Run-Time Environments

Run-Time Support Package

- Consists of routines which execute the generated target code
- Handles allocation and deallocation of data objects
- Activates procedures when called
- May be multiple active instances of a single function if recursion is allowed
- Semantics of procedures heavily influence design of run-time support packages

Procedures

- Procedure Definition:
- A declarations of procedure
- Associate an identifier (procedure name) with a statement (procedure body)
- A procedure that returns a value is sometimes referred to as a function
- Textbook also treats full program as a procedure
- Procedure calls pass arguments (actual parameters) to parameters (formal parameters)

Flow of Control

- Control flows sequentially
- Execution of a program consists of a sequence of steps
- At each step, control is at some specific point in the program
- Execution of a procedure
- Starts at the beginning of the procedure
- Eventually returns control to the point immediately following procedure call

Procedure Activation and Lifetime

- A procedure is *activated* when called
- The *lifetime* of an activation of a procedure is the sequence of steps between the first and last steps in the execution of the procedure body
- A procedure is *recursive* if a new activation can begin before an earlier activation of the same procedure has ended
- Can be depicted using trees

Example Program

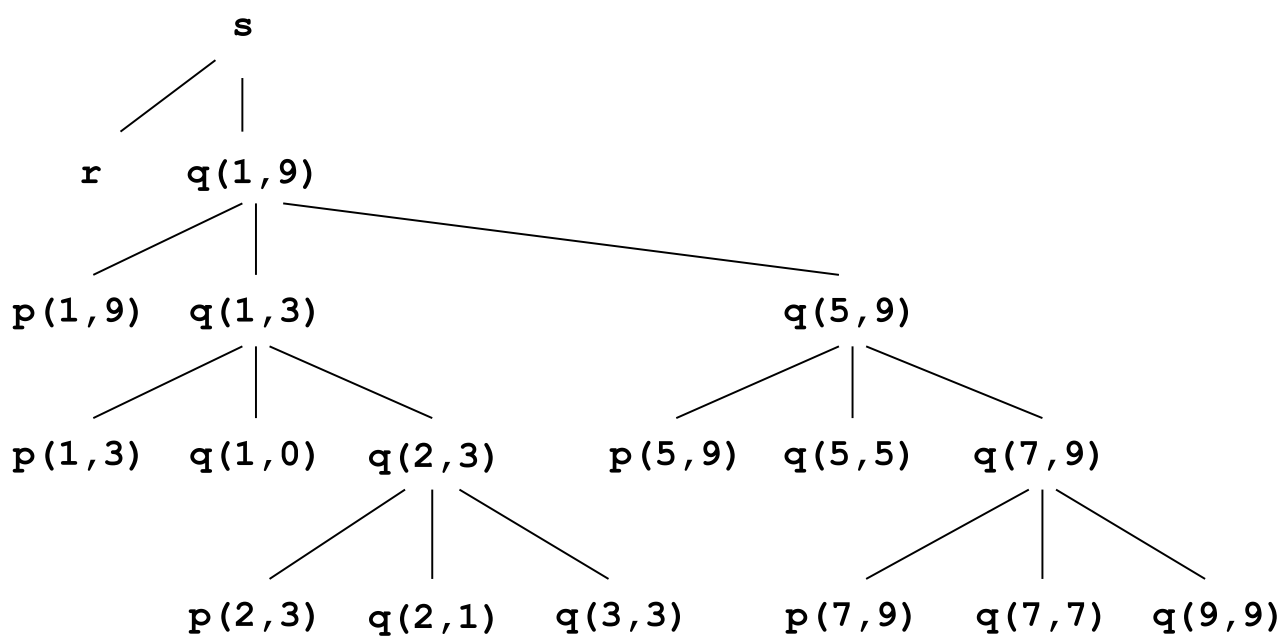
```
program sort(input, output);
        var a : array [0..10] of integer;
        procedure readarray;
        var i : integer;
        begin
                for i := 1 to 9 read(a[i])
        endl

        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
        a[0]:= -9999; a[10] := 999;
        readarray;
        quicksort(1,9);
end.
```

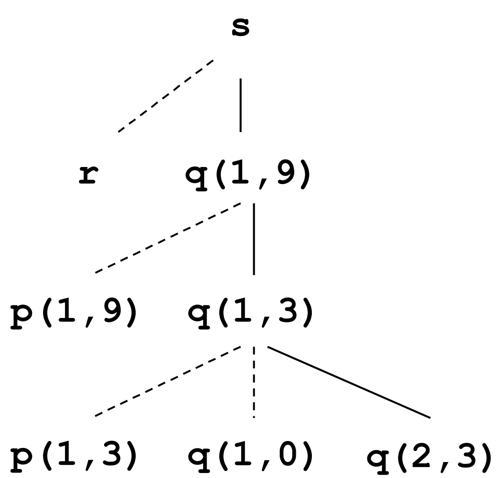
Activation Trees

- Each node represents an activation of a procedure
- The root represents the activation of the program
- The node for **`a`** is the parent of the node for **`b`** if and only if control flows from activation **`a`** to **`b`**
- The node for **`a`** is to the left of node **`b`** if and only if the lifetime of **`a`** occurs before the lifetime of **`b`**
- Often convenient to talk of control being "at a node"

Activation Tree

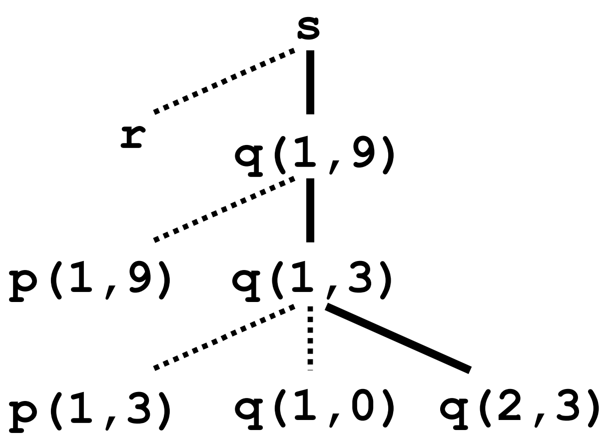
```
Execution begins...
enter readarray
leave readarray
enter quicksort(1,9)
enter partition(1,9)
leave partition(1,9)
enter quicksort(1,3)
     ...
leave quicksort(1,3)
enter quicksort(5,9)
     ...
leave quicksort(5,9)
leave quicksort(1,9)
Execution terminated.
```





Control Stack

Activation tree:



Control stack:

s
q(1,9)
q(1,3)
q(2,3)

Activations:

```
begin sort
 enter readarray
 leave readarray
 enter quicksort(1,9)
  enter partition(1,9)
  leave partition(1,9)
  enter quicksort(1,3)
   enter partition(1,3)
   leave partition(1,3)
   enter quicksort(1,0)
   leave quicksort(1,0)
   enter quicksort(2,3)
   …
```

- Flow of control in a program corresponds to a depth-first traversal of activation tree
- A stack called a control stack can keep track of live procedure activations
- A node is pushed as activation of procedure begins
- Node is popped when activation of procedure ends

Scope

- The scope of a declaration is the portion of the program to which the declaration applies
- Sometimes convenient to speak of scope of name itself as opposed to the declaration
- A declaration that applies only within a procedure is said to be local to the procedure
- The same name can be used multiple times in a program with different scopes
- When a name is encountered:
  - The scope rules of a language determine which declaration of the name applies
  - At compile time, the symbol table can be used to determine the appropriate declaration

