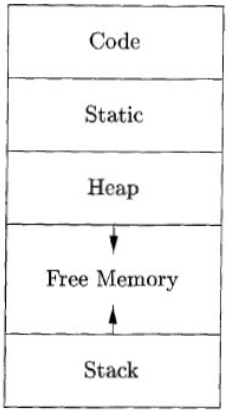
Run-time Environment

We need memory to store:

- Code – Code is of fixed size and memory requirement statically determined
- Data - global data objects can be statically allocated
- Stack area is used to keep track of control stack / activations
- Heap is used to store all other data (especially memory allocated and de-allocated under program control via malloc, free, etc.)
- Stack and Heap grow in opposite directions



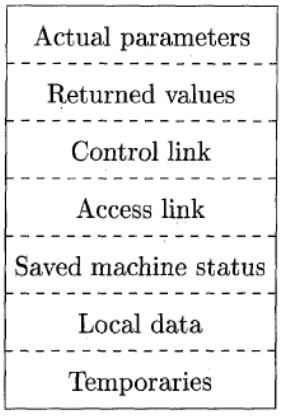
Typical memory organisation

Storage Allocation Strategies

- Static allocation lays out storage for all data objects at compile time.
  - Restrictions: size of object must be known and alignment requirements must be known at compile time.
  - No recursion.
  - No dynamic data structure
- Stack allocation manages the run time storage as a stack
  - The activation record is pushed on as a function is entered.
  - The activation record is popped off as a function exits.
  - Restrictions:
    - values of locals cannot be retained when an activation ends.
    - A called activation cannot outlive a caller.
- Heap allocation -- allocates and deallocates stroage as needed at runtime from a data area known as heap.
  - Most flexible: no longer requires the activation of procedures to be LIFO.
  - Not efficient: need true dynamic memory management.

Program is made up of procedures
Procedure = name + parameters + local variables + return value + body
Procedure call ⇒ body of the procedure is executed; control returns to instruction immediately following procedure call
Activation: each execution of a procedure
Lifetime: sequence of steps between 1st and last step in the execution of procedure body (including calls to other procedures from within the body)
Lifetimes of procedure activations are either non-overlapping or nested (as in PASCAL)

Activation Records (also called frames)
Information(memory) needed by a single execution of a procedure.



A general activation record

We need to allocate spaces onto the stack for managing procedure calls

- Stack grows with each call and shrinks with each procedure return/terminate
- Each procedure call pushes an activation record into the stack

An activation record generally contains (although contents vary with the language being implemented):

1. temporary variables (compiler created),
2. Local data, variables,
3. Saved machine status (PC, other register values)
4. Access link (optional) - specifies where non-local data can be found
5. Control link (optional) - points to activation record of caller
6. Actual parameters (registers may also be used)
7. Return value

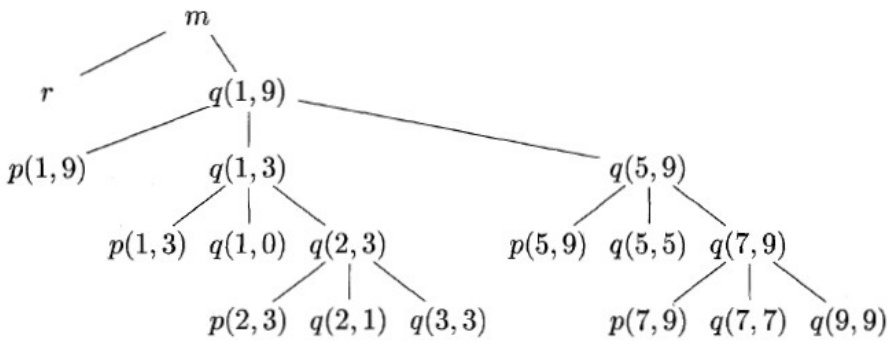
Activation tree: graphical representation of flow of activations when a program is running

