Dynamic Storage Allocation

Programming Langages
CSCI-GA.2110

Dynamic Storage Allocation

Dynamic Storage allocation refers to the run-time allocation of storage for variables, structures, procedure calls, etc.

- For most languages, the total amount of storage required for these of objects cannot be determined at compile time.
    - FORTRAN is the exception.

Some language features requiring dynamic allocation:

- Recursion
- Pointers, explicit allocation (new, malloc, cons)
- Dynamic data structures (lists, trees, graphs, dynamic arrays)
- Higher order functions (lambda's, etc.)

Stack-based vs. Heap Based Allocation

In imperative languages like C and Ada, most data structures are allocated on the stack.

- Last-In-First-Out allocation is appropriate for activation records because local variables and parameters do not outlive the procedure calls that created them.
- When a procedure returns, its activation record is removed from the stack.
- The local variables are automatically "deallocated" i.e. the space they occupy becomes available for reuse.
- Works fine for local variables that are integers, booleans, characters, floats, records and arrays. The only way to access these structures is via the name of the variable.
- When the scope of the local variables is exited (by procedure return), their values can no longer be accessed and thus deallocation is safe.

Stack-based vs. Heap Based Allocation

What about pointers?

- Pointers are used for aliasing. They allow multiple identifiers to refer to the same object.
- A pointer's value is the address of object that it points to.
- Pointer values may be assigned, passed as parameters, etc.
- No copying of the object is entailed in these operations.
- In some object-oriented languages, such as Java, all objects have pointer semantics.
    - A variable declared to be of a object type is always a pointer.

In general, when the procedure containing a local pointer variable p returns, we cannot assume that the object that p points to can longer be accessed.

- The pointer value may have been assigned to a non-local variable, stored in a global structure, or returned by the procedure.

```
struct foo *f()
{ struct foo *c;
  c = (struct foo *) malloc(sizeof(struct foo));
  return c;
}
```

```
(define (f g x y)
  (let ((c (cons x y)))
     (g c)))
```

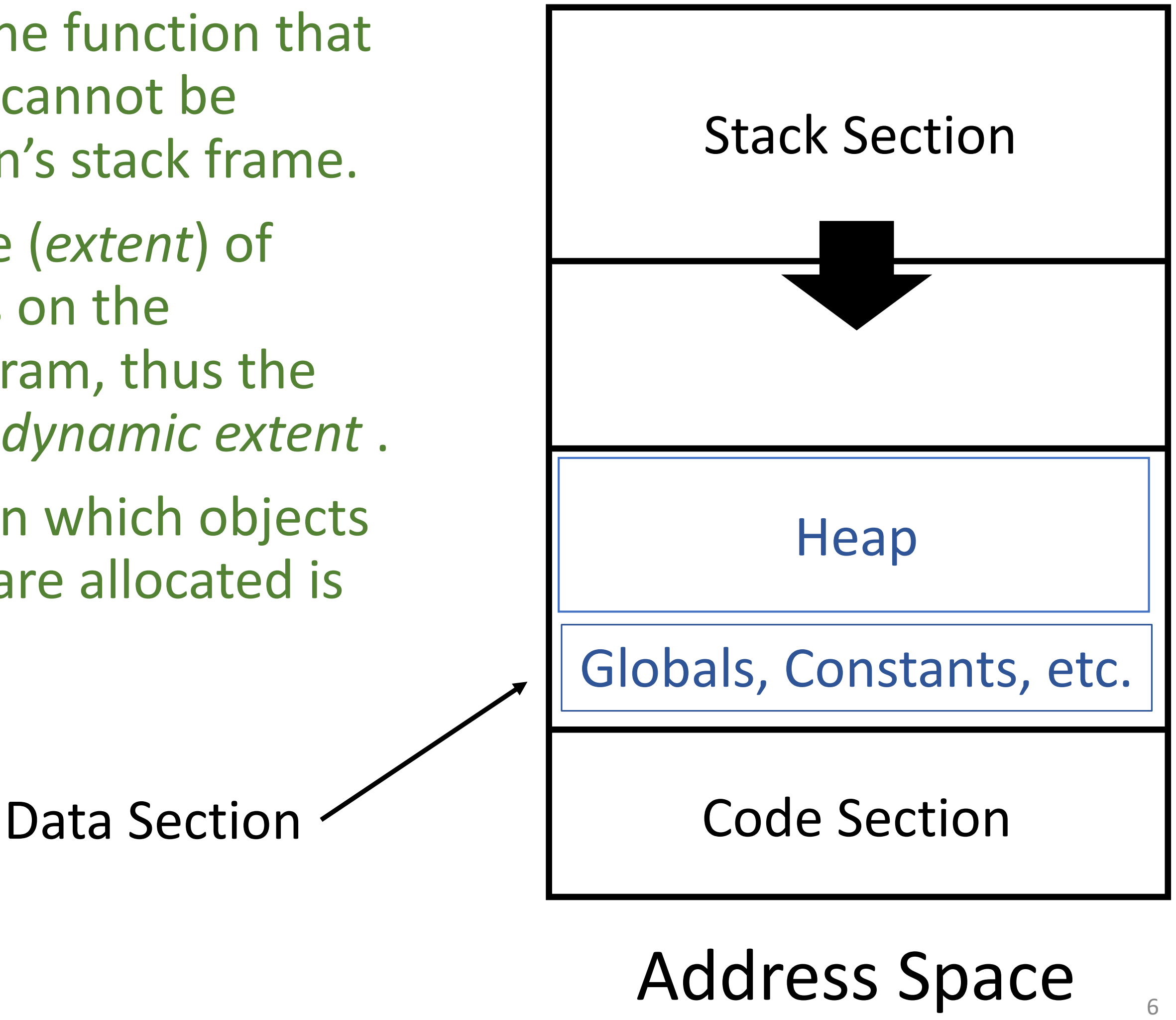
```
class C { … }

C f() {
  C c = new C();
  return c;
}
```

```
fun f x =
  let fun c y = x + y
  in c
  end
```