Scope Rules

- *Environment* determines name-to-object bindings: which objects are in *scope*?

```
program prg;
  var y : real;
function x(a : real) : real;
  begin … end;
procedure p;
  var x : integer;
  begin
    x := 1;
    …
  end;
begin
  y := x(0.0);
  …
end.
```

Variable **x** locally declared in **p**

A function **x**

Bindings of Names

- Informally, a "data object" corresponds to a storage location that can hold values
- Even if a name is declared only once, it can denote different data objects at run time
- An environment maps a name to a storage location (l-value)
- A state maps a storage location to a value (r-value)
- If an environment associates storage location **`s`** with name **`x`**:
  - We say that **`x`** is bound to **`s`**
  - The association itself is referred to as a binding of **`x`**
- A binding is the dynamic counterpart of a declaration

Name Binding

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

```
var i;
…
i := 0;
…
i := i + 1;
```

Static and Dynamic Notions of Bindings

Static Notion	*Dynamic Notion*
Definition of a procedure	Activations of the procedure
Declaration of a name	Bindings of the name
Scope of a declaration	Lifetime of a binding

Factors Influencing Run-Time Environment

- May procedures be recursive?
- What happens to the values of local names when control returns from an activation of a procedure?
- May a procedure refer to nonlocal names?
- How are parameters passed when a procedure is called?
- May procedures be passed as parameters?
- May procedures be returned as results?
- May storage be allocated dynamically under program control?
- Must storage be deallocated explicitly?

Run-Time Memory

- Run-time memory is divided into code and data areas
- The data areas generally include static data, a stack, and a heap
- Static data consists of data that is known at compile-time, e.g. globals
- The stack stores activation records and locals
- The heap stores all other information, e.g. dynamically allocated memory

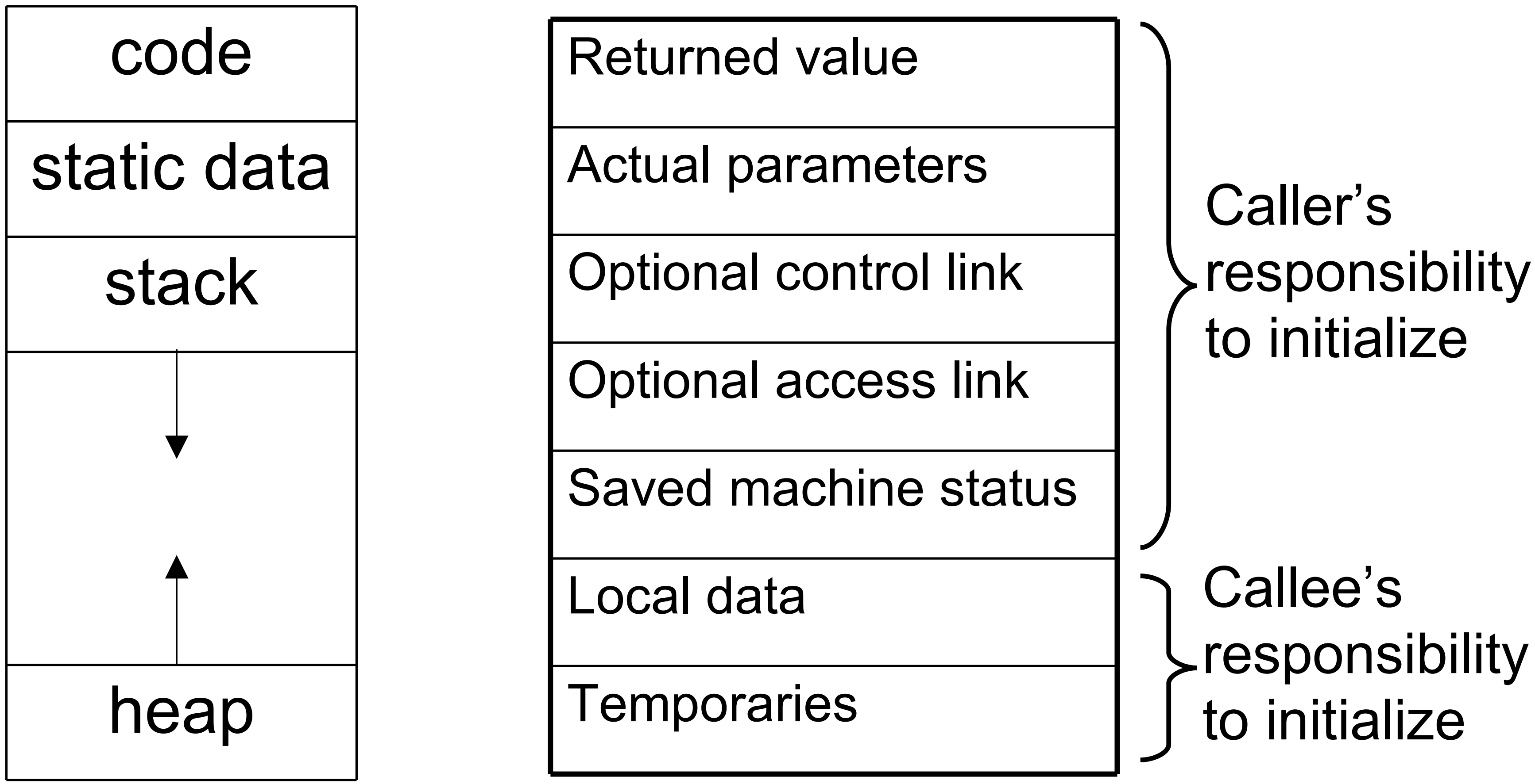
Activation Records

- *Activation records* (subroutine frames) hold the state of a subroutine
- Each activation record generally resides in a contiguous block of memory
- For many languages (e.g. Pascal, C), the activation record is:
  - Pushed to top of run-time stack when procedure is called
  - Pop off of stack when control returns to caller
- Activation records consist of several fields
- *Calling sequences* are code statements to create activations records on the stack and enter data in them
  - Caller's calling sequence enters actual arguments, control link, access link, and saved machine state
  - Callee's calling sequence initializes local data
  - Callee's return sequence enters return value
  - Caller's return sequence removes activation record

Fields of Activation Records

- A field for temporary values such as those arising in the evaluation of expressions
- A field for local data
- A field for saved machine status, e.g. the program counter and machine registers that need to be restored
- An optional field for an access link to refer to nonlocal data held in other activation records
- An optional field for a control link pointing to the activation record of the caller
- A field for actual parameters (i.e. arguments supplied by the calling procedure)
- A field for the return value

Activation Records
(Subroutine Frames)



Compile-Time Layout of Local Data

- The amount of storage needed for a name is determined from its type
- The field of an activation record for local data is laid out as declarations in a procedure
- A offset keeps track of how much memory has been allocated for previous declarations
- This offset determines a relative address from some base, e.g. the start of the activation record
- Some constraints may be imposed by the target machine, e.g. integers may have to be aligned

Data Layouts Used by Two C Compilers

Type	Size (bits)		Alignment (bits)	
	Machine 1	Machine 2	Machine 1	Machine 2
`char`	8	8	8	64*
`short`	16	24	16	64
`int`	32	48	32	64
`long`	32	64	32	64
`float`	32	64	32	64
`double`	64	128	32	64
`char*`	32	30	32	64
other ptrs.	32	24	32	64
structures	≥ 8	≥ 64	32	64

