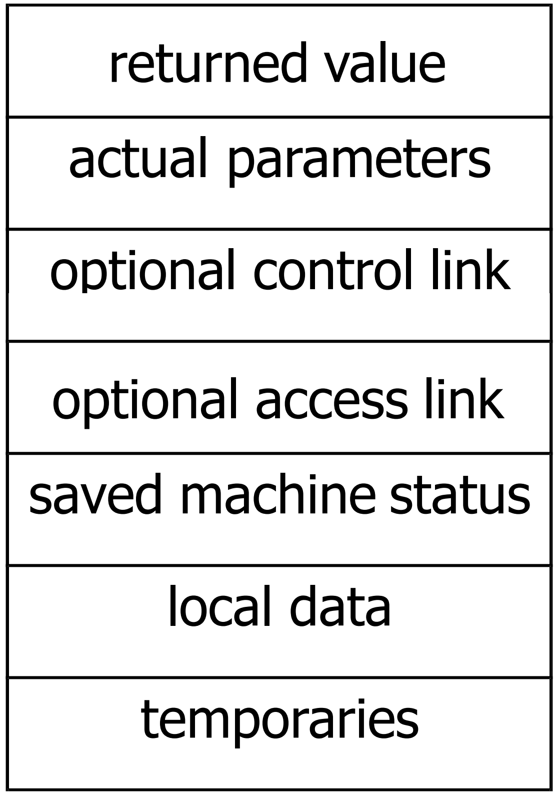


Fig: A typical Activation Record

Activation Record

1) Temporary values
   Ex. those arising in the evaluation of expressions
2) Local data
   Data that is local to an execution of the procedure
3) Saved machine status
   State of the machine info before procedure is called. Values of program counter and machine registers that have to be restored when control returns from the procedure

returned value
actual parameters
optional control link
optional access link
saved machine status
local data
temporaries

Activation Record

4) Access Link
   refer to non-local data held in other activation records
5) Control link
   points to the activation record of the caller
6) Actual parameters
   used by the calling procedure to supply parameters to the called procedure
   (in practice these are passed in registers)
7) Returned value
   used by the called procedure to return a value to the calling procedure
   (in practice it is returned in a register)

returned value
actual parameters
optional control link
optional access link
saved machine status
local data
temporaries

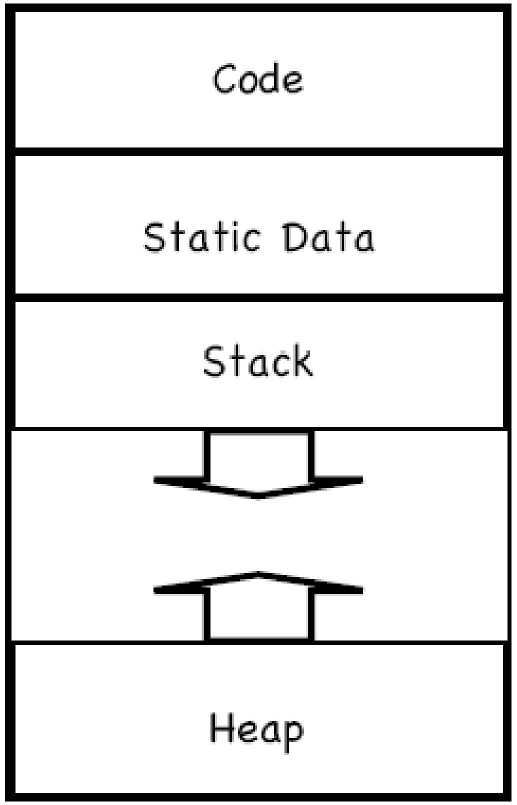
Local data

- The field for local data is set when declarations in a procedure are examined during compile time
  - Variable-length data is not stored here
  - Keep a count of the memory locations that have been allocated so far
  - Determine a relative address (offset) of the storage for a local with respect to some position (e.g. beginning of the frame)
    - Multibyte objects are stored in consecutive bytes and given the address of the first byte

Run-time Storage

- Run-Time Environment
- Storage Organization
- Storage Allocation Strategies
- Dynamic Storage Allocation

Organization of storage



- Fixed-size objects can be placed in predefined locations.
- The heap and the stack need room to grow, however.

Run-time stack and heap

- The STACK is used to store:
  - Procedure activations.
  - The status of the machine just before calling a procedure, so that the status can be restored when the called procedure returns.
- The HEAP stores data allocated under program control

Storage Allocation Strategies

- The various storage allocation strategies to allocate storage in different data areas of memory are:

1. Static Allocation
   - Storage is allocated for all data objects at compile time
2. Stack Allocation
   - The storage is managed as a stack
3. Heap Allocation (It is one of Dynamic Storage Allocation)
   - The storage is allocated and deallocated at runtime from a data area known as heap

Static Allocation

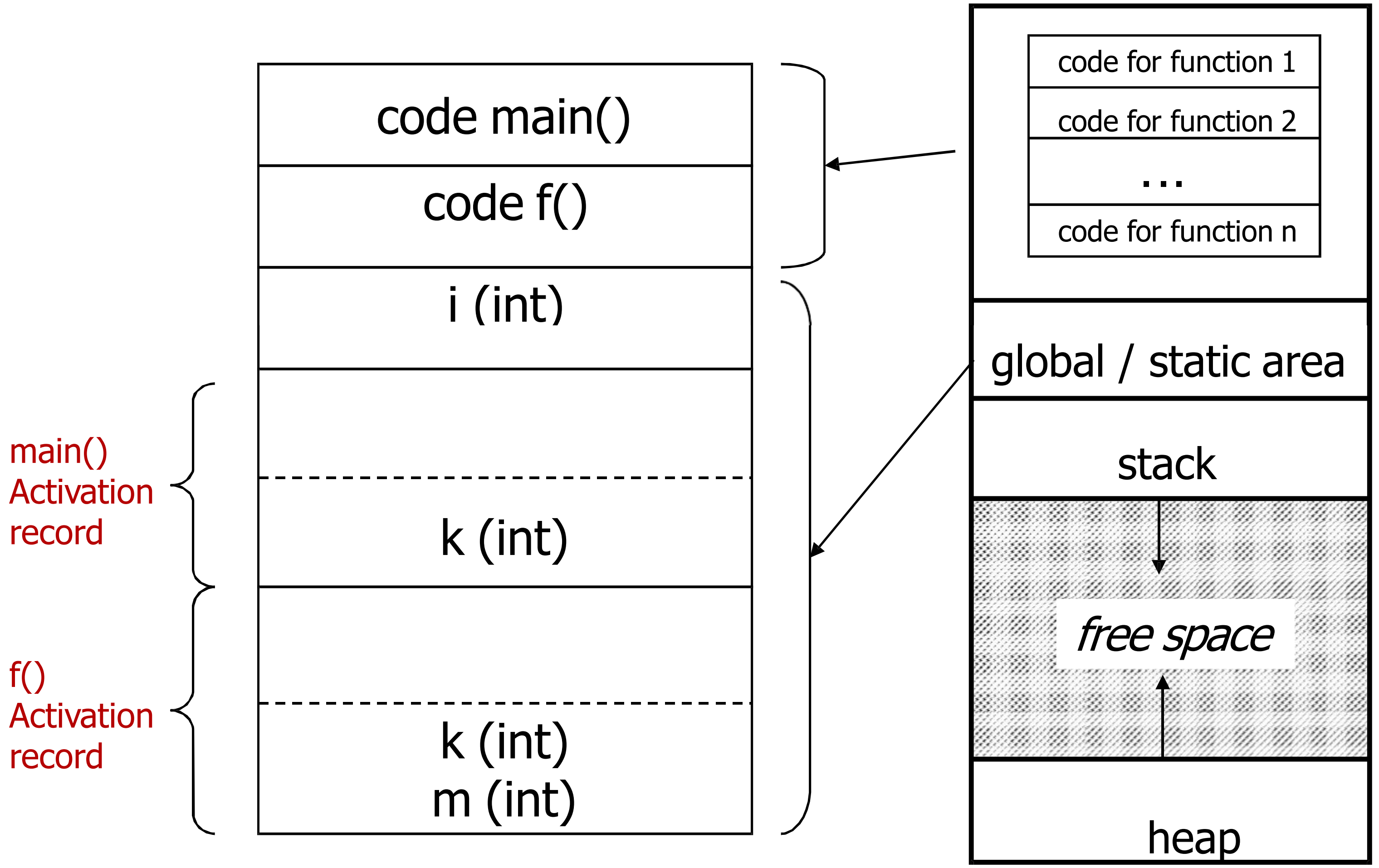
- In a static environment (Fortran 77) there are a number of restrictions:
  - Size of data objects are known at compile time
  - No recursive procedures
  - No dynamic memory allocation
- Only one copy of each procedure activation record exists at time t
  - We can allocate storage at compile time
    - Bindings do not change at runtime
    - Every time a procedure is called, the same bindings occur

Static Allocation

- Statically allocated names are bound to relocatable storage at compile time.
- Storage bindings of statically allocated names never change.
- The compiler uses the type of a name (retrieved from the symbol table) to determine storage size required.
- The required number of bytes (possibly aligned) is set aside for the name.
- The relocatable address of the storage is fixed at compile time.