The object that c points to in the above code cannot be deallocated when the procedure f exits.

- If an object outlives the function that created it, the object cannot be stored in that function’s stack frame.
- Generally, the lifetime (*extent*) of such objects depends on the execution of the program, thus the object is said to have *dynamic extent* .
- The area of memory in which objects with dynamic extent are allocated is called the *heap* .



If the program creates a sufficient number of heap-allocated objects, the heap will fill up.

- At this point no more heap space will be available for more objects and the program will crash.
- However, some of the objects in the heap may no longer be needed in the program. The storage for these objects should be reclaimed, i.e. made available for subsequent use.
- Reclaiming heap-allocated objects can be specified explicitly by the user (via calls to free or delete) or can be performed automatically by the system.

Automatic reclamation of heap allocated objects is generally called *garbage collection* .

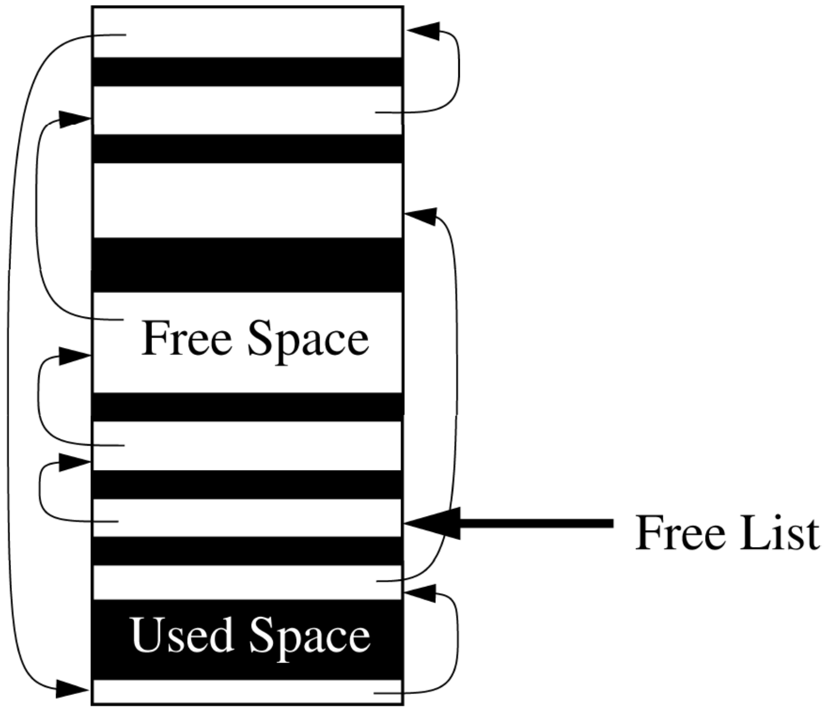
Heap Storage Allocation: Free-List vs. Heap Pointer

