Run Time Environment

Mrs. Sunita M Dol

(sunitaaher@gmail.com)

Assistant Professor, Dept of Computer Science and Engineering

Walchand Institute of Technology, Solapur
(www.witsolapur.org)

Run Time Environment

- Source language issues
- Storage organization and allocation strategies
- Parameter passing

Introduction

- 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.

Fig: Position of intermediate code generator

Introduction

- 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

Fig: Position of intermediate code generator

Source Language Issues

- **Procedures**
  - A program is no more than a collection of procedures.
  - A procedure definition is a declaration that associates an identifier with a statement (procedure body)
  - The identifier is called the procedure name.
  - The statement is called the procedure body.
  - A procedure call is an invocation of a procedure within an executable statement.

Fig: Position of intermediate code generator

Source Language Issues

- **Procedures**
  - Procedures that return values are normally called function, but we'll just use the name "procedure."
  - When a procedure name appears in an executable statement, it is called at that point
  - The formal parameters are special identifiers declared in the procedure definition.
  - The formal parameters must correspond to the actual parameters in the function call.

Fig: Position of intermediate code generator

Source Language Issues

- **Procedures**
  - Example - m and n are formal parameters of the quicksort procedure. The actual parameters in the call to quicksort in the main program are 1 and 9.
  - Actual parameters can be a simple identifier, or more complex expressions.

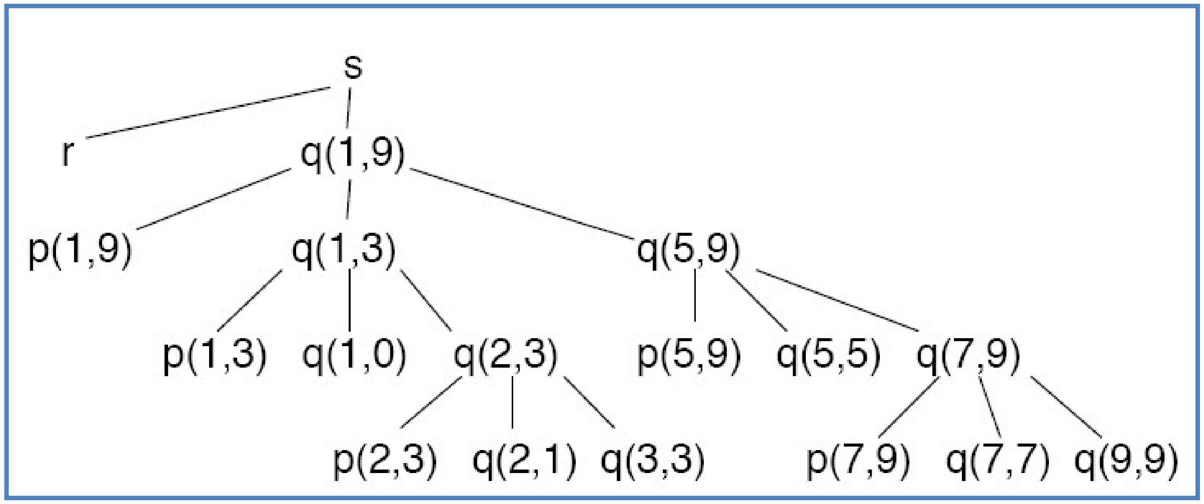
```
program sort( input, output );
  var a: array [ 0..10 ] of integer;
     procedure readarray;
                var i: integer;
                begin
                for i := 1 to 9 do read( a[i] )
                end;
     function partition( y, z: integer ) : integer;
                var i, j, x, 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
        a[0] := -9999; a[10] := 9999;
        readarray;
        quicksort(1,9);
  end.
```

Source Language Issues

- **Activation Tree**
  - Let's assume, as in most mainstream programming languages, that we have sequential program flow.
  - Procedure execution begins at the first statement of the procedure body.
  - When a procedure returns, execution returns to the instruction immediately following the procedure call.
  - Every execution of a procedure is called an activation.
  - The lifetime of an activation of procedure P is the sequence of steps between the first and last steps of P's body, including any procedures called while P is running.
  - Normally, when control flows from one activation to another, it must (eventually) return to the same activation.
  - When activations are thusly nested, we can represent control flow with activation trees

Source Language Issues

- **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.