Static Allocation

```
int i = 10;

int f(int j)
{
  int k;
  int m;

}

main()
{
  int k;
  f(k);
}
```



Static allocation

- Limitations:
  - The size required must be known at compile time.
  - Recursive procedures cannot be implemented statically.
  - No data structure can be created dynamically as all data is static.

Stack-based Allocation

- In a stack-based allocation, the previous restrictions are lifted (Pascal, C, etc)
  - procedures are allowed to be called recursively
    - Need to hold multiple activation records for the same procedure
    - Created as required and placed on the stack
      - ✓ Each record will maintain a pointer to the record that activated it
      - ✓ On completion, the current record will be deleted from the stack and control is passed to the calling record
  - Dynamic memory allocation is allowed
  - Pointers to data locations are allowed

Stack-dynamic allocation

- Storage is organized as a stack.
- Activation records are pushed and popped.
- Locals and parameters are contained in the activation records for the call.
- This means locals are bound to fresh storage on every call.
- We just need a stack_top pointer.
- To allocate a new activation record, we just increase stack_top.
- To deallocate an existing activation record, we just decrease stack_top.

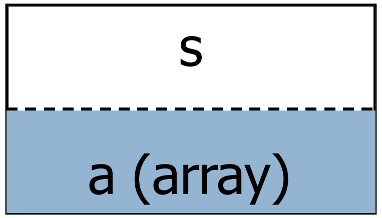
Stack-based Allocation

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer):
Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Position in Activation Tree

s

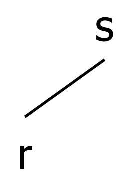
Activation Records On Stack



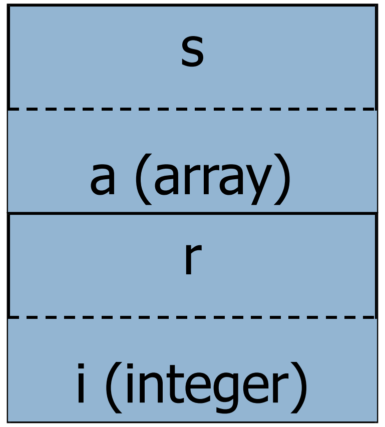
Stack-based Allocation

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer):
Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Position in Activation Tree



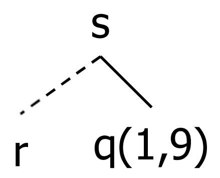
Activation Records On Stack



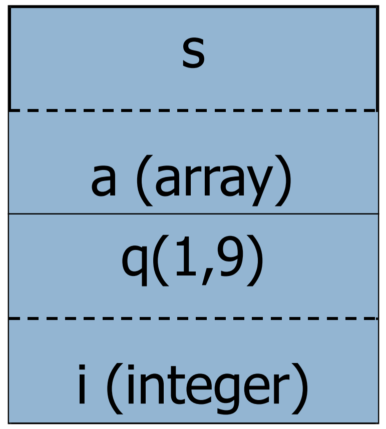
Stack-based Allocation

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer):
Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Position in Activation Tree