Heap space for objects can be allocated in one of two ways:

- Free List and Heap Pointer

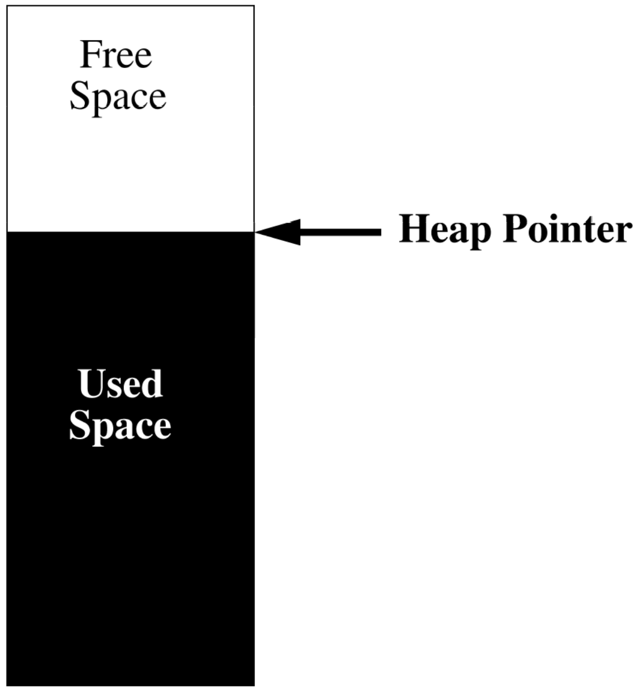
Free List Method

- Available blocks of storage are chained together in a linked list, called the *free list*.
- When a block of storage is required, it is removed from the free list.
- When a block of storage is reclaimed, it is inserted in the free list.
- Disadvantages:
  - The free list may have to searched for a block of the right size.
  - Leads to fragmented memory (analogous to segmented virtual memory in operating systems).
  - Expensive allocation, cheap reclamation



Heap Pointer Method

- Maintain available space as a contiguous block of bytes.
- A heap pointer, analogous to a stack pointer, points to the next available byte.
- When an object is allocated, the heap pointer is bumped up by the appropriate amount.
- Disadvantages:
  - Requires compaction of live objects (pushing them together until they are all adjacent).
  - Cheap allocation, expensive reclamation (?)



Storage Reclamation: Explicit vs. Automatic

Some languages require the programmer to explicitly specify when the space occupied by an object can be reclaimed.

- e.g. C, Ada, C++
- Library procedures to reclaim objects are provided to the user to call.
  - e.g. free(), delete()

Other language support automatic storage reclamation (*garbage collection).*

- *e.g. Scheme, ML, Java, Scala, Python*

Categories of Garbage Collection (GC)

Simply put, the two tasks that a garbage collector performs are:

1. Find each block of storage occupied by an object that cannot subsequently be referenced during execution (a *dead* object), and
2. Make that storage available for subsequent use.

There are two major categories of garbage collection:

- Mark/Sweep Collectors and Copying Collectors
  - When the heap fills up, these collectors traverse memory, determining the set of all *live* objects (i.e. those not dead).
  - The complement of this set is the set of dead objects, whose storage can be reclaimed.
- Reference Counting Collectors
  - During execution, these determine when a particular object has become dead.
  - At that point, the storage for that object is reclaimed.