*8 bits in a character array

Storage-Allocation Strategies

- Static allocation lays out storage for all data objects at compile time
- Stack allocation manages run-time storage as a stack
- Heap allocation allocates and deallocates storage as needed from a heap
- Any of one these strategies can be used to manage activation records

Static Allocation (2)

- From type of a name, compiler determines the amount of storage to set aside
- The address consists of an offset from the end of the activation record for procedure
- Compiler must decide where activation records go relative to target code
- Once decisions are made, all storage is fixed, all addresses are known

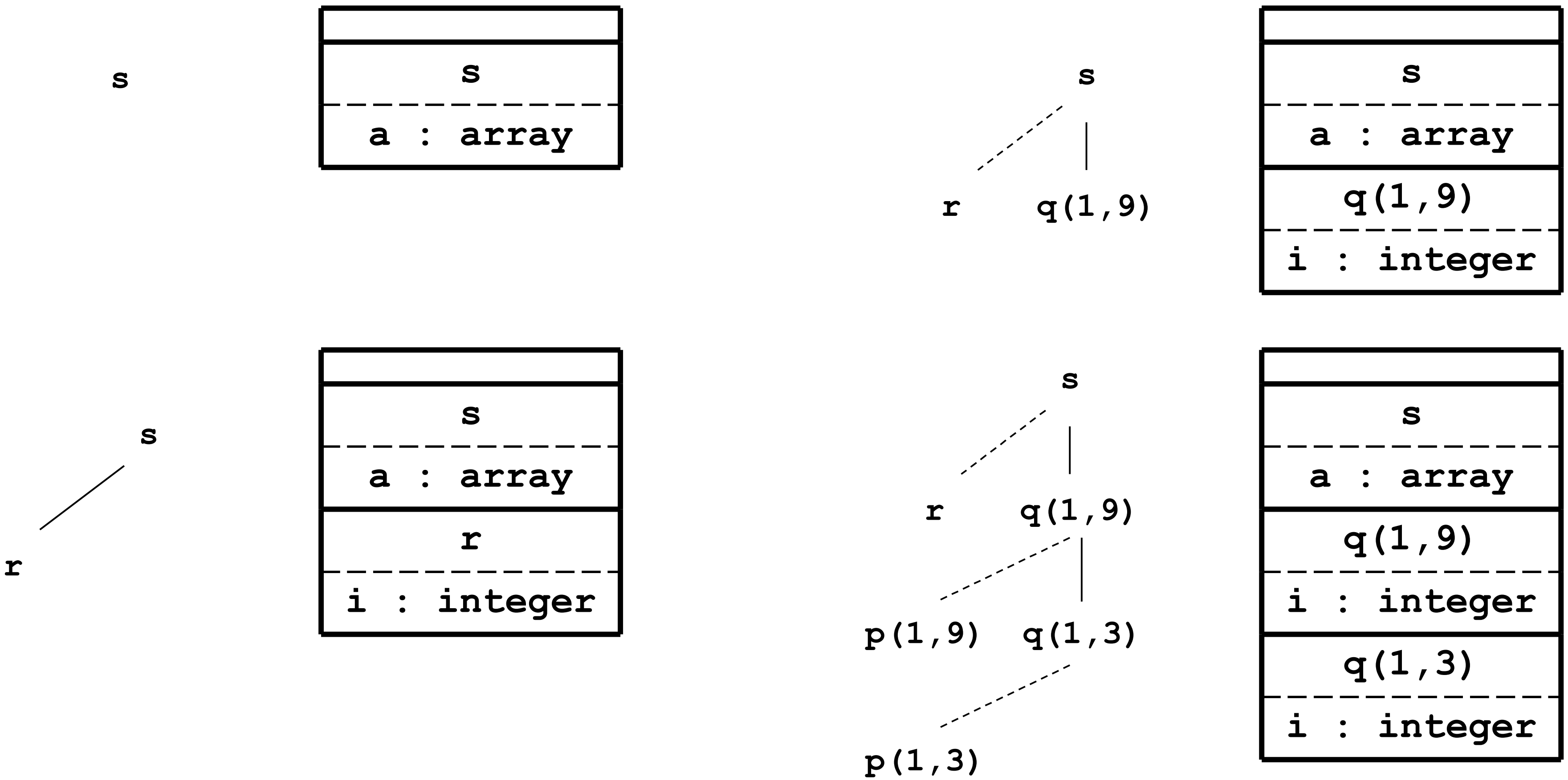
Limitations of Static Allocation

- The size of all data objects must be known at compile time
- Recursive procedures are restricted since all activations use the same bindings
- Data structures cannot be created dynamically

Stack Allocation

- Based on the idea of a control stack
- Activation records are pushed and popped as activation begins and ends, respectively
- Storage for locals in each call of a procedure is contained in the activation record
- Locals are thus bound to fresh storage in each activation
- The values of locals are deleted when the activation ends

Stack of Activation Records



Calling Sequences (1)

- Procedure calls are implemented by generating calling sequences in target code
- A call sequence allocates an activation record and enters information in fields
- A return sequence restores the state of the machine so calling procedure can continue
- Calling sequences and activation records differ even for implementations of same language
- Code in a calling sequence is often divided between calling procedure and called procedure
- No exact division of run-time tasks between caller and callee

Calling Sequences (2)

- Principle for designing activation records: fields of fixed size placed in the middle
- Fixed size fields: access link, control link, machine status information
  - Links are optional, decision as to whether or not to use is part of compiler design
  - If same machine-status information saved for each activation, same code can do saving and restoring
  - Programs such as debuggers will have an easier time deciphering stack contents when an error occurs

Temporaries in Activation Records

- Size of field for temporaries eventually fixed as compile time
- May not be known to front end, since careful optimization may reduce number of temporaries
- As far as front end is concerned, the size of this field is unknown
  - For this reason, temporaries generally placed at end of activation record
  - Offsets of locals relative to fields in the middle are therefore not affected

Parameters in Activation Records

- Each call has its own actual parameters (arguments)
- These arguments are communicated to the activation record of the called procedure
- Various schemes exist to pass parameters (discussed more later)
- In run-time stack, the activation record of the caller is just below that of the callee
- Advantages of placing fields for parameters and return value next to activation record of caller
  - Caller can access using offset from the end of its own activation record
  - No need for caller to know about local data or temporaries of the called procedure

Calling Sequence Possibility

- The caller evaluates arguments and places them on stack in new activation record
- The caller stores a return address into new activation record
- The callee saves the old value of **`fp`**, register values and other status information
- The callee initializes its own local data and begins execution
- This scheme allows for the number of arguments of the called procedure to depend on the call