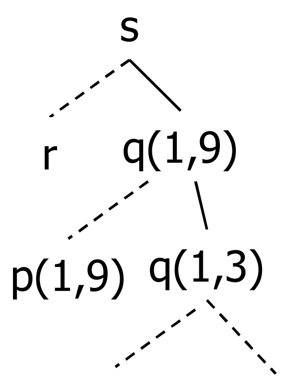
Activation Records On Stack



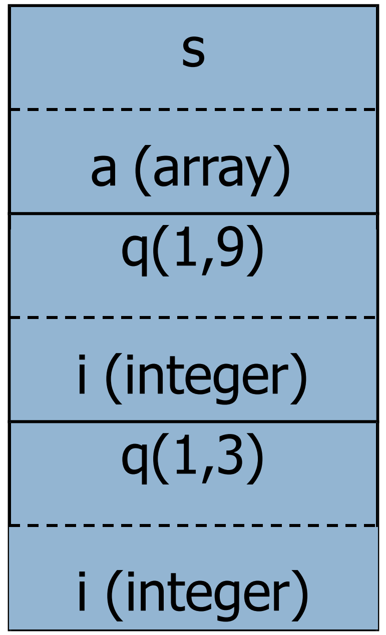
Stack-based Allocation

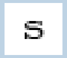
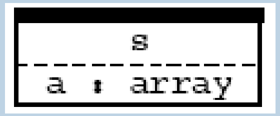
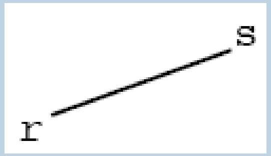
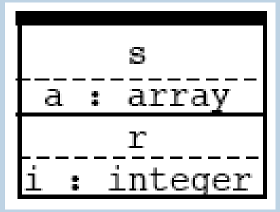
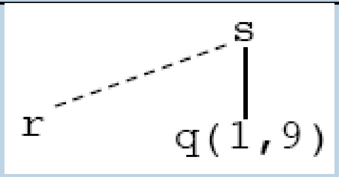
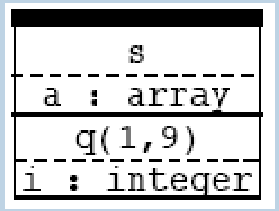
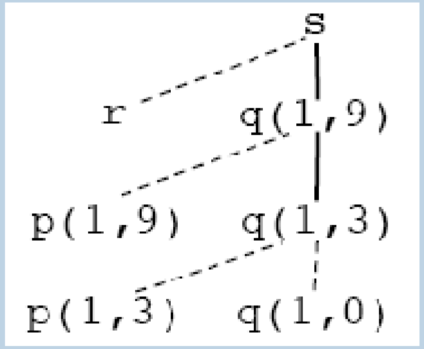
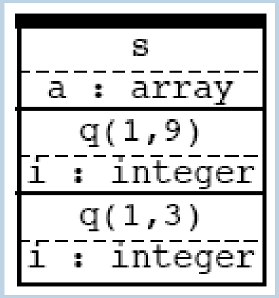
```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer):
Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Position in Activation Tree



Activation Records On Stack



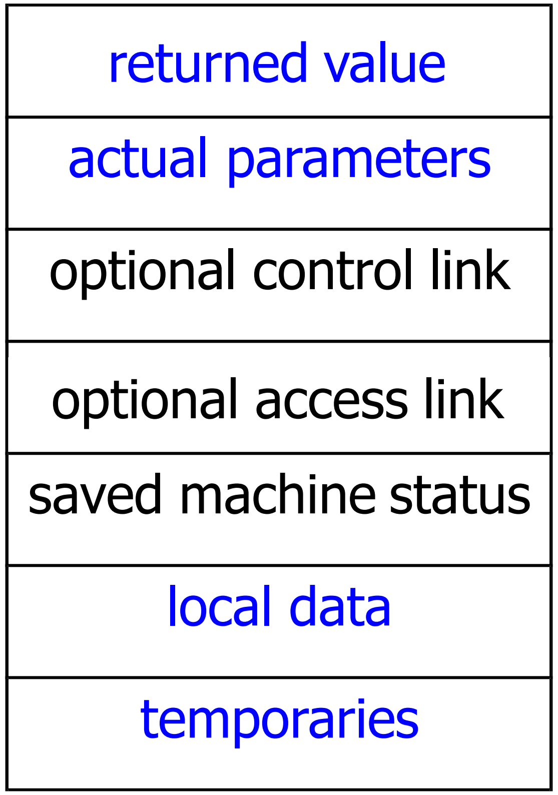
Position in Activation Tree	Activation Records on the Stack
	
	
	
	

Address generation in stack allocation

- The position of the activation record on the stack cannot be determined statically.
- Therefore the compiler must generate addresses RELATIVE to the activation record.
- We generate addresses of the form
  stack_top + offset

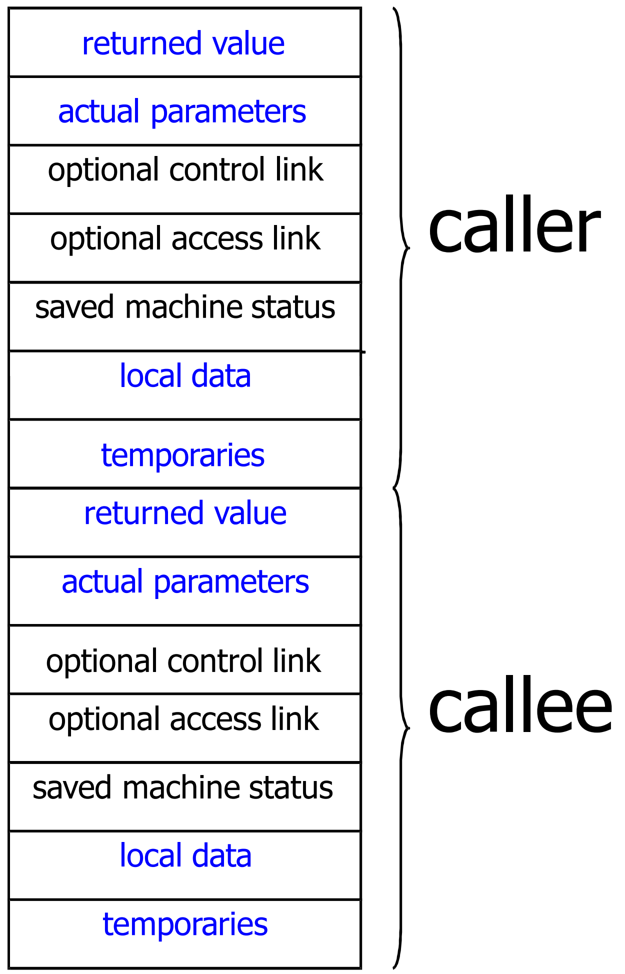
Calling Sequences

- Procedure calls are implemented by generating calling sequences in the target code
  - Call sequence: allocates activation record and enters information into fields
  - Return sequence: restores the state of the machine so that the calling procedure can continue execution



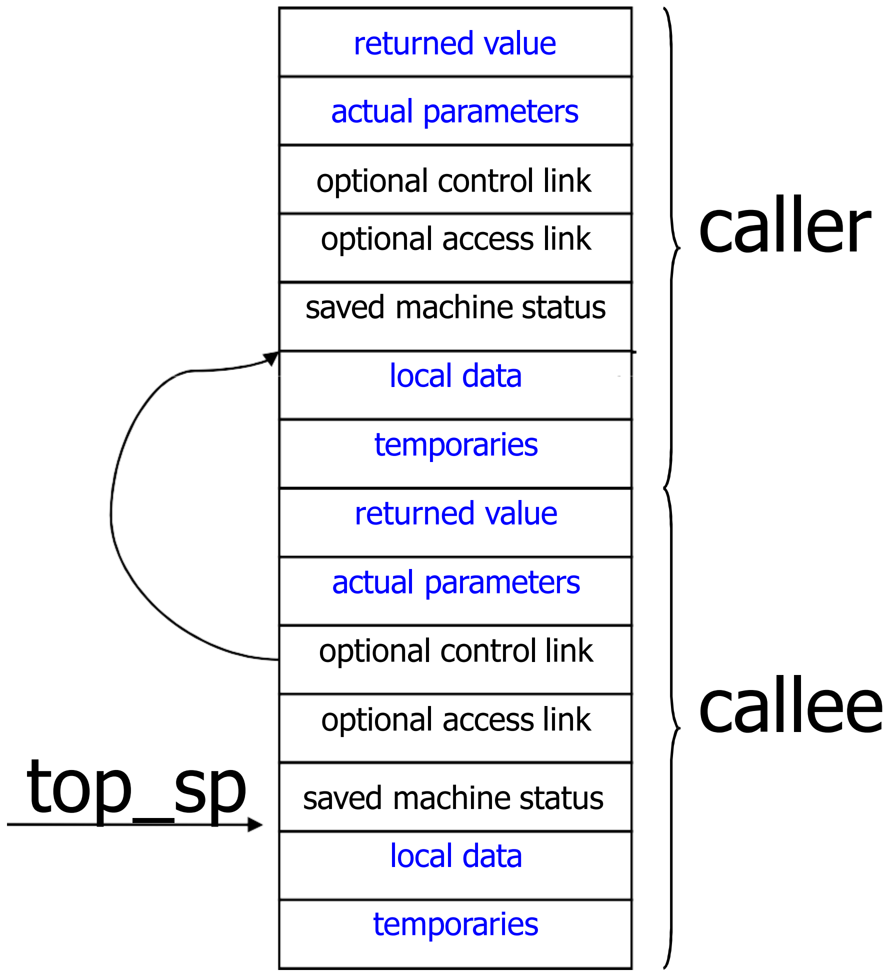
Calling Sequences

- Why placing returned value and actual parameters next to the activation record of the caller?
  - Caller can access these values using offsets from its own activation record
  - No need to know the middle part of the callee′s activation record



Calling Sequences

- How do we calculate offset?
  - Maintain a register that points to the end of the machine status field in an activation record
  - Top_sp is known to the caller, so it can be responsible for setting it before control flows to the called procedure
  - Callee can access its temporaries and local data using offsets from top_sp

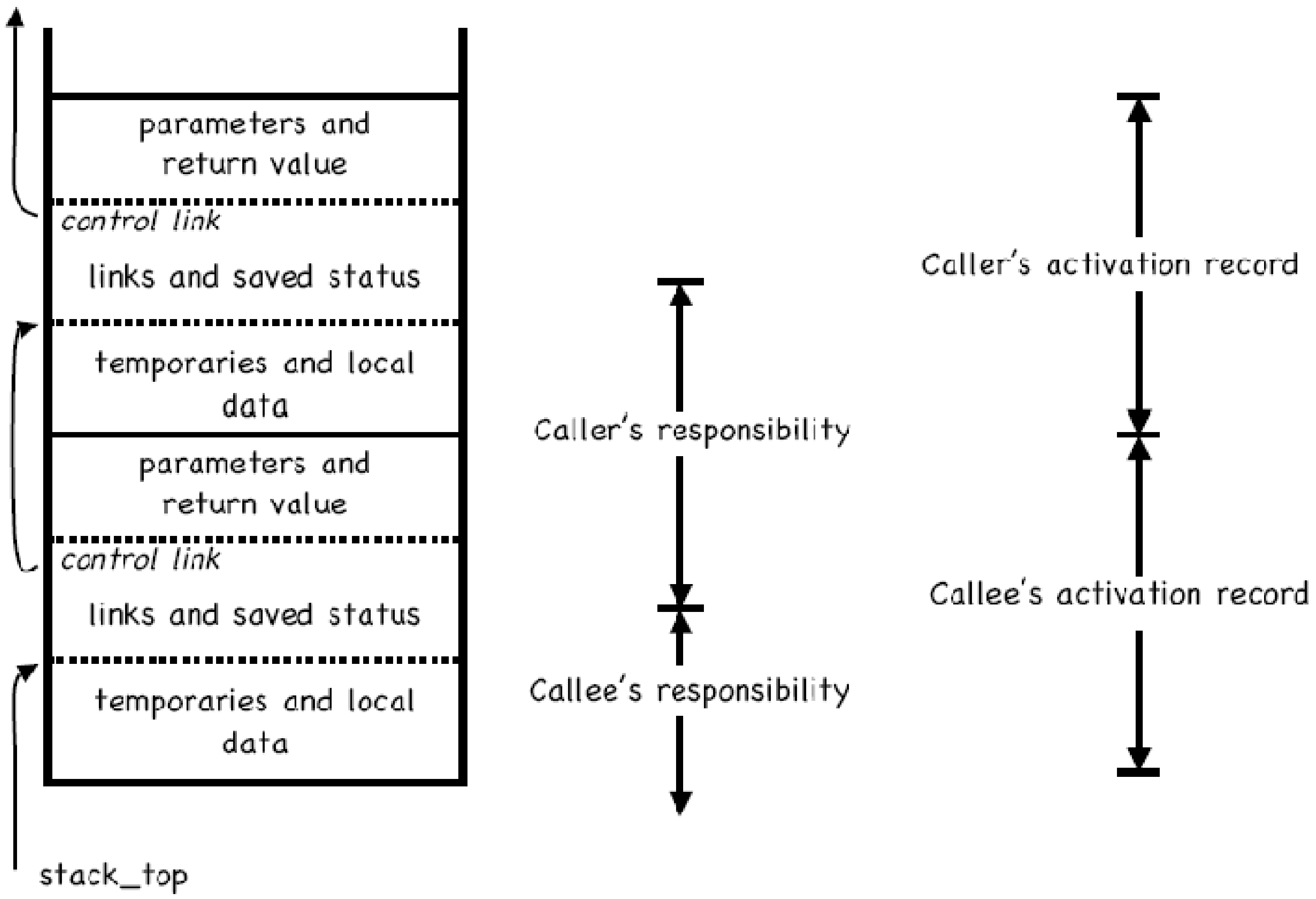


Call Sequence

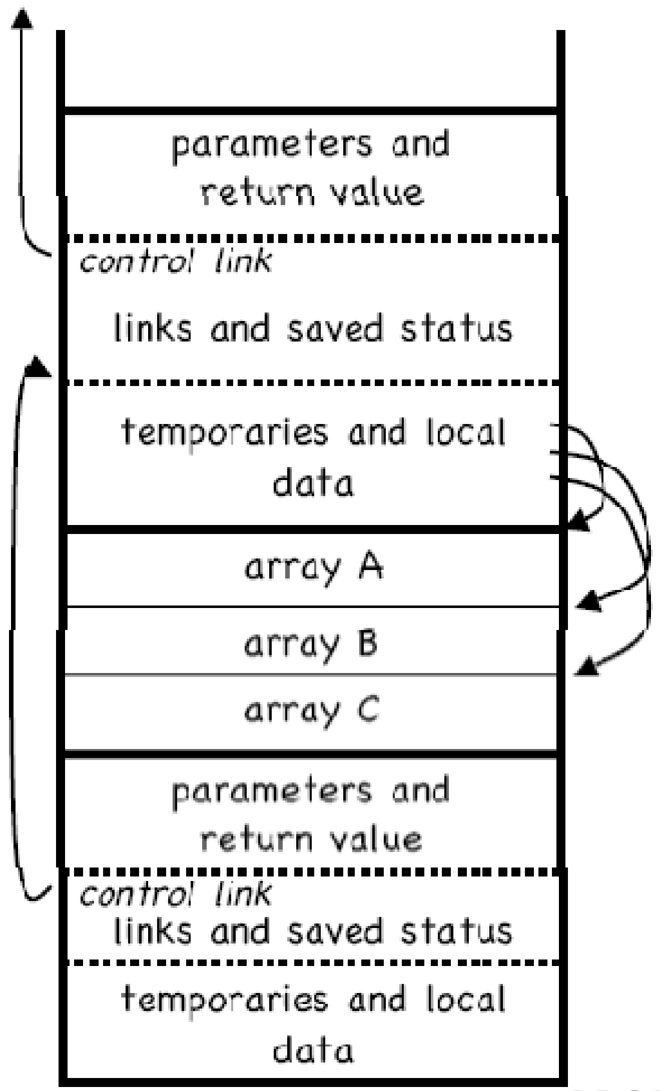
- The caller evaluates actuals
- The caller
  - stores a return address and the old value of top_sp into the callee′s activation record
  - increments top_sp; that is moved past the caller′s local data and temporaries and the callee′s parameter and status fields
- The callee
  - saves register values and other status information
- The callee
  - initializes its local data and begins execution

Return Sequence

- The callee places a return value next to the activation record of the caller
- Using the information in the status field, the callee
  - restores top_sp and other registers
  - braches to a return address in the caller’s code
- Although top_sp has been decremented, the caller can copy the returned value into its own activation record and use it to evaluate an expression


parameters and return value
control link
links and saved status
temporaries and local data
parameters and return value
control link
links and saved status
temporaries and local data
stack_top
Caller's responsibility
Callee's responsibility
Caller's activation record
Callee's activation record

Example of variable- length data


parameters and return value
control link
links and saved status
temporaries and local data
array A
array B
array C
parameters and return value
control link
links and saved status
temporaries and local data

- All variable-length data is pointed to from the local data area.

Dangling References

- Whenever storage is deallocated, the problem of dangling references arises
  - Occurs when there is a reference to storage that has been deallocated
  - Logical error
    - Mysterious bugs can appear

Dangling References

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

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

What's the problem?

Dangling References

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

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

- Local variable i only exists in dangle()