Mark/Sweep and Copying Garbage Collection

During execution, an object x is *live* (i.e. can be referenced) if:

- It is pointed to directly by a variable, i.e. there is a pointer to x in an activation record or global data area.
- There is a register containing a temporary or intermediate value that points to x.
- There is an object y in the heap containing a pointer to x, and y is itself live.

All live objects in the heap can be found by a graph traversal.

- The starting points for the traversal are the local variables on the stack, the global variables, and the registers.
  - These are called the *roots* of the traversal.
- Any object that is not reachable from the roots is dead and can be reclaimed.

Mark/Sweep Garbage Collection

- Each object contains an extra bit called a *mark bit* .
- When the heap fills up, program execution is suspended and the garbage collection process begins.
- The garbage collector traverses the heap, starting at the roots, and sets the mark bit of each object encountered. This traversal is called the *mark phase* .
- When the traversal is complete, all live objects will have their mark bits set. Thus, any object whose mark bit is not set is unreachable. The collector then scans the entire memory, placing each object whose mark bit is not set on the free list. This is called the *sweep phase*.

Mark/Sweep Garbage Collection

```
Garbage_collect()
{   for each root pointer p do
        mark(p);
    sweep() ;
}


mark(p)
{ if p->mark != 1 then {

     p->mark := 1
     for each pointer field p->x do
         mark(p->x);

  }
}


sweep()
{ for all objects o in memory do
     if o.mark == 0 then insert(o, free_list);
}
```

Cost of Mark/Sweep GC

Let L = amount of storage occupied by live objects
M = size of heap

$$\begin{aligned} \text{Cost}_{M/S} &= O(L) + O(M) \\ &= O(M) \quad \text{since } M > L. \end{aligned}$$

- The cost of Mark/Sweep is proportional to the size of the heap, no matter how many of live objects there are.

Disadvantages of Mark/Sweep Collection:

- Non-compacting, so free list must be used.
- Expensive allocation (searching the free list)
- Program suspends during GC.
  - unsuitable for real-time programs.
  - known as a "stop-and-collect" method.

Copying Garbage Collection

There are two heaps, the FROM space and the TO space, only one of which is in use at any time.

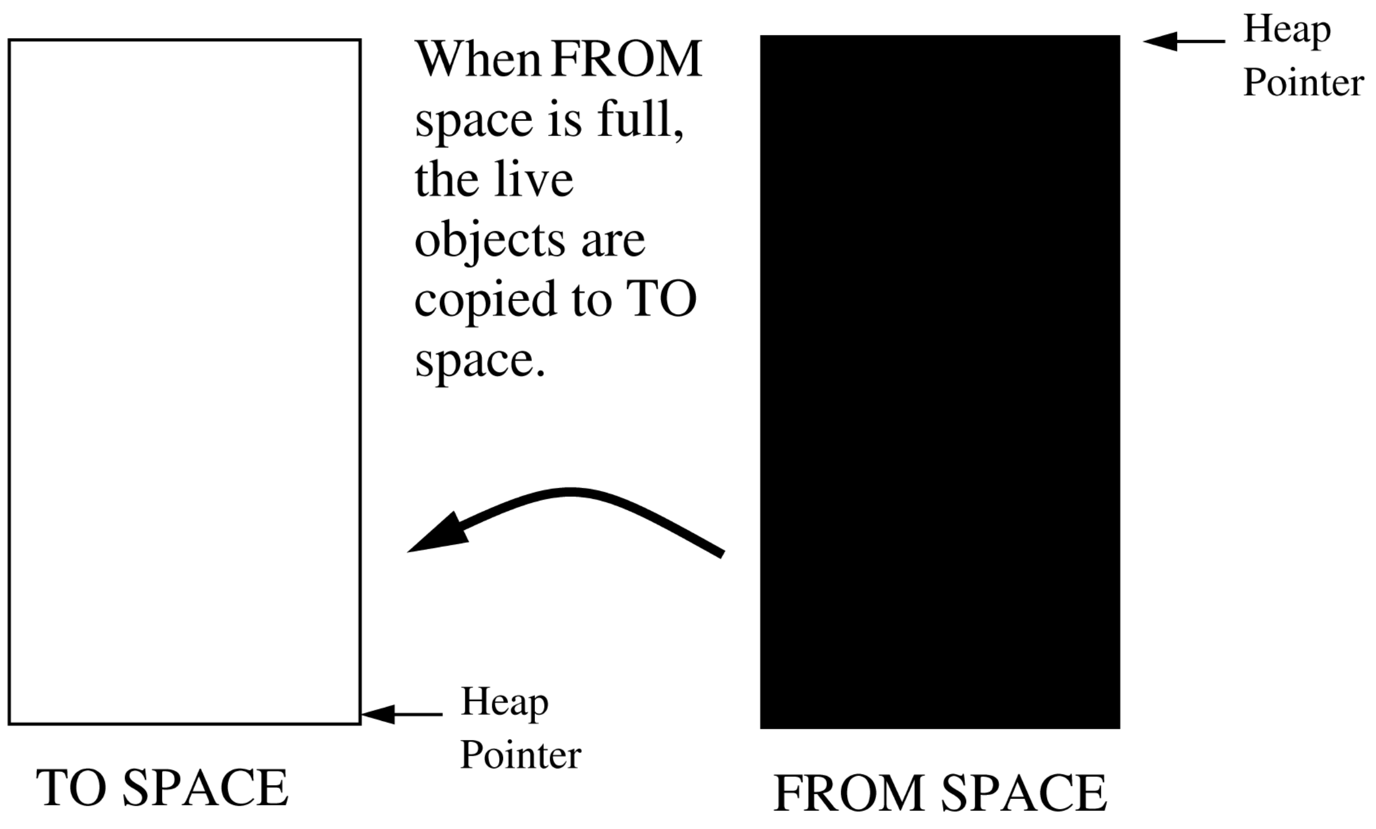
Objects are allocated in the FROM space. When the FROM space fills up, the garbage collection process begins.