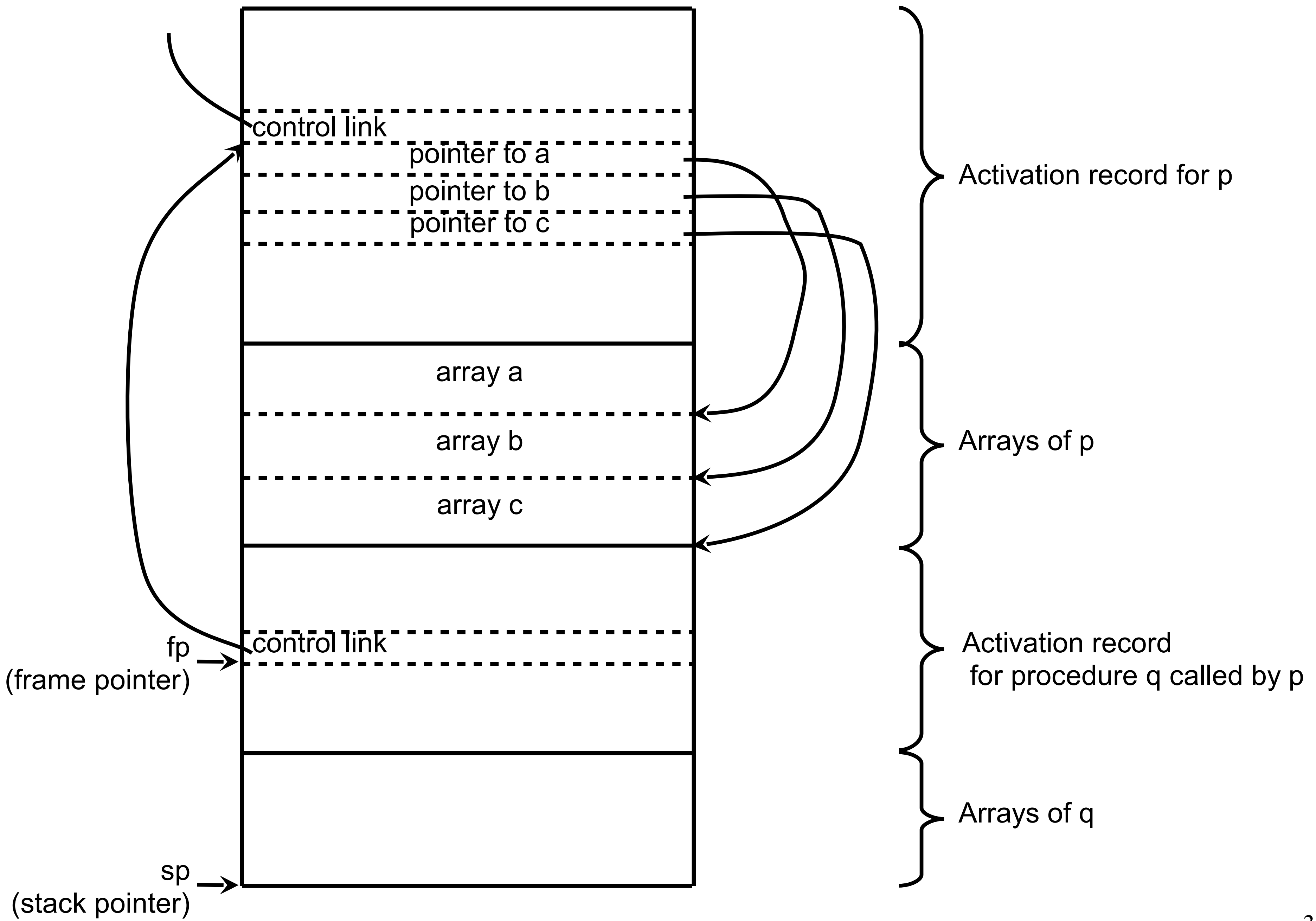
Return Sequence Possibility

- The callee places a return value next to the activation record of the caller
- Using information in the status field:
  - The callee restores **`fp`** and other registers
  - The callee executes a branch to the appropriate return address in the caller's code
- The caller may copy the returned value into its own activation record

Handling Variable-Length Data

- Some languages allow procedures to accept variable-length parameters
- Such data does not get stored in the activation record for the procedure
- Example, variable-length arrays for procedure `p`:
  - A pointer to the start of each array appears in the activation record for `p`
  - The relative addresses of these pointers are known at compile time so target code can access the arrays
  - Arrays appear after activation record of `p`

Variable-Length Data Example



Accessing the Stack

- Accessing the stack is done through two pointers, **`sp`** and **`fp`**
- The pointer **`sp:`**
  - Points to the actual top of the stack
  - Denotes location where next activation record will be placed
- The pointer **`fp`**:
  - Used to locate local data
  - Often points to end of machine-status field
- The control link of each activation record points to the previous value of **`fp`**
- Code to reposition **`sp`** and **`fp`** when a procedure returns can be generated at compile time

Dangling References

- A dangling reference occurs when there is a reference to storage that has been deallocated
- It is a logical error to use dangling references

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

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

Limits of Stack Allocation

- The values of local names can not be retained when an activation ends
- A called activation can never outlive the caller
  - Will always be true if activation trees correctly depict flow of control for the language
  - If not true, storage can not be organized as a stack (last-in, first-out)

Heap Allocation

- Heap allocation parcels out pieces of contiguous storage as needed
- Can be used for activation records or other data objects
- Pieces may be deallocated in any order
- Heap will therefore consist of alternate areas that are free and in use
- If used for activation records:
  - Can not assume that activation record of called procedure follows activation record of caller
  - May be free space in between current activation records; up to heap manager to make use of space

Access to Nonlocal Names

- The scope rules of a language determine the treatment of references to nonlocal names
- One common rule is the lexical-scope rule (a.k.a. the static scope rule)
  - The declaration that applies to a name is determined by examining program text alone
  - Used for most common languages (e.g. C, Pascal)
  - Often a "most closely nested" stipulation goes along with this strategy
- An alternative rule is the dynamic-scope rule
  - Declaration applicable to a name is determines at run-time by considering current activations
  - Used by languages including Lisp and APL

Blocks

- A block is a statement containing its own local declarations
- In C, a block (compound statement) has syntax: **`{declarations statements}`**
- Delimiters mark the beginning and end of a block
  - Delimiters ensure that two blocks are either independent or one is nested inside the other
  - This property is referred to as block structure

The Most Closely Nested Rule

- The scope of a declaration in block **`B`** includes **`B`** (minus holes)
- If a name, **`x`**, is not declared in block **`B`**, and an occurrence of **`x`** is in **`B`**, then:
  - This **`x`** is in the scope of a declaration of **`x`** in an enclosing block **`B'`**
  - **`B'`** must have the following two properties:
    - **`B'`** has a declaration of **`x`**
    - **`B'`** is more closely nested around **`B`** than any other block with a declaration of **`x`**

Scope in C Example (1)

```
int main()
{
        int a = 0;
        int b = 0;
        {
            int b = 1;
            {
                int a = 2;
    B1    B2    printf("%d %d\n", a, b);
B0          }
            {
                int b = 3;
          B3    printf("%d %d\n", a, b);
            }
            printf("%d %d\n", a, b);
        }
        printf("%d %d\n", a, b);
};
```

Deklaration	Scope
`int a = 0;`	B_0 - B_2
`int b = 0;`	B_0 - B_1
`int b = 1;`	B_1 - B_3
`int a = 2;`	B_2
`int b = 3;`	B_3

a_0
b_0
b_1
a_2, b_3

Scope in C Example (2)

- Each declaration initializes a name to the number of the block in which it is declared
- The scope of the declaration of **`b`** in $\mathbf{B_0}$ does not include $\mathbf{B_1}$
  - This is because redeclared in $\mathbf{B_1}$
  - The scope of the declaration of **`b`** in $\mathbf{B_0}$ is therefore $\mathbf{B_0} - \mathbf{B_1}$
  - The gap is referred to as a hole