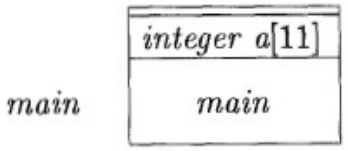
- Each node is an activation of a procedure (root of the tree is formed with “main”)
- Node ***a*** is parent of ***b*** implies control flows from activation ***a*** to ***b***
- For two siblings ***a*** , ***b*** (where ***a*** is to the left of ***b***) implies that lifetime of ***a*** is before lifetime of ***b.*** (Children of the same parent are executed in sequence from left to right)
- Actual flow of control corresponds to depth-first traversal of tree
  - Sequence of procedure calls is defined by the preorder traversal of activation tree
  - Sequence of procedure returns is defined by the postorder traversal of activation tree

EXAMPLE

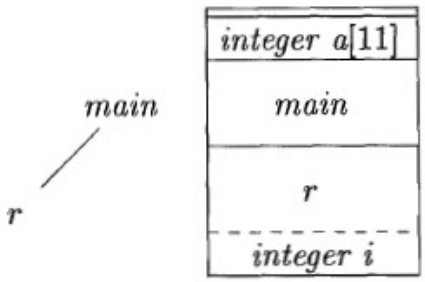
```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

```
enter main()
    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)
leave main()
```

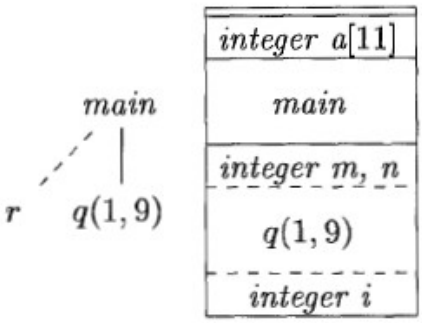




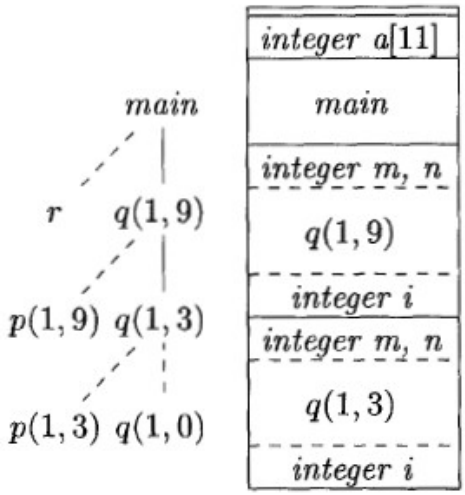
(a) Frame for *main*



(b) r is activated



(c) r has been popped and $q(1,9)$ pushed



(d) Control returns to $q(1,3)$

How are the activation record pushed and popped ?
Appropriate code must be generated by the compiler.
What makes this happen is known as calling sequence (how to implement a procedure call).
A calling sequence allocates an activation record and enters information into its fields (push the activation record).
On the other hand the return sequence restores the state of the machine so that the calling procedure can continue execution.

A ***Calling sequence***, i.e. the code that allocates activation record and the code for entering information in it is generated which is followed by a ***Return sequence***, i.e. the code to restore the state of the machine.

<u>A possible calling sequence:</u>

The caller evaluates actuals and push the actuals on the stack
The caller saves return address(pc) the old value of sp into the stack
The caller increments the sp
The callee saves registers and other status information
The callee initializes its local variables and begin execution.

A possible return sequence:

The callee places a return value next to the activation record of the caller.
The callee restores other registers and sp and return (jump to pc).
The caller copies the return value to its activation record.

In today's processors, there is usually special support for efficiently realizing calling/return sequence.

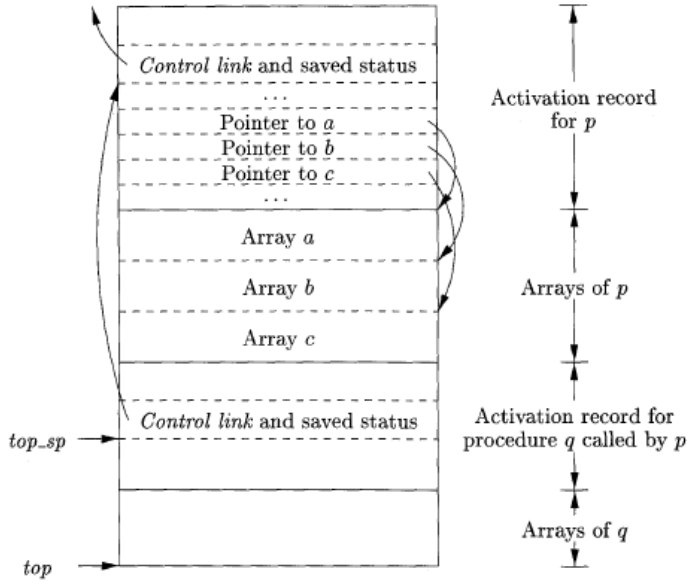
Variable length parameter lists

Callee code must handle variable length parameter lists.
Usually, first parameter specifies number of parameters.

Variable Length Data

If size of local array can not be determined at compile time, then arrays must be allocated at runtime.

- Allocate <ptr>
- Allocate array[] at runtime
  (grow stack at runtime)
- ptr = array



Access to local variables:

- Stack relative: variable is synonymous with relative location of the activation record (+/- offset to stack)

Access to nonlocal variables (non-nested)

Nonlocal variables in C (without nested procedures) need to be accessed.
All data declared outside procedures are static.
Compiler determines relative address of variable with respect to module/file into data segment.
Linker merges all segments into a single segment and changes the offsets leading to the global address.

Some languages without nested procedures may still have nested scopes (blocks).
How to manage with nested scopes?

- Treat a block as a parameter-less procedure
- Allocate space for all blocks in a procedure.
- Block variables are treated local variables for procedures

- Block variables allocated when block is entered and deallocated on leaving block
- Flow of control is simple: called only from point just before block and returns to point just after block

Access to nonlocal variables (nested procedures)

Access link: if procedure P is nested immediately within Q in the source text, then the access link in Activation Record of P points to the access link in the Activation Record for the most recent activation for Q.