Source Language Issues

- **Activation Tree**
  - 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

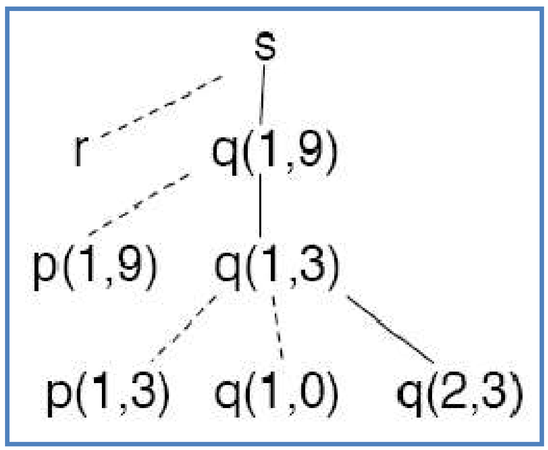
Source Language Issues

- **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

Source Language Issues

- **Control Stack**

  This partial activation tree corresponds to control stack (growing downward)



Source Language Issues

- **Declarations**
  - Every DECLARATION associates some information with a name.
  - In Pascal and C, declarations are EXPLICIT:
    var i : integer;
    assocates the TYPE integer with the NAME i.

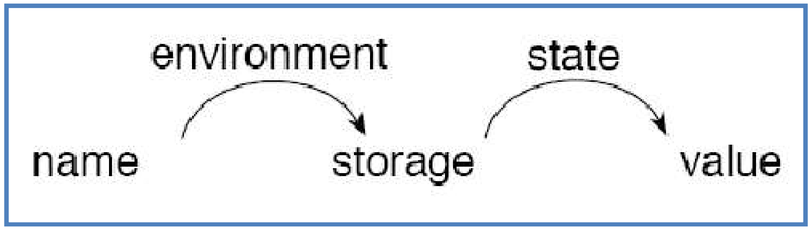
  - Some languages like Perl and Python have IMPLICIT declarations.

Source Language Issues

- **Scope of a Declaration**
  - The scoping rules of a language determine where in a program a declaration applies.
  - The scope of a declaration is the portion of the program where the declaration applies.
  - An occurrence of a name in a procedure P is local to P if it is in the scope of a declaration made in P.
  - If the relevant declaration is not in P, we say the reference is non-local.
  - During compilation, we use the symbol table to find the right declaration for a given occurrence of a name.
  - The symbol table should return the entry if the name is in scope, or otherwise return nothing.

Source Language Issues

- **Binding of names**
  - The environment is a function mapping from names to storage locations.
  - The state is a function mapping storage locations to the values held in those locations.



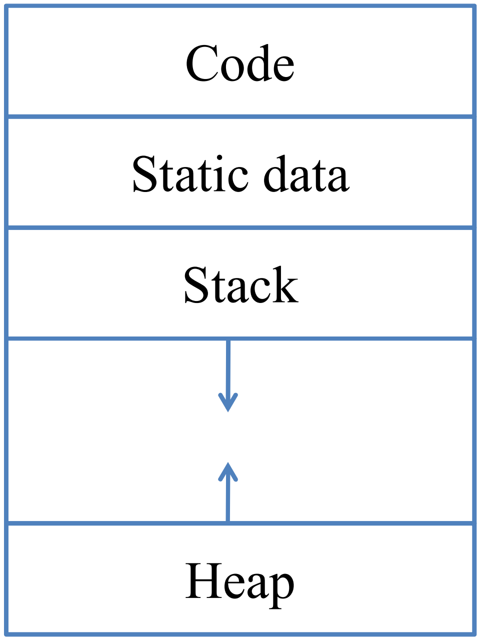
  - Environments map names to l-values.
  - States map l-values to r-values.

Source Language Issues

- **Binding of names**
  - When an environment maps name x to storage location s, we say “x is bound to s”. The association is a binding.
  - Assignments change the state, but NOT the environment:
    pi := 3.14
  - changes the value held in the storage location for pi, but does NOT change the location (the binding) of pi.
  - Bindings do change, however, during execution, as we move from activation to activation.