Implementing Block Structure (1)

- Block structure can be implemented using stack allocation
- Since the scope of a declaration does not extend outside the block in which it appears:
  - Space for declared name is allocated when block is entered, deallocated when control leaves block
  - This view treats block as a "parameterless procedure"
    - Called only from the point just before the block
    - Returning only to the point just after the block
- This can be a bit more confusing depending on the language's rules for `goto` statements

Implementing Block Structure (2)

- An alternative is to allocate storage for complete procedures at one time
- If there are blocks within a procedure:
  - Allowances are made for storage needed for declarations within these blocks
  - Some times two locals can share the same storage (e.g. **`a`** in $\mathbf{B}_2$ and **`b`** in $\mathbf{B}_3$ in example)

Implementing Block Structure (3)

- In the absence of variable-length data:
  - Maximum storage needed during execution of a block can be determined at compile time
  - Variable-length data can be handled using pointers (as with activation records)
- Common to conservatively assume that all control paths in a block can be taken

Scope without Nested Procedures

- In the absence of nested procedures:
  - Lexical scope can be implemented with the stack-allocation strategy directly
  - Storage for all names declared outside any procedure can be allocated statically
  - Any name must be local to the current activation or else in a known static address
- Makes it easier to pass procedures to functions or return procedures as results

Non-Nested Procedures Example

```
program pass(input, output);
  var m : integer;

  function f(n : integer) : integer;
    begin f := m + n end; {f}

  function g(n : integer) : integer;
    begin g := m + n end; {g}

  procedure b(function h(n : integer) : integer);
    begin write(h(2)) end; {b}

  begin
    m := 0;
    b(f); b(g); writeln
  end.
```

Scope with Nested Procedures

- Nesting Depth
  - Let the name of the main program be at nesting depth 1
  - Add 1 to the nesting depth when move from any enclosing to an enclosed procedure
  - With the occurrence of any name, associate the nesting depth of the procedure in which it is declared
- Access Links
  - An access link is an extra pointer added to each activation record
  - For any procedure **`p`**:
    - Let **`q`** be the procedure in which **`p`** is immediately nested in the source text
    - The access link in an activation record for **`p`** will point to the record for the most recent activation of **`q`**

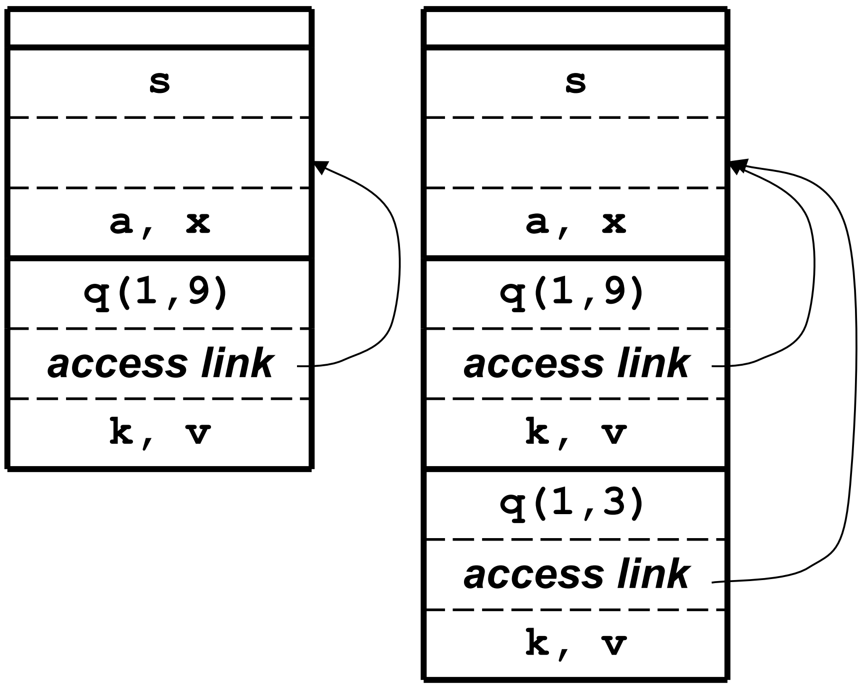
Nested Procedures Example (1)

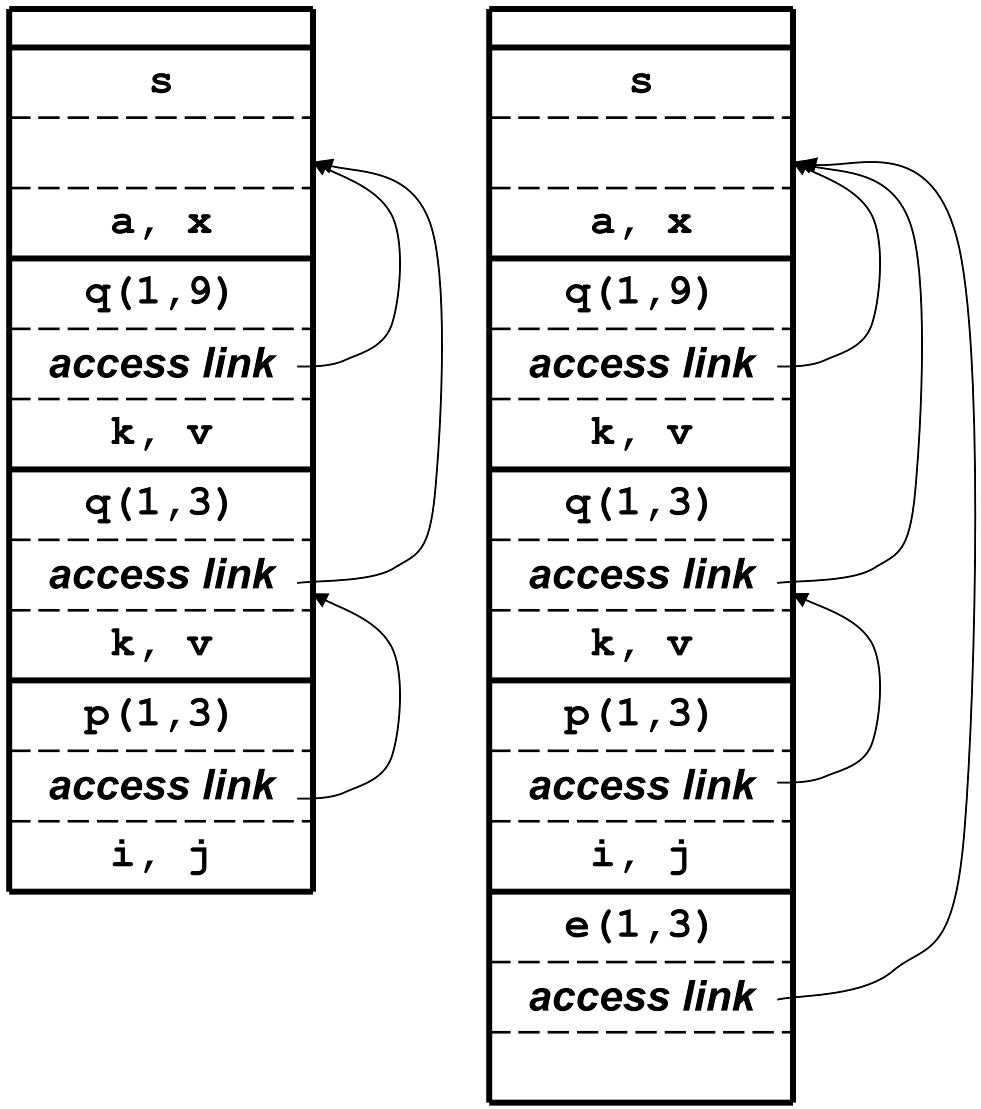
```
program sort(input, output);
    var a: array [0..10] of integer;
        x: integer;

    procedure readarray;
        var i : integer;
        begin ... a ... end { readarray } ;

    procedure exchange( i, j: integer);
        begin
            x : = a[i]; a[i] := a[j]; a[j] := x
        end { exchange } ;

    procedure quicksort( m, n: integer);
        var k, v : integer;

        function partition( y, z: integer): integer;
            var i, j: integer;
            begin ... a ...
                  ... v ...
                  ... exchange(i,j); ...
            end { partition }

        begin ... end { quicksort };

begin ... end { sort };
```