As the collector traverses the graph, it copies each object it encounters from the FROM space to the (empty) TO space.

When the traversal is complete, all live objects will have been copied into TO space.

At this point, the spaces are "flipped", *i.e.* the TO space becomes the FROM space, the FROM space becomes the TO space, and program execution resumes.

Copying Garbage Collection



When FROM space is full, the live objects are copied to TO space.

Properties of Copying GC

Each live object is copied into the first available location in TO space.

When the traversal is complete, the live storage will have been compacted.

- Thus, heap allocation is performed via a heap pointer.

Since objects are being moved from FROM space to TO space, pointers to the object must be updated to reflect its new address.

- This is achieved by leaving a "forwarding address" in the old object once it is moved.
- If a pointer to the old object is encountered later on during the traversal, the pointer is updated with the forwarding address.

How can thec collector tell if an address is a "forwarding address" or an ordinary pointer?

- A forwarding address is an address in TO space. No ordinary pointer can point from FROM space into TO space.

Copying GC

```
Garbage_collect()
{ for each root pointer p do
        traverse(p);
}


Traverse(p)
{ if p-> contains a forwarding address q then
      p:= q /*assume pass by reference*/
  else {
      new_p := copy(p, TO_SPACE);
      write_forward_address(p,new_p);
      for each pointer field p->x do
          traverse(p->x) ;
  }
}
```

Cost of Copying Collection

Let L = amount of storage occupied by live object
M = size of each heap

$\text{Cost}_{\textit{copy}} = O(L)$

- Unlike mark/sweep GC, the cost is proportional to the amount of live storage, not the size of the heap.
- Experimental Data for LISP: $L = 0.3 * M$

Doesn't each heap have to be half the size of the heap in mark/sweep GC, meaning GC will happen twice as frequently?

- No. In a virtual memory system, only the FROM space need occupy physical memory. The TO space will be swapped out to disk. Can go ahead and allocate large heaps.

Disadvantage of Copying GC

- "Stop-and-copy" is unsuitable for real-time.

Generational Copying Garbage Collection

Experimental Observation: The older an object gets, the longer it can be expected to live.

- Many objects, representing temporary or intermediate values, live for only a short time (i.e. become garbage soon after creation).
- Those objects that live longer tend to correspond to the important objects.
    - globally accessible variables, central data structures of the program, etc.