Nesting depth (ND):

ND of top-level procedure (e.g. main) is 1
if P is nested immediately within Q , $ND_P = ND_Q + 1$

Setting up access links:
Let procedure P (nesting depth N_p) call Q (nesting depth N_q) -

1. If $N_p < N_q$: Q must be declared within P
   - *make access link in AR_Q point to AR_P (located just below AR_Q in stack)*

2. If $N_p \geq N_q$: enclosing procedures at depths $1 \ldots N_{(q-1)}$ must be same for both P and Q
   - *follow $N_p - N_q + 1$ access links from caller (P) to reach the most recent AR of the procedure that statically encloses both P and Q most closely*
   - *set access link in AR_Q to point to this AR.*

Using access links:
Suppose we have a reference to non-local variable *a* in procedure P. To find storage location for *a* :

1. Follow $N_p - N_a$ links from the AR at top of stack (AR_p).
2. The AR reached corresponds to the procedure that *a* is local to.
3. Storage location of *a* is at a known offset within the *local variables - temps* section of this AR

Heap Management
Heap stores data that lives indefinitely.
A **Memory manager** subsystem is responsible for allocation and de-allocation of space within the heap.
In many languages **garbage collection** process runs to find spaces within the heap that are no longer used and reallocate them to other data items.

Memory Manager

- Keeps track of all the free space in heap at all time
- Performs allocation through interaction with OS and also de-allocation
- Desired properties:
  - Space efficiency: minimize total heap space needed by programs
  - Program efficiency: making good use of memory subsystem
  - Low overhead should be maintained for allocation and de-allocation processes
- Allocation policies are needed to avoid Heap Fragmentation

Garbage Collection

- Garbage: data that cannot be referenced
- Garbage collection: reclamation of garbage from heap
- It is assumed that Objects have a type that can be determined by garbage collector at run-time.
- Also assumed that References to objects are always to the address of the beginning of the object.
- Requirements:
  - Overall execution time can be slowed down due to garbage collection. This needs to be avoided
  - Space usage: must avoid fragmentation
  - Maximum pause time must be minimized
  - Program locality should be maintained

Reference-Counting for Garbage Collection

- Every object must have a field for reference count
- This field counts the number of references to the object
- If count reaches zero, the object is deleted

There are many algorithms for Garbage Collection

- **Mark and Sweep**
- **Mark-and-Compact**

Incremental Garbage Collectors

- **Generational Garbage Collectors**

<u>Mark and Sweep</u>

(Reference: http://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/)

Any garbage collection algorithm must perform 2 basic operations. One, it should be able to detect all the unreachable objects and secondly, it must reclaim the heap space used by the garbage objects and make the space available again to the program.

The above operations are performed by Mark and Sweep Algorithm in two phases:
1) Mark phase
2) Sweep phase

Mark Phase: When an object is created, its mark bit is set to 0(false). In the Mark phase, we set the marked bit for all the reachable objects (or the objects which a user can refer to) to 1(true). Now to

perform this operation we simply need to do a graph traversal, a depth first search approach would work for us. Here we can consider every object as a node and then all the nodes (objects) that are reachable from this node (object) are visited and it goes on till we have visited all the reachable nodes.

- Root is a variable that refer to an object and is directly accessible by local variable. We will assume that we have one root only.
- We can access the mark bit for an object by: markedBit(obj).

Algorithm -Mark phase:

```
Mark(root)
   If markedBit(root) = false then
      markedBit(root) = true
      For each v referenced by root
          Mark(v)
```

Note: If we have more than one root, then we simply have to call Mark() for all the root variables.

Sweep Phase: As the name suggests it “sweeps” the unreachable objects i.e. it clears the heap memory for all the unreachable objects. All those objects whose marked value is set to false are cleared from the heap memory, for all other objects (reachable objects) the marked bit is set to false.
Now the mark value for all the reachable objects is set to false, since we will run the algorithm (if required) and again we will go through the mark phase to mark all the reachable objects.

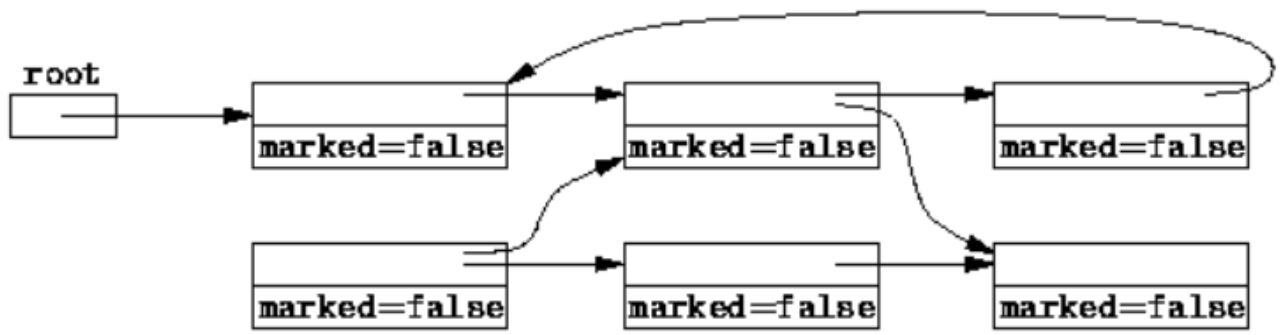
Algorithm – Sweep Phase