Nested Procedures Example (2)

- The declaration of **`quicksort`** is at nesting depth 2
- The declaration of **`partition`** is at nesting depth 3
- The names of **`a`**, **`v`**, and **`i`** in **`partition`** have nesting depths 1, 2, and 3
- The activation record for **`quicksort`** will always point to the record for **`sort`**
- The record for **`partition`** will always point to that of the most recent activation of **`quicksort`**

Nested Procedures Example (3)





The access link points to the activation record of the static parent procedure:
s is parent of **r**, **e**, and **q**
q is parent of **p**

Algorithm for Finding a Nonlocal

- Suppose procedure **p** at nesting depth n_p refers to nonlocal **a** with nesting depth $n_a < n_p$
- If control is in **p**, activation record for **p** must be at top of stack
- First follow $n_p - n_a$ access links (computed at compile time)
  - Easy if access links point to access links
  - Brings us to activation record for procedure that **a** is local to
- Storage for **a** at fixed offset to some position in record (fixed position could be access link)

Setting Up Access Links

- Code to set up access links is part of calling sequence
- Suppose procedure **`p`** with nesting depth n_p calls procedure **`x`** with nesting depth n_x
- If $n_p < n_x$
  - Procedure **`x`** must be declared within **`p`**
  - Access link of **`x`** points to access link of **`p`**
- If $n_p >= n_x$
  - There must be some common enclosing procedure
  - Following $n_p - n_x + 1$ access from **`p`** brings us to activation record of common ancestor
  - This is record to which access link for **`x`** must point

Passing Procedures as Parameters (1)

- Lexical scope rules apply when a nested procedure is passed as a parameter
- The access link must be passed along with procedure parameter
- Calling procedure must determine access link for passed procedure
- When procedure parameter is activated, access link is used for activation record

Passing Procedures as Parameters (2)

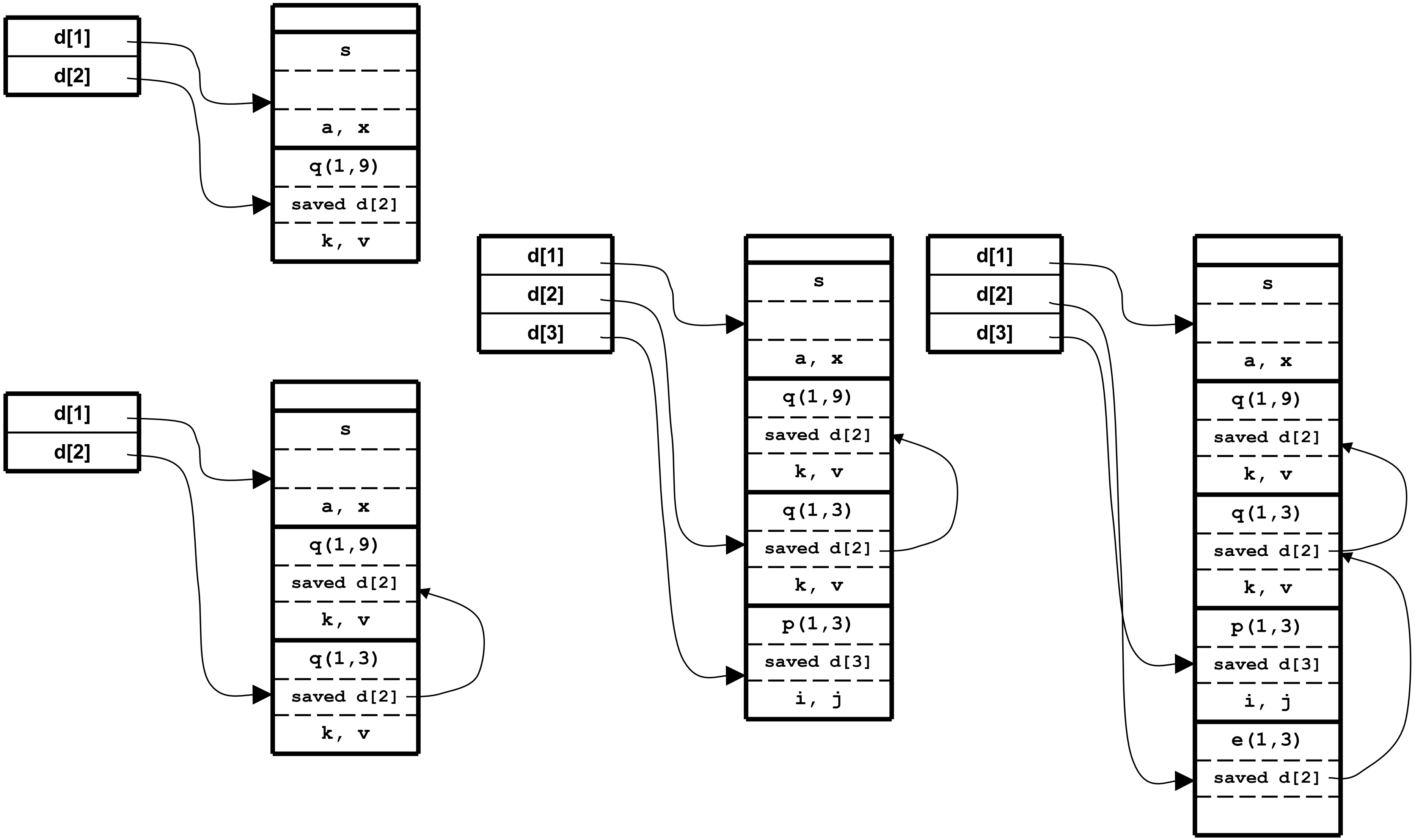
```
Program param(input, output);

        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 passed as a parameter must take its access link along with it.

Displays

- A display is an array of pointers to activation records
  - Maintained so that any nonlocal **`a`** at nesting depth **`i`** is in activation record pointed to by display **`d[i]`**
  - Faster than using access links since only need to access element of **`d`** and follow one pointer
  - Display is updated as part of call and return sequence
- Simple approach for maintaining the display
  - Use access links in addition to the display
  - Whenever an access link to an activation record at nesting depth **`n`** is followed, **`d[n]`** is updated
- A better method exists if no procedure parameters
  - Save the value of **`d[i]`** in every new activation record
  - Set **`d[i]`** to point to the new activation record
  - Restore **`d[i]`** just before activation ends

Display Example



Dynamic Scope

- Under dynamic scope:
  - A new activation inherits the existing bindings of nonlocal names to storage
  - A nonlocal `a` in the called activation refers to the same storage as in the calling activation
- The output of a program may depend on whether lexical or dynamic scope is used

Dynamic Scope Example