- When procedure completes execution and control is transferred to main(), the space for i does not exist anymore (pop activation record for dangle off the stack)
- Pointer p is a dangling reference

Stack Allocation Advantages and Disadvantages

Advantages:

- It supports recursion as memory is always allocated on block entry.
- It allows to create data structures dynamically.
- It allows an array declaration like A(I, J), since actual allocation is made only at execution time. The dimension bounds need not be known at compile time.

Disadvantages:

- Memory addressing has to be effected through pointers and index registers which may be store them, static allocation especially in case of array reference.

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

There are two aspects of dynamic allocation :

- Runtime allocation and de-allocation of data structures
- Languages like Algol have dynamic data structures and it reserves some part of memory for it.

If a procedure wants to put a value that is to be used after its activation is over then we cannot use stack for that purpose. That is language like Pascal allows data to be allocated under program control. Also in certain language a called activation may outlive the caller procedure. In such a case last-in-first-out queue will not work and we will require a data structure like heap to store the activation. The last case is not true for those languages whose activation trees correctly depict the flow of control between procedures.

Heap Allocation

- Some languages do not have tree-structured allocations.
- In these cases, activations have to be allocated on the heap.
- This allows strange situations, like callee activations that live longer than their callers' activations.
- This is not common.

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
- For each size of interest keep a linked list of free blocks of that size

Heap Allocation

- Fill a request of size swith block of size s' where s' is the smallest size greater than or equal to s
- For large blocks of storage use heap manager
- For large amount of storage computation may take some time to use up memory so that time taken by the manager may be negligible compared to the computation time

Heap Allocation

- For efficiency reasons we can handle small activations and activations of predictable size as a special case as follows:

1. For each size of interest, keep a linked list if free blocks of that size
2. If possible, fill a request for size s with a block of size s', where s' is the smallest size greater than or equal to s. When the block is eventually de-allocated, it is returned to the linked list it came from.
3. For large blocks of storage use the heap manger.

Heap manger will dynamically allocate memory. This will come with a runtime overhead. As heap manager will have to take care of defragmentation and garbage collection. But since heap manger saves space otherwise we will have to fix size of activation at compile time, runtime overhead is the price worth it.

Access to non-local names

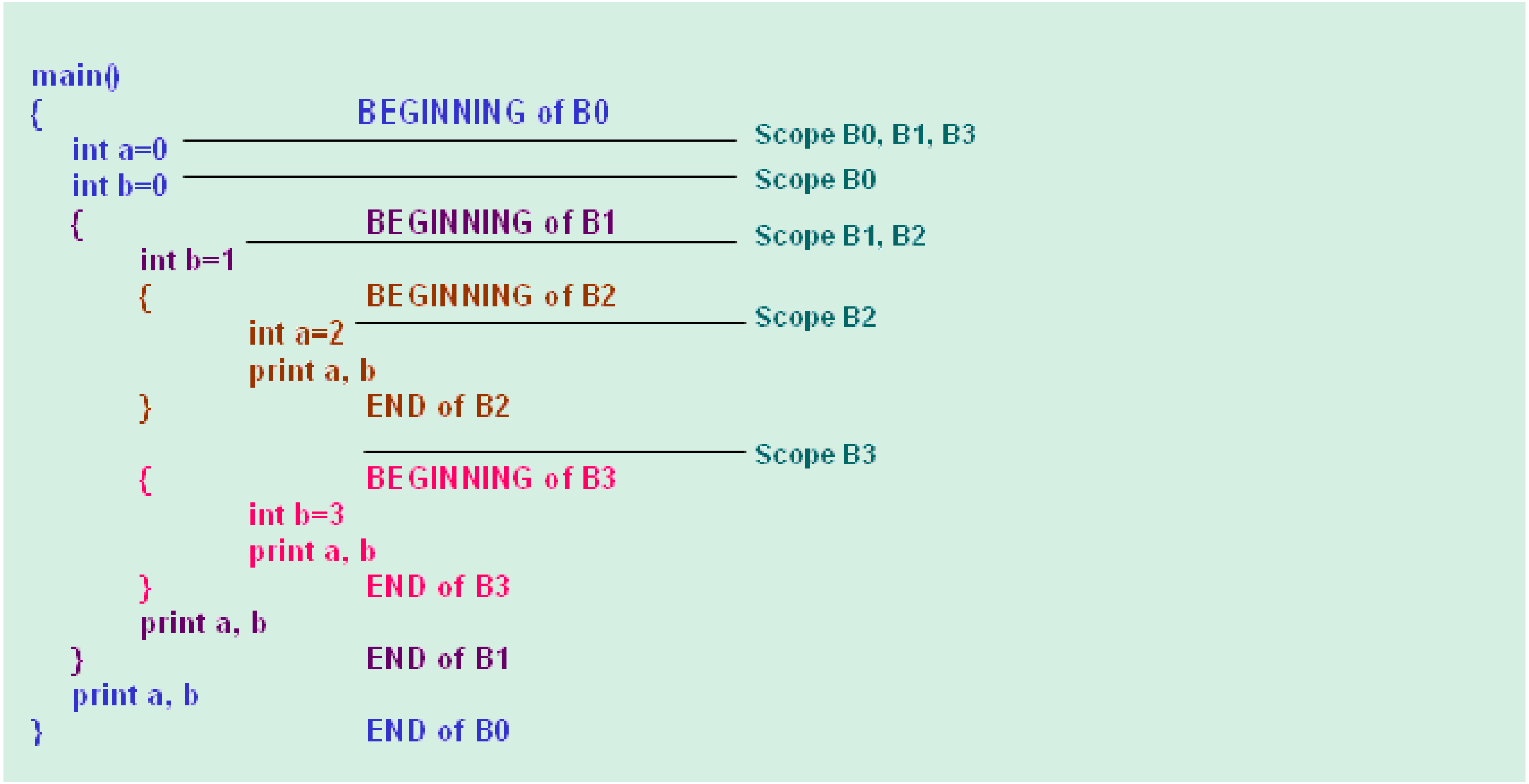
- Scope rules determine the treatment of non-local names
- A common rule is lexical scoping or static scoping (most languages use lexical scoping)
- The scope rules of a language decide how to reference the non-local variables. There are two methods that are commonly used:

1. Static or Lexical scoping: It determines the declaration that applies to a name by examining the program text alone. E.g., Pascal, C and ADA.
2. Dynamic Scoping: It determines the declaration applicable to a name at run time, by considering the current activations. E.g., Lisp

Block

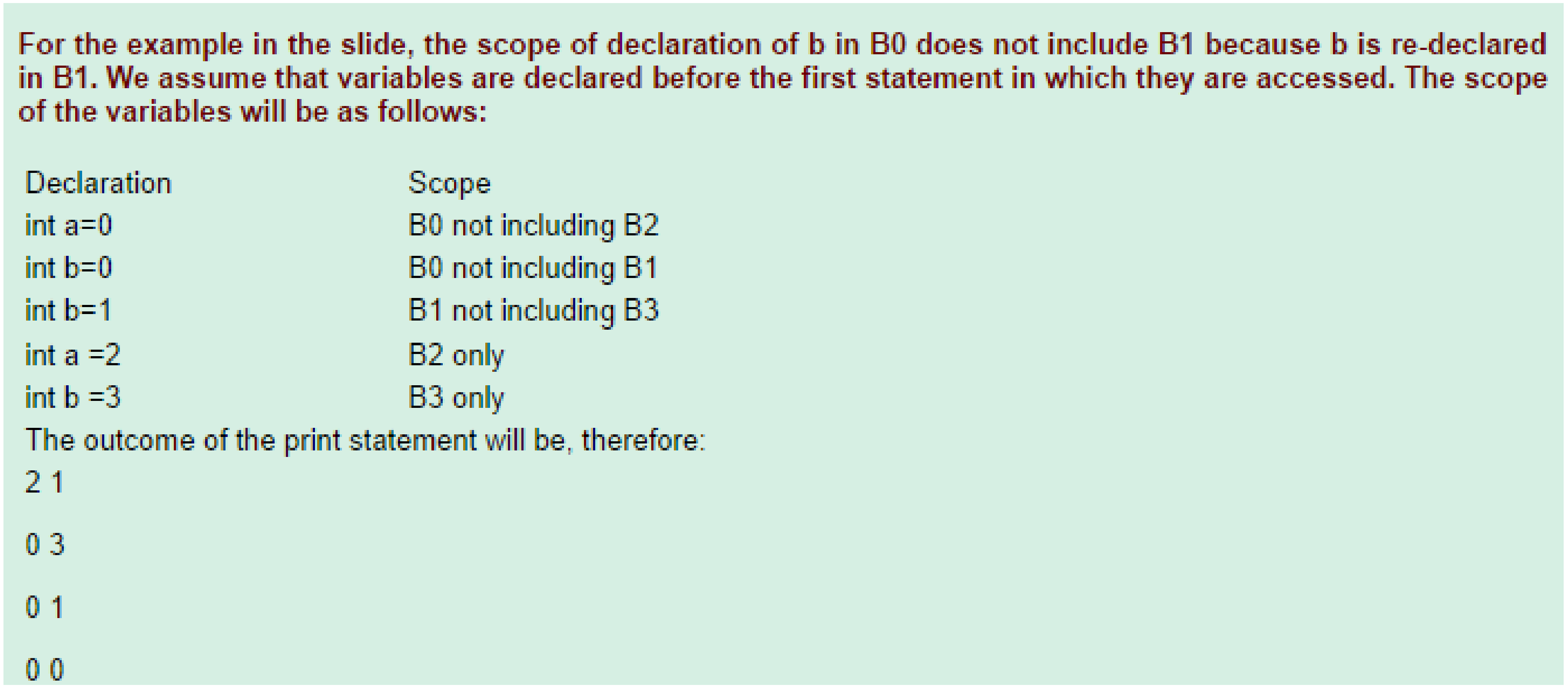
- Blocks can be nested
- The property is referred to as block structured
- Scope of the declaration is given by most closely nested rule
  - The scope of a declaration in block B includes B
  - If a name X is not declared in B then an occurrence of X is in the scope of declarator X in B' such that
    - B' has a declaration of X
    - B' is most closely nested around B

Block Example