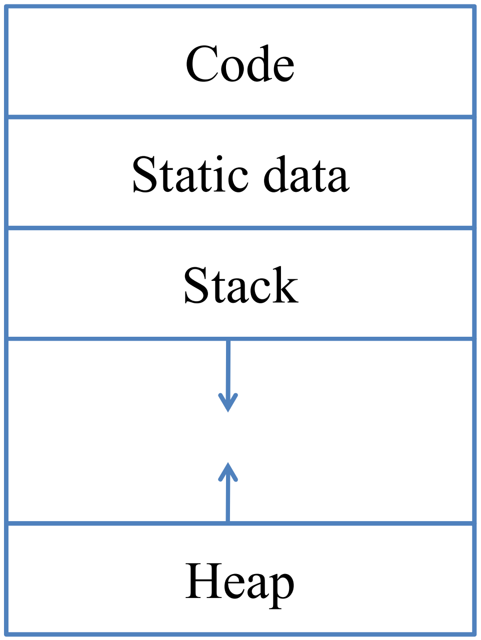
Storage Organization

- **Binding of names**
  - The runtime storage might be subdivided into
    - ✓ Target code
    - ✓ Data objects
    - ✓ Stack to keep track of procedure activation
    - ✓ Heap to keep all other information
  - The size of generated target code is fixed at compile time, so the compiler can place it in a statically determined area.
  - The size of some of data object may also be known at compile time, so the compiler can place it in a statically determined area



Storage Organization

- **Binding of names**
  - 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
  - The size of stack and the heap can change as the program executes where they can grow towards each other as needed.



Storage Organization

- **Activation Records**
  - 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

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

Storage Organization

- **Compile time layout of local data**
  - Usually the byte is the smallest addressable unit of storage.
  - We lay out locals in the order they are declared.
  - Each local has an offset from the beginning of the activation record (or local data area of the record).
  - Some data objects require alignment with machine words.
  - Any resulting wasted space is called padding.

Type	Size (typical)	Alignment (typical)
char	8	8
short	16	16
int	32	32
float	32	32
double	64	32

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 deallocates storage as needed at runtime from heap