```
program dynamic(input, output);
        var r : real;

        procedure show;
                begin write (r :5:3 ) end;

        procedure small;
                var r : real;
                begin r := 0.125; show end;

begin
        r := 0.25;
        show; small; writeln;
        show; small; writeln;
end.
```

Implementing Dynamic Scope

- Deep Access
  - Dispense with access links
  - Use control links to search stack for first activation record containing storage for nonlocal name
  - Search may go deep into stack
  - Depth of search depends on input, can not be determined at compile time
- Shallow Access
  - Hold the current value of each name in statically allocated storage
  - When a new activation of procedure `p` occurs, a local name `n` in `p` takes over storage statically allocated for `n`
  - Previous value of `n` can be saved in activation record for `p`, must be restored when activation of `p` ends

Parameter Passing

- When one procedure calls another, communication is done through:
  - nonlocal names
  - parameters of the called procedure
- Several methods exist for associating actual and formal parameters
  - call-by-value
  - call-by-reference
  - copy-restore
  - call-by-name

L-values and R-values

- Consider an assignment, e.g. **`a[i] := a[j]`**
- The term l-value refers to the storage represented by an expression
- The term r-value refers to the value contained in such storage
- If an expression appears to the left of an assignment symbol, it represents an l-value
- If an expression appears to the right of an assignment symbol, it represents an r-value

Call-by-Value

- The simplest method of passing parameters
- The actual parameters are evaluated and their r-values are passed to the called procedure
- Used in C and sometimes Pascal
- A formal parameter is treated like a local name, so storage for it is in the activation record
- The caller evaluates the actual parameters and places the r-values in the storage for the formals

Using Call-by-Value

- Operations on formal parameters do not affect values in activation record of caller
- A procedure called by value can affect its caller in two ways:
  - Using nonlocals
  - Through pointers that are explicitly passed as value

Call-by-Value Example

```
#include <stdio.h>

void swap(int *, int *);

int main(void) {
  int a = 1, b = 2;

  swap(&a, &b);
  printf("a is now %d, b is now %d\n", a, b);
}

void swap(int *x, int *y) {
  int temp;

  temp = *x; *x = *y; *y = temp;
}
```

Call-by-Reference

- Also known as call-by-address and call-by-location
- The caller passes a pointer to the storage address of each parameter
- If an actual parameter is a name or expression with an l-value, the l-value itself is passed
- If the actual parameter is an expression without an l-value:
  - The expression is evaluated in a new location
  - The address of that location is passed

Call-by-Reference Example

```
program reference(input, output);
var a, b : integer;

procedure swap(var x, y: integer);
  var temp : integer;
  begin
    temp := x;
    x := y;
    y := temp
  end;

begin
  a := 1; b := 2;
  swap(a, b);
  writeln('a = ', a); writeln('b =', b)
end.
```

Copy-Restore

- Also known as copy-restore linkage, copy-in copy-out, or value-result
- A hybrid between call-by-value and call-by-reference
- The calling sequence:
  - The actual parameters are evaluated before a call
  - The r-values of the actuals are passed to the called procedure as in call-by-value
  - In addition, the l-values of the actual parameters having l-values are determined before the call
- The return sequence:
  - When control returns, the current r-values of the actuals are copied back into the l-values of the actuals
  - The l-values computed before the call are used (only actuals having l-values are copied)

Copy-Restore Example

```
program copyout(input, output);
var a : integer;

procedure unsafe(var x : integer);
  begin
    x := 2;
    a := 0
  end;

begin
  a : = 1;
  unsafe(a);
  writeln(a)
end.
```

Call-by-Name

- Traditionally defined by the "copy-rule" of Algol
- The procedure is treated as if it were a macro
  - The body is substituted for the call in the caller
  - Actual parameters are literally substituted for the formals
  - Such a literal expansion is called macro-expansion of in-line expansion
  - Local names of the called procedure are kept distinct from names of the calling procedure
  - The actual parameters are surrounded by parentheses if necessary to preserve their integrity
  - Implementations use a form of in-line code expansion (*thunk*) to evaluate parameters
- Supposedly, there is no way to write a correct version of swap using call-by-name!

```
swap(i,a[i])

temp:=i;
i:=a[i];
a[i]=temp;
```

Call-by-Name

- "Whereas Europeans generally pronounce his name the right way ('Nick-louse Veert'), Americans invariably mangle it into 'Nickel's Worth.' This is to say that Europeans call him by name, but Americans call him by value."
  - Adriaan van Wijngaarden introducing Niklaus Wirth at the IFIP Congress (1965).

Dynamic Storage Allocation

- Many languages provide facilities for dynamic allocation under program control
- Storage for such data is generally taken from a heap
- The allocation can be explicit or implicit
- Allocated data is often retained until it is explicitly deallocated
- Deallocated memory can be reused

Dynamic Allocation Example

```
program table(input, output);

type link = ↑ cell;
cell = record
  key, info : integer;
  next : link
end;
var head : link;

procedure insert (k, i : integer);
var p : link;
begin
  new(p); p↑.key := k; p↑.info := i;
  p↑.next := head; head := p
end;

begin
  head := nil;
  insert(7,1); insert(4,2); insert(76,3);

  writeln(head↑.key, head↑.info);
  writeln(head↑.next↑.key, head↑.next↑.info);
  writeln(head↑.next↑.next↑.key,
          head↑.next↑.next↑.info);
end.
```

Explicit Allocation (1)

- The simplest form of dynamic allocation involves fixed sized blocks
- Using a linked list of blocks requires little overhead
  - A portion of each block will link to the next block
  - A pointer to the first available block is also maintained
  - Allocation consists of taking a block off the list
  - Deallocation consists of putting a block back on the list
  - The compiler does not need to know the type of object that will be held in each block

Explicit Allocation (2)

- When variable-sized blocks are allocated, storage can become fragmented
- The heap may consist of alternate blocks that are free and in use
- Allocation and deallocation must be careful in dealing with fragmentation issues
- With a simple scheme, a program can not allocate a block larger than the largest free block
- When a block is deallocated, if it is next to a free block it is combined with the free block (block coalescing)

First-fit, Best-fit, Next-fit ...

- First-fit
  - To allocate the requested memory in the first hole in which it fits (fast, but lots of small holes)
- Best-fit
  - To allocate the requested memory in the smallest hole that is large enough (low locality)
- Next-fit
  - To allocate in the chunk that has last been split
- Worst-fit
  - put the object in the largest possible hole
  - under what workload is this good?
    - objects need to grow
    - eg. database construction
    - eg. network connection table

Heap Deallocation

- No deallocation
  - Stop when space run out
- Explicit (manual) deallocation
  - free (C, PL/1), delete (C++), dispose (Pascal), deallocation (Ada)
  - May lead to memory leak and dangling reference
- Implicit deallocation
  - Reference count
  - Garbage collection

Garbage

- Dynamically allocated storage can become unreachable
- For example, in program just examined, let's say a new line reads:

```
head↑.next := nil;
```

- Some languages perform garbage collection
- In other languages, the program must explicitly deallocate storage
- In languages without garbage collection, garbage remains until program finishes

Dangling References

- A dangling reference occurs when storage that has been deallocated is referenced
- For example, in program just examined, let's say a new line reads:

```
dispose(head↑.next);
```

- A dangling reference can lead to unpredictable behavior in a program

Garbage Collection (GC)

- Remove the burden of manual deallocation from the programmer by automatically deallocating unreachable data objects
- Dates back to the initial implementation of Lisp in 1958
- Java, Perl, Modula-3, Prolog and Smalltalk offers garbage collection

Soundness and Completeness

- For any program analysis
  - Sound?
    - are the operations always correct?
    - usually an absolute requirement
  - Complete?
    - does the analysis capture all possible instances?
- For Garbage Collection
  - sound = does it ever delete current memory?
  - complete = does it delete all unused memory?

GC Assumptions

- We assume objects have a type that can be determined by GC at runtime. From the type information, we can tell how large the object is, and which components contain pointers (to other objects)
- We assume references to objects are always to the address of the beginning of the object. Thus all references to the same object have the same value and can be identified

Type Safety

- Based on our assumption, GC cannot be applied to type unsafe languages such as C and C++ (where integer can be cast as pointer, arithmetic operations can be applied to pointers, …)
- Since most C/C++ programs do not generate pointers arbitrarily, some unsound GCs may work in practice. A conservative approach can also be used (treat any bit pattern that may form a valid address as pointers).

Reachability and Root Set

- Root set
  - All the data that can be accessed directly by a program without having to dereference any pointer
  - For example, in Java, the root set is all the static field members and all variables on the stack
- Impact of compiler optimizations
  - Reference variables might have been kept in registers
  - Compiler may use arithmetic operations to generate new pointers
- In such cases, the compiler should assist GC to find the correct root set
  - Restrict GC invocation at certain safe point
  - Annotation to inform GC
  - To ensure a reference to the base address of every reachable object

Change of Reachable Objects

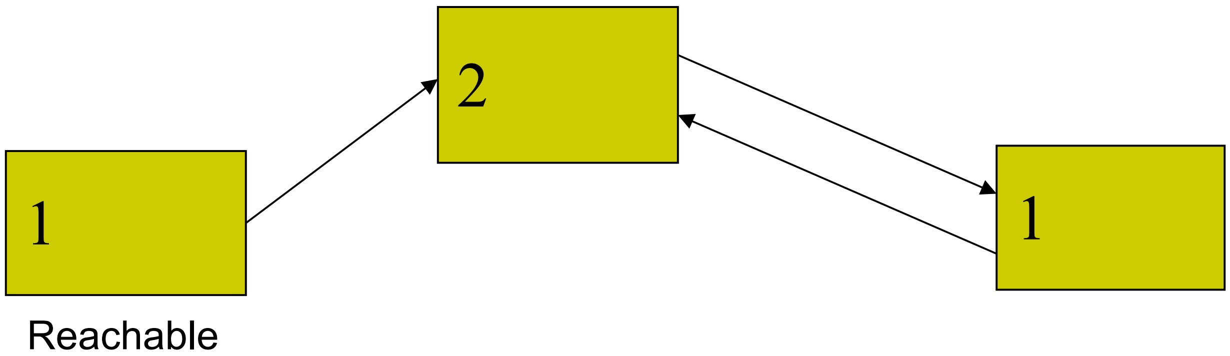
- Object Allocation: adds members to reachable set
- Parameter passing and return value
- Reference assignments
  - x=y, x is now a reference to the object pointed to by y. the original reference in x is now lost. If it is the last reference to the object, this object becomes unreachable.
- Procedure return.
  - The frame holding local variables are popped off. Some variables there may hold the last reference to an object

Reference Count-Based GC

- We maintain a count of the references to an object, as the program performs actions that may change the reachability set. When the count goes to 0, the object becomes unreachable
  - Allocation: set ref count of the new object to 1
  - Parameter passing and return value: ref count for each object passed is incremented
  - Reference assignment (e.g. x=y): *y increment and *x decrement
  - Procedure return: ref count in each reference hold in stack variables is decremented
- Transitive loss of reachability → if an object is no longer reachable, decrement the ref count for each reference it holds

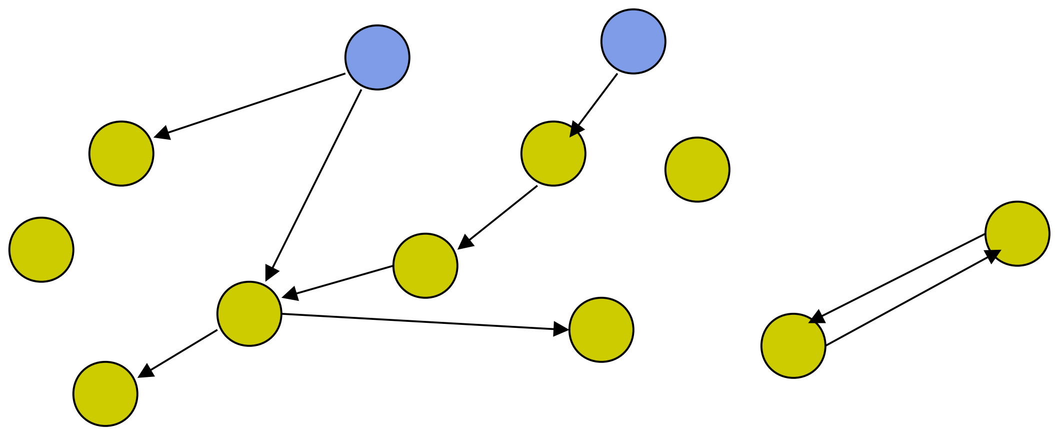
Reference Count - Based GC

- Disadvantages
  - constant cost, even when lots of space
  - optimize the common case!
  - can't detect cycles
- Has fallen out of favor



Trees

- Top-level objects
  - In the root set
- Garbage collector starts top-level
  - Builds a graph of the reachable objects



Mark and Sweep

- Two-pass algorithm
  - Execution is temporarily suspended
  - First pass: walk the graph and mark all objects
    - everything starts unmarked
  - Second pass: sweep the heap, remove unmarked
    - not reachable implies garbage
- Soundness?
  - Yes: any object not marked is not reachable
- Completeness?
  - Yes, since any object unreachable is not marked

Copy Collectors

- Instead of just marking as we trace
  - copy each reachable object to new part of heap
  - needs to have enough space to do this
  - no need for second pass
- Advantages
  - one pass
  - compaction
- Disadvantages
  - higher memory requirements

Compaction

- Compaction is a process which moves all blocks to one end of the heap
- This leaves all free space together as one large block, preventing fragmentation
- Only a benefit when dealing with a scheme allowing variable-sized blocks
- Requires information about all pointers into blocks
- When a used block is moved, all pointers to it have to be updated

Incremental Garbage Collection

- Problem of simple GC: pauses
- Collection „parallel“ to mutator
- Separate process
  – Concurrent
- Interwoven with mutator
  – Inserted into allocate routine („new“)

Pros and Cons

- Hard to state generally
  - No need to pause the mutator
  - (except short breaks)
- Better response times
- Slower overall than simple GC
  - Overhead of synchronizing mutator with collector