```
main()
{                         BEGINNING of B0
    int a=0                                          Scope B0, B1, B3
    int b=0                                          Scope B0
    {                       BEGINNING of B1
        int b=1                                      Scope B1, B2
        {                   BEGINNING of B2
            int a=2                                  Scope B2
            print a, b
        }                   END of B2
                                                     Scope B3
        {                   BEGINNING of B3
            int b=3
            print a, b
        }                   END of B3
        print a, b
    }                       END of B1
    print a, b
}                           END of B0
```

Block Example



For the example in the slide, the scope of declaration of b in B0 does not include B1 because b is re-declared in B1. We assume that variables are declared before the first statement in which they are accessed. The scope of the variables will be as follows:

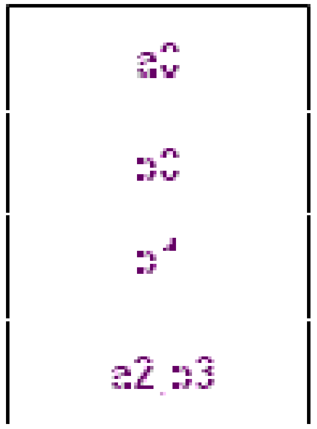
Declaration	Scope
int a=0	B0 not including B2
int b=0	B0 not including B1
int b=1	B1 not including B3
int a =2	B2 only
int b =3	B3 only

The outcome of the print statement will be, therefore:

```
2 1

0 3

0 1

0 0
```

Blocks

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



Blocks

- There are two methods of implementing block structure:

1. Stack Allocation : This is based on the observation that scope of a declaration does not extend outside the block in which it appears, the space for declared name can be allocated when the block is entered and de-allocated when controls leave the block. The view treat block as a "parameter less procedure" called only from the point just before the block and returning only to the point just before the block.
2. Complete Allocation : Here you allocate the complete memory at one time. If there are blocks within the procedure, then allowance is made for the storage needed for declarations within the books. If two variables are never alive at the same time and are at same depth they can be assigned same storage.

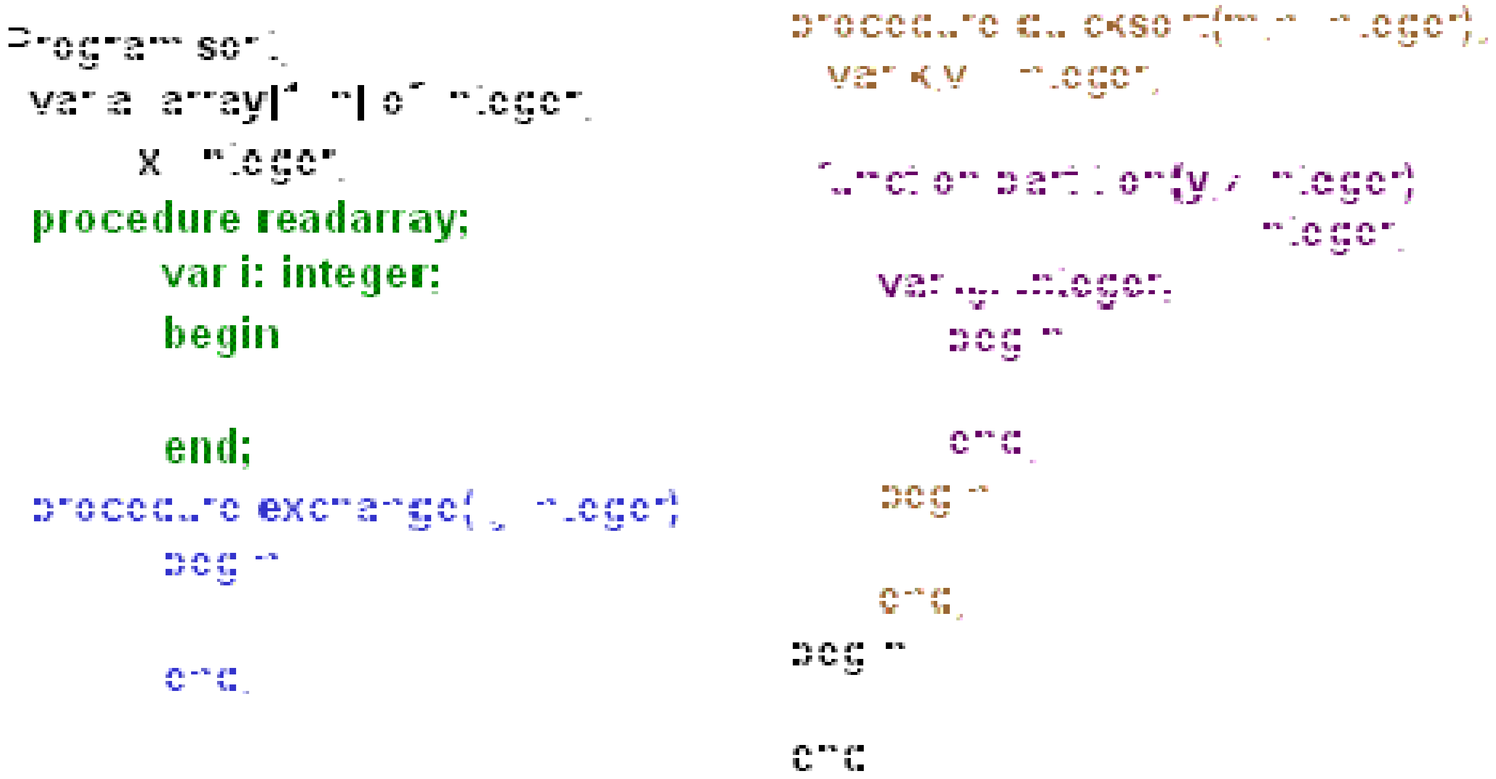
Lexical scope without nested procedures

- 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
- A non local name must be local to the top of the stack
- Stack allocation of non local has advantage:
  - Non locals have static allocations
  - Procedures can be passed/returned as parameters

Lexical scope without nested procedures

- In languages like C nested procedures are not allowed. That is, you cannot define a procedure inside another procedure. So, if there is a non- local reference to a name in some function then that variable must be a global variable. The scope of a global variable holds within all the functions except those in which the variables have been re- declared. Storage for all names declared globally can be allocated statically. Thus their positions will be known at compile time. In static allocation, we use stack allocation. Any other name must be a local of the activation at the top of the stack, accessible through the top pointer. Nested procedures cause this scheme to fail because a non-local may then refer to a local of parent variable which may be buried deep in the stack and not at the top of stack. An important benefit of static allocation for non- locals is that declared procedures can freely be passed as parameters and returned as results (a function is passed in C by passing a pointer to it).

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.
```

Scope with nested procedures

- The above example contains a program in Pascal with nested procedure sort

1. Readarray
2. Exchange
3. Quicksort
4. Partition

- Here we apply the most closely nested rule for deciding scoping of variables and procedure names. The procedure exchange called by partition , is non-local to partition. Applying the rule, we first check if *exchange* is defined within *quicksort* ; since it is not, we look for it in the main program sort.

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

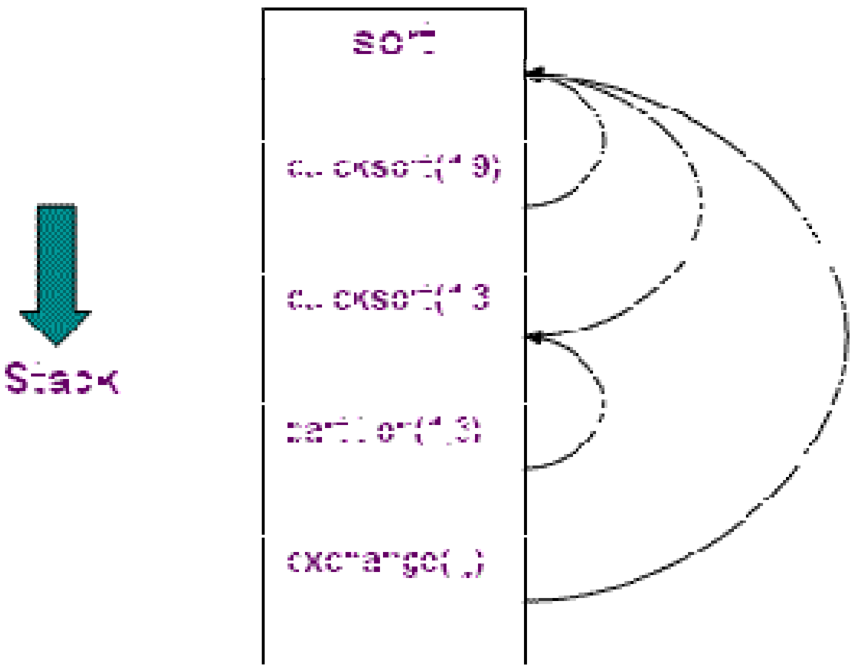
- Include a field 'access link' in the activation record
- If p is nested in q then access link of p points to the access link in most recent activation of q

Nesting Depth

- Nesting Depth: The notion of nesting depth is used to implement lexical scope. The main program is assumed to be at nesting depth 1 and we add 1 to the nesting depth as we go from an enclosing to an enclosed procedure.
- Access Links: To implement the lexical scope for nested procedures we add a pointer called an access link to each activation record. If a procedure p is nested immediately within q in the source text, then the access link in an activation record for p points to the access link in the record for most recent activation of q .

Nesting Depth

- The access links for finding storage for non-locals are shown below.



Access to non local names

- Suppose procedure p at depth np refers to a non- local a at depth na, then storage for a can be found as
  - follow (np-na) access links from the record at the top of the stack
  - after following (np-na) links we reach procedure for which a is local
- Therefore, address of a non local a in procedure p can be stored in symbol table as

(np-na, offset of a in record of activation having a)

Access to non local names

- Suppose procedure p at depth np refers to a non-local a with nesting depth na = np. The storage for a can be found as follows:
  - When control is in p, an activation record for p is at the top of the stack. Follow the (np - na) access links from the record at the top of the stack.
  - After following (np - na) links, we reach an activation record for the procedure that a is local to. As discussed earlier, its storage is at a fixed offset relative to a position in the record. In particular, the offset can be relative to the access link.
- The address of non-local a in procedure p is stored as following in the symbol table:

  (np - na, offset within the activation record containing a)

How to setup access links?

- suppose procedure p at depth np calls procedure x at depth nx.
- The code for setting up access links depends upon whether the called procedure is nested within the caller.