Storage Allocation Strategies

- **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

Storage Allocation Strategies

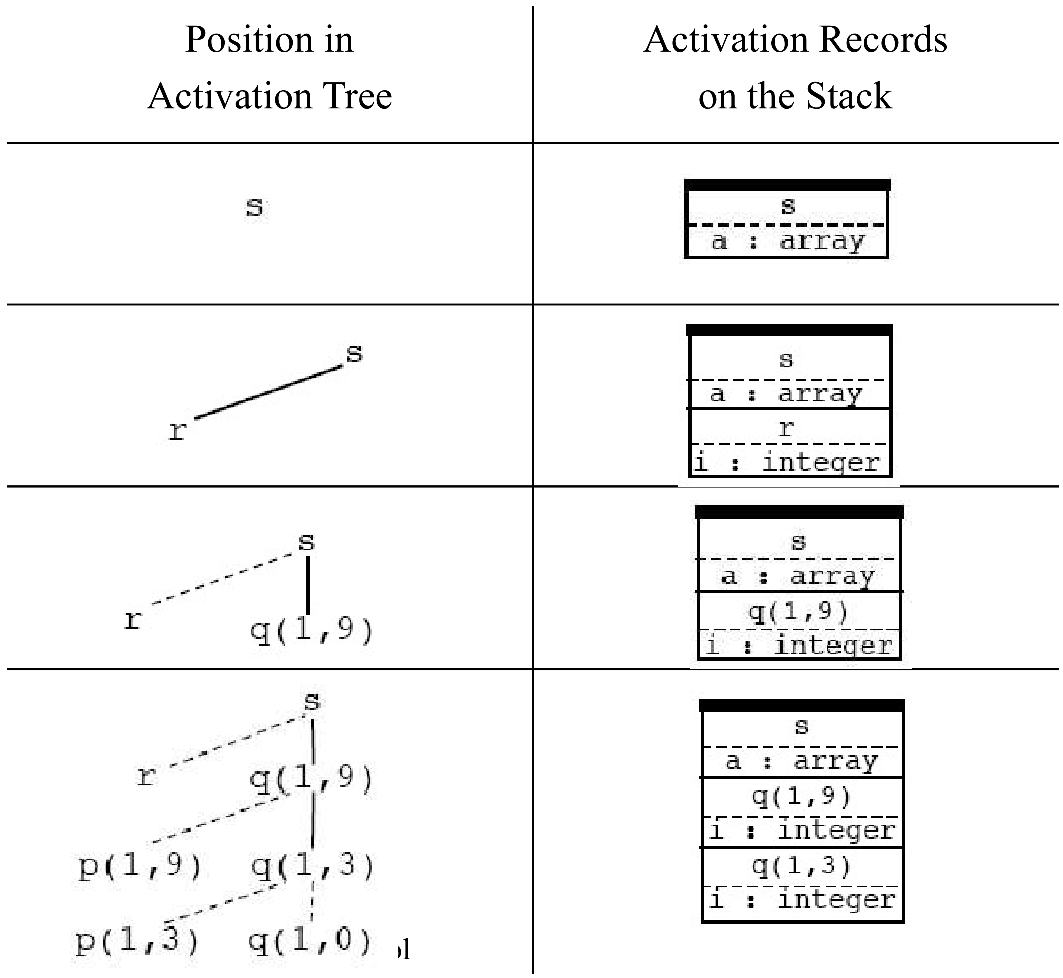
- **Static allocation**:
  - Address of a storage consists of an offset from the end of an activation record
  - Compiler decides location of each activation
  - All the addresses can be filled at compile time
  - Constraints
    - ✓ Size of all data objects must be known at compile time
    - ✓ Recursive procedures are not allowed
    - ✓ Data structures cannot be created dynamically

Storage Allocation Strategies

- **Stack 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

Storage Allocation Strategies

- **Stack allocation**:

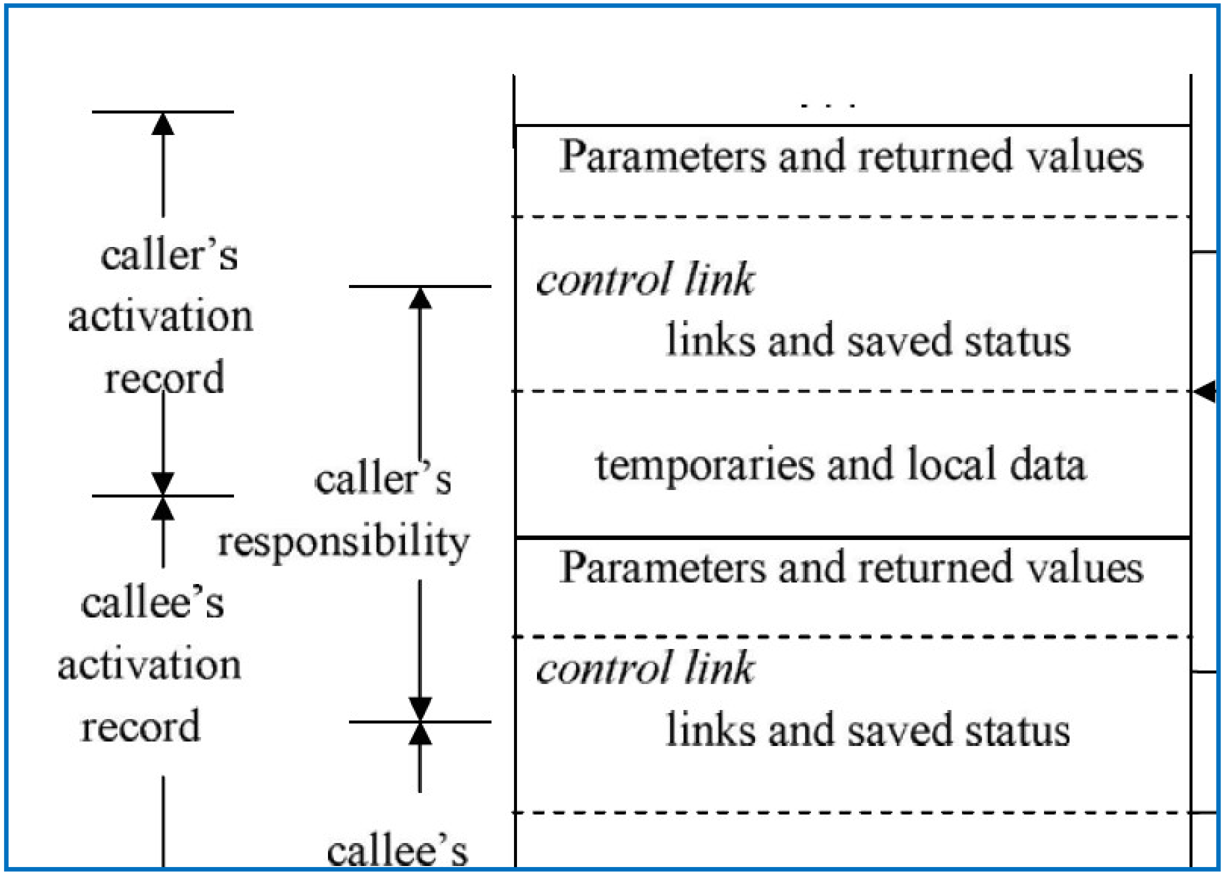


Storage Allocation Strategies

- **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

Storage Allocation Strategies

- **Stack allocation**:



Storage Allocation Strategies

- **Stack allocation**:
  - **Calling Sequence**
    - ✓ The CALLING SEQUENCE for a procedure allocates an activation record and fills its fields in with appropriate values.
    - ✓ The RETURN SEQUENCE restores the machine state to allow execution of the calling procedure to continue.
    - ✓ Some of the calling sequence code is part of the calling procedure, and some is part of the called procedure.
    - ✓ What goes where depends on the language and machine architecture.

Storage Allocation Strategies

- **Stack allocation**:
  - **Calling Sequence – Sample calling sequence**
    1. Caller evaluates the actual parameters and places them into the activation record of the callee.
    2. Caller stores a return address and old value for stack_top in the callee's activation record.
    3. Caller increments stack_top to the beginning of the temporaries and locals for the callee.
    4. Caller branches to the code for the callee.
    5. Callee saves all needed register values and status.
    6. Callee initializes its locals and begins execution.

Storage Allocation Strategies

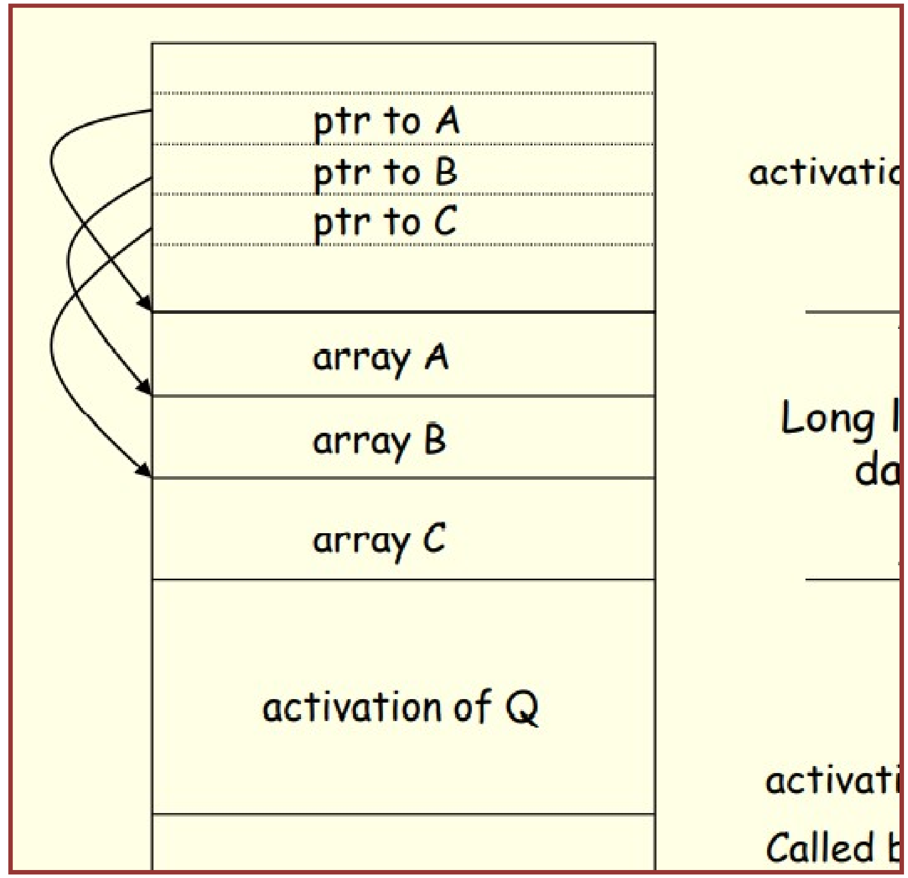
- **Stack allocation**:
  - **Calling Sequence – Sample return sequence**
    1. Callee places the return value at the correct location in the activation record (next to caller's activation record)
    2. Callee uses status information previously saved to restore stack_top and the other registers.
    3. Callee branches to the return address previously requested by the caller.