```
Sweep()
For each object p in heap
   If markedBit(p) = true then
      markedBit(p) = false
   else
      heap.release(p)
```

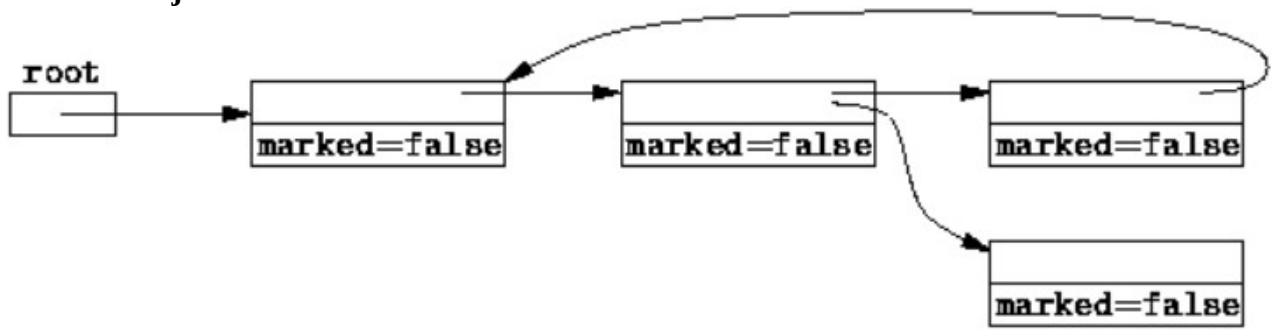
The mark-and-sweep algorithm is called a tracing garbage collector because is traces out the entire collection of objects that are directly or indirectly accessible by the program.

Example:

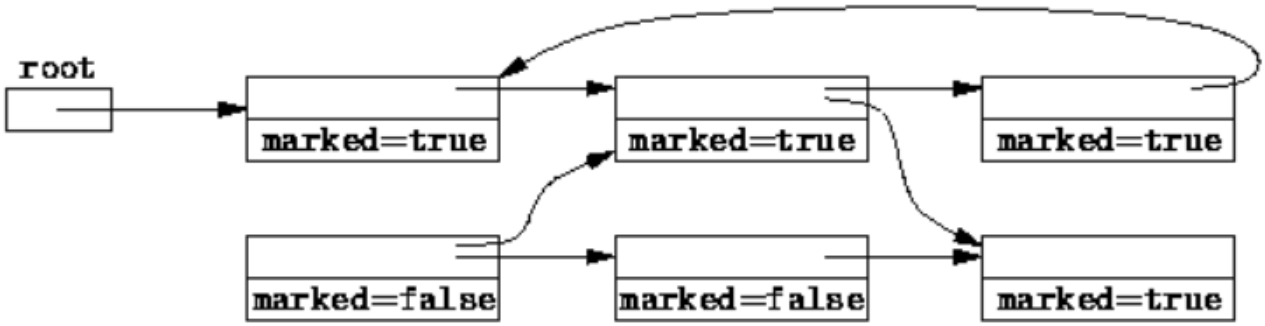
a) All the objects have their marked bits set to false.



b) Reachable objects are marked true



c) Non reachable objects are cleared from the heap.



Advantages of Mark and Sweep Algorithm

- It handles the case with cyclic references, even in case of a cycle, this algorithm never ends up in an infinite loop.
- There are no additional overheads incurred during the execution of the algorithm.

Disadvantages of Mark and Sweep Algorithm

- The main disadvantage of the mark-and-sweep approach is the fact that normal program execution is suspended while the garbage collection algorithm runs.
- Other disadvantage is that, after the Mark and Sweep Algorithm is run several times on a program, reachable objects end up being separated by many, small unused memory regions.

Mark and Compact

Two main phases:
1) Tracing/marking - Mark all the objects.
2) Compacting – a) Relocate the objects b) Update the pointer values of all the references to objects that were moved.
(The number/order of passes and the way in which objects are relocated varies).

Compaction order - Three ways to rearrange objects in the heap:
a) Arbitrary: objects are relocated without regard for their original order.

(Fast, but leads to poor spatial locality.)
b) Linearising: objects are relocated to be adjacent to related objects (siblings, pointer and reference, etc.)
c) Sliding: objects are slid to one end of the heap, “squeezing out” garbage, and maintaining the original allocation order in the heap.
(Used by most modern mark-compact collectors.)