$np < nx$

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

$np \geq nx$

- From scoping rules enclosing procedure at the depth 1,2,. ,nx-1 must be same. Follow np-(nx-1) links from the caller, we reach the most recent activation of the procedure that encloses both called and calling procedure

Procedure Parameters

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

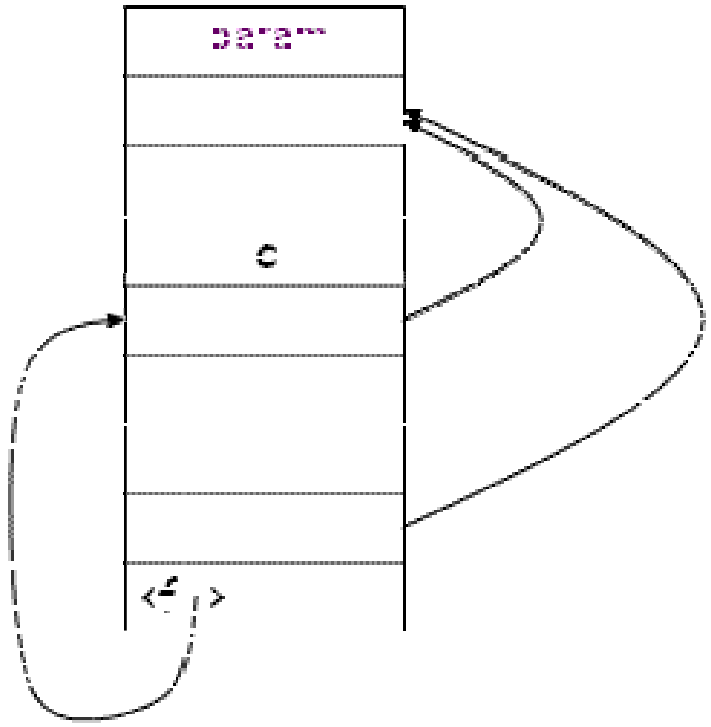
Procedure Parameters

- Scope of mdoes not include procedure b
- within b, call h(2) activates f
- it outputs f(2) which is 2
- 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

Procedure Parameters

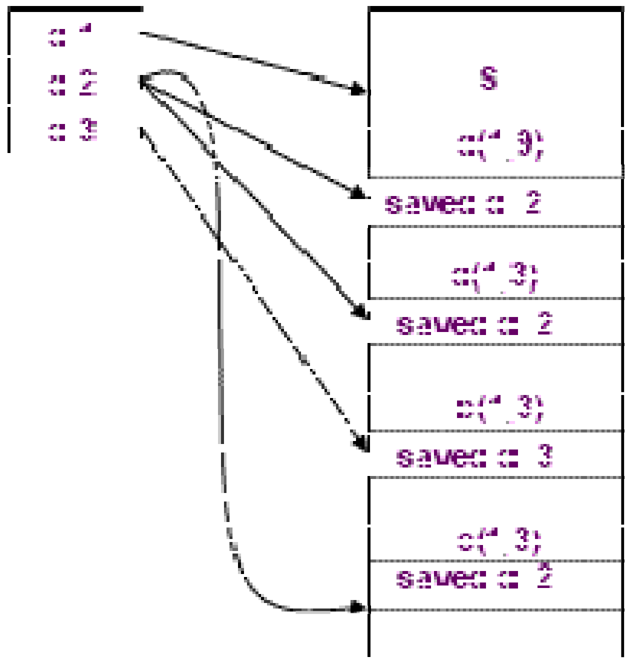
- Lexical scope rules apply even when a nested procedure is passed as a parameter. In the program shown in the previous slide, the scope of declaration of m does not include the body of b. Within the body of b, the call h(2) activates f because the formal h refers to f. Now how to set up the access link for the activation of f? The answer is that a nested procedure that is passed as a parameter must take its access link along with it, as shown in the next slide. When procedure c passes f, it determines an access link for f, just as it would if it were calling f. This link is passed along with f to b. Subsequently, when f is activated from within b, the link is used to set up the access link in the activation record for f.

Procedure Parameters



Displays

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



Dynamic Scope

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

- **In dynamic scope , a new activation inherits the existing bindings of non local names to storage. A non local name a in the called activation refers to the same storage that it did in the calling activation. New bindings are set up for the local names of the called procedure, the names refer to storage in the new activation record.**

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.
```

Consider the example shown to illustrate that the output depends on whether lexical or dynamic scope is used.

Dynamic Scoping: 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
- The outputs under the lexical and the dynamic scoping are as shown. Under dynamic scoping, when show is called in the main program, 0.250 is written because the variable r local to the main program is used. However, when show is called from within small, 0.125 is written because the variable r local to small is used.

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

Implementing Dynamic Scope

We will discuss two approaches to implement dynamic scope. They bear resemblance to the use of access links and displays, respectively, in the implementation of the lexical scope.

1. Deep Access : Dynamic scope results if access links point to the same activation records that control links do. A simple implementation is to dispense with access links and use control links to search into the stack, looking for the first activation record containing storage for the non- local name. The term deep access comes from the fact that search may go deep into the stack. The depth to which the search may go depends on the input of the program and cannot be determined at compile time.
2. Shallow Access : Here the idea is to hold the current value of each name in static memory. When a new activation of a procedure p occurs, a local name n in p takes over the storage for n. The previous value of n is saved in the activation record for p and is restored when the activation of p ends.

Parameter Passing

Call by value

- actual parameters are evaluated and their rvalues 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
- This is, in a sense, the simplest possible method of passing

Parameter Passing

Call by value

- This is, in a sense, the simplest possible method of passing parameters. The actual parameters are evaluated and their r-values are passed to the called procedure. Call-by-value is used in C, and Pascal parameters are usually passed this way. Call-by-Value can be implemented as follows:

1. A formal parameter is treated just like a local name, so the storage for the formals is in the activation record of the called procedure.
2. The caller evaluates the actual parameters and places their r-values in the storage for the formals. A distinguishing feature of call-by-value is that operations on the formal parameters do not affect values in the activation record of the 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 lvalue 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

Call by reference (call by address)

- When the parameters are passed by reference (also known as call-by-address or call-by location), the caller passes to the called procedure a pointer to the storage address of each actual parameter.

1. If an actual parameter is a name or an expression having an l-value, then that l-value itself is passed.
2. However, if the actual parameter is an expression, like a + b or 2, that has no l-value, then the expression is evaluated in a new location, and the address of that location is passed.

- A reference to a formal parameter in the called procedure becomes, in the target code, an indirect reference through the pointer passed to the called procedure.

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

Copy restore (copy-in copy-out, call by value result)

- This is a hybrid form between call-by-value and call-by-reference (also known as copy-in copy-out or value-result).

1. Before control flows to the called procedure, the actual parameters are evaluated. The r-values of the actuals are passed to the called procedure as in call-by-value. In addition, however, the l-values of those actual parameters having l-values are determined before the call.
2. When the control returns, the current r-values of the formal parameters are copied back into the l-values of the actuals, using the l-values computed before the call. Only the actuals having l-values are copied.

Parameter Passing

Call by name (used in Algol)

- names are copied
- local names are different from names of calling procedure

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

Parameter Passing

Call by name (used in Algol)

- This is defined by the copy-rule as used in Algol.

1. The procedure is treated as if it were a macro; that is, its body is substituted for the call in the caller, with the actual parameters literally substituted for the formals. Such a literal substitution is called macro-expansion or inline expansion.
2. The local names of the called procedure are kept distinct from the names of the calling procedure. We can think of each local of the called procedure being systematically renamed into a distinct new name before macro-expansion is done.
3. The actual parameters are surrounded by parentheses if necessary to preserve their integrity.

Run-time Storage

- Run-Time Environment
- Storage Organization
- Storage Allocation Strategies
- Dynamic Storage Allocation

Dynamic Allocation

- Returning the address of a local variable is defined to be a logical error (e.g. in C)
- In a dynamic environment there is no such restriction
  - All variables and activation records must be maintained for as long as there are references to them
    - Callee outlives the caller
  - It is also possible to return pointers to local functions
  - Must deallocate space when procedures and variables are no longer needed (garbage collection)

Dynamic Allocation

- Use a heap to maintain these records
  - Also called free store
  - Heap management is challenging

Language Facility for Dynamic Storage Allocation

- Storage is usually taken from heap
- Allocated data is retained until deallocated
- Allocation can be either explicit or implicit
  - Pascal: explicit allocation and de-allocation by new() and dispose()
  - Lisp: implicit allocation when cons is used, and de- allocation through garbage collection

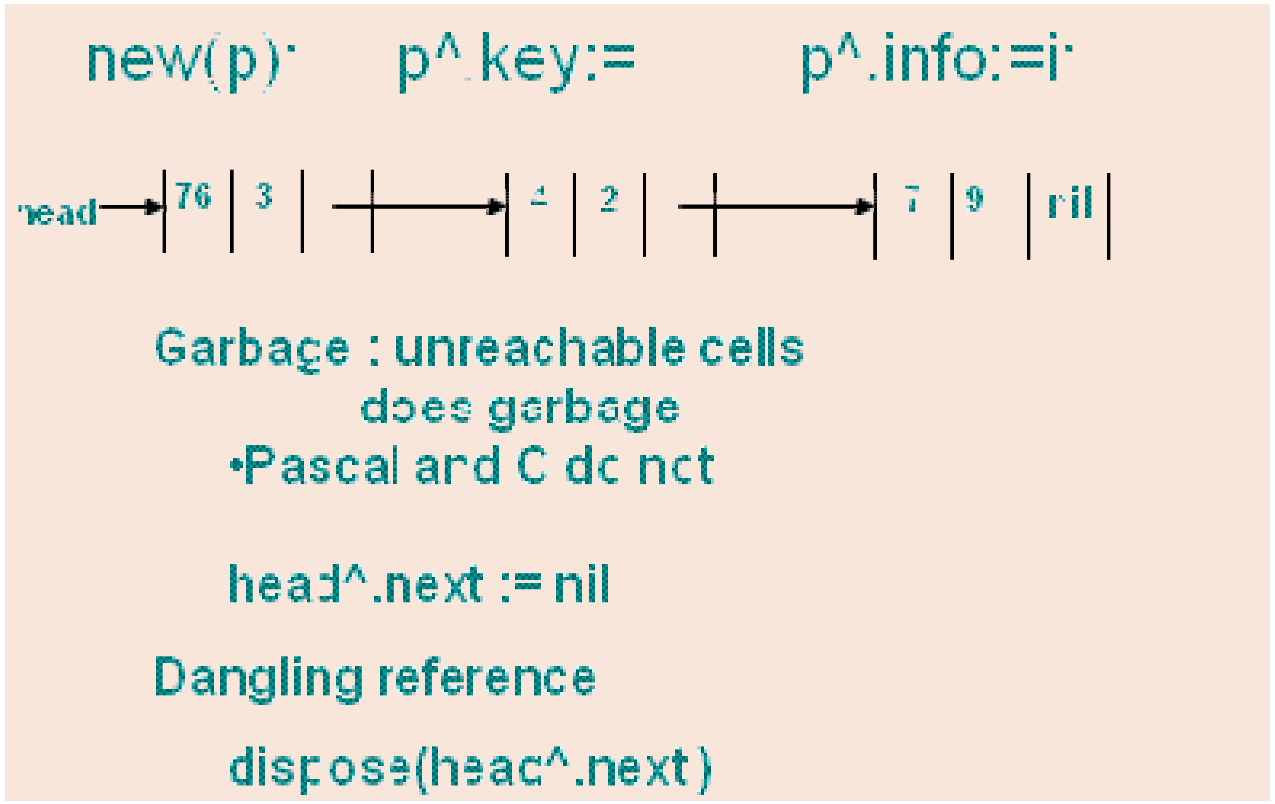
Language Facility for Dynamic Storage Allocation

- Static storage allocation is usually done on the stack, as this is a convenient way to take care of the normal scoping rules, where the most recent values have to be considered, and when the scope ends, their values have to be removed.
- But for dynamic allocation, no such prior information regarding the use of the variables is available. So we need the maximum possible flexibility in this. For this a heap is used.
- For the sake of a more efficient utilization of memory, the stack grows downwards and the heap grows upwards, starting from different ends of the available memory. This makes sure that all available memory is utilized.

Language Facility for Dynamic Storage Allocation

- Pascal allows for explicit allocation and de-allocation of memory. This can be done by using the new() and dispose() functions.
- However, in Lisp, continuous checking is done for free memory.
- When less than 20 percent of the memory is free, then garbage collection is performed.
- In garbage collection, cells that can no longer be accessed are de-allocated. (Storage that has been allocated but can no longer be accessed is called 'garbage'.)

Dynamic Storage Allocation


head
76
3
4
2
7
9
nil

new(p); p^.key:= p^.info:=i;

Garbage : unreachable cells

does garbage

•Pascal and C do not

head^.next := nil

Dangling reference

dispose(head^.next)

Dynamic Storage Allocation

Generally languages like Lisp and ML which do not allow for explicit de-allocation of memory do garbage collection. A reference to a pointer that is no longer valid is called a 'dangling reference'. For example, consider this C code:

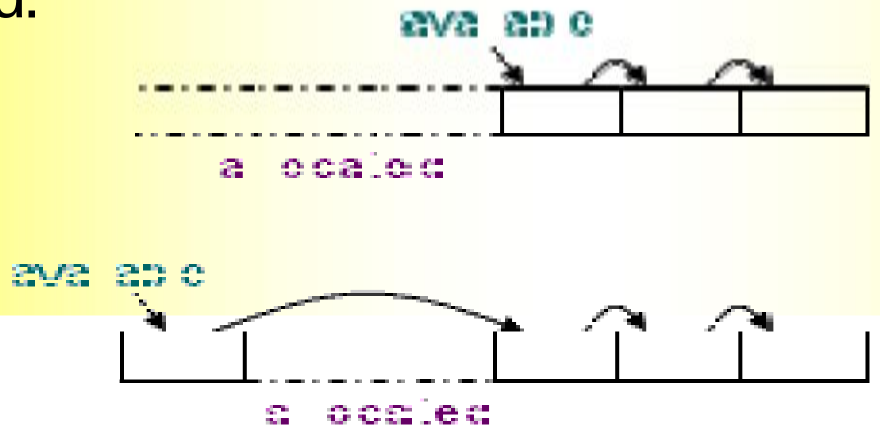
```
int main (void)
{
int* a=fun();
}
int* fun()
{
int a=3;
int* b=&a;
return b;
}
```

Here, the pointer returned by fun() no longer points to a valid address in memory as the activation of fun() has ended. This kind of situation is called a 'dangling reference'. In case of explicit allocation it is more likely to happen as the user can de-allocate any part of memory, even something that has to a pointer pointing to a valid piece of memory.

Dynamic Storage Allocation

Explicit Allocation of Fixed Sized Blocks

- Link the blocks in a list
- Allocation and de-allocation can be done with very little overhead
- The simplest form of dynamic allocation involves blocks of a fixed size.
- By linking the blocks in a list, as shown in the figure, allocation and de-allocation can be done quickly with little or no storage overhead.



Dynamic Storage Allocation

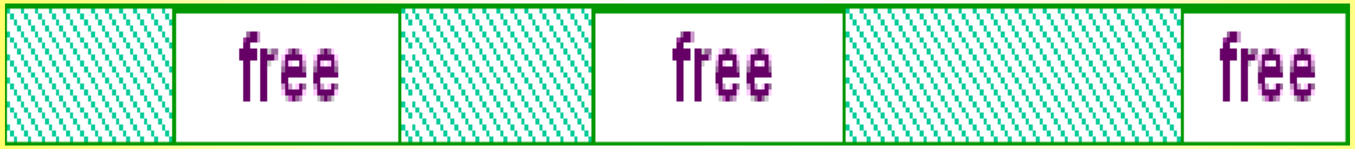
Explicit Allocation of Fixed Sized Blocks .

- blocks are drawn from contiguous area of storage
- An area of each block is used as pointer to the next block
- A pointer available points to the first block
- Allocation means removing a block from the available list
- De-allocation means putting the block in the available list
- Compiler routines need not know the type of objects to be held in the blocks
- Each block is treated as a variant record

Dynamic Storage Allocation

Explicit Allocation of Variable Size Blocks

- Storage can become fragmented
- Situation may arise
  - If program allocates five blocks
  - then de-allocates second and fourth block



- Fragmentation is of no consequence if blocks are of fixed size
- Blocks can not be allocated even if space is available

Dynamic Storage Allocation

First Fit Method

When a block of size sis to be allocated

-search first free block of size f ≥ s

-sub divide into two blocks of size sand f-s

-time overhead for searching a free block

When a block is de-allocated

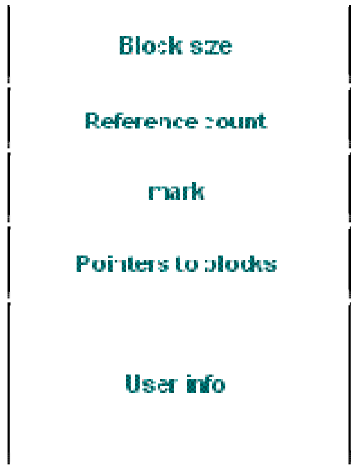
-check if it is next to a free block

-combine with the free block to create a larger free block

Dynamic Storage Allocation

Implicit De-allocation

- Requires co-operation between user program and run time system
- Runtime system needs to know when a block is no longer in use
- Implemented by fixing the format of storage blocks
- Implicit deallocation requires cooperation between the user program and run time package,



THANK YOU!!!!!!!!!!

My Blog : anandgharu.wordpress.com