Storage Allocation Strategies

- **Stack allocation**:
  - **Variable-length data**
    - ✓ In some languages, array size can depend on a value passed to the procedure as a parameter.
    - ✓ This and any other variable-sized data can still be allocated on the stack, but BELOW the callee's activation record.
    - ✓ In the activation record itself, we simply store POINTERS to the to-be-allocated data.

Storage Allocation Strategies

- **Stack allocation**:
  - **Variable-length data**
    - ✓ Example - All variable-length data is pointed to from the local data area.



Storage Allocation Strategies

- **Dangling References**
  - Referring to locations which have been deallocated

```
main()
{
        int *p;
        p = dangle(); /* dangling reference */
}
int *dangle()
{
        int i=23;
        return &i;
}
```

Storage Allocation Strategies

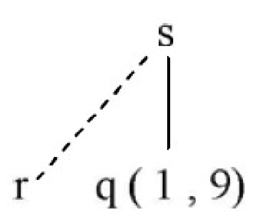
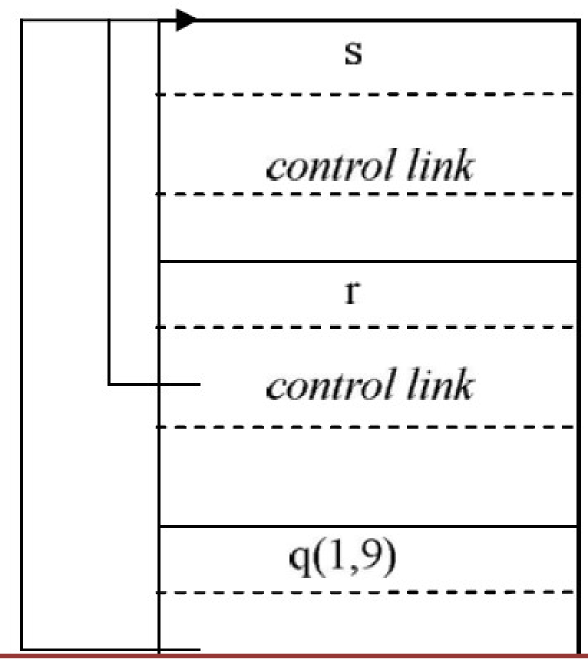
- **Heap allocation**
  - Stack allocation cannot be used if:
    - ✓ The values of the local variables must be retained when an activation ends
    - ✓ A called activation outlives the caller
  - In such a case de-allocation of activation record cannot occur in last-in first-out fashion
  - Heap allocation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

Storage Allocation Strategies

- **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
  - 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

Storage Allocation Strategies

- **Heap allocation**

Position in the activation tree	Activation records in the heap	
		Retain record

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

References

- Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullmann “Compilers- Principles, Techniques and Tools”, Pearson Education.

Thank You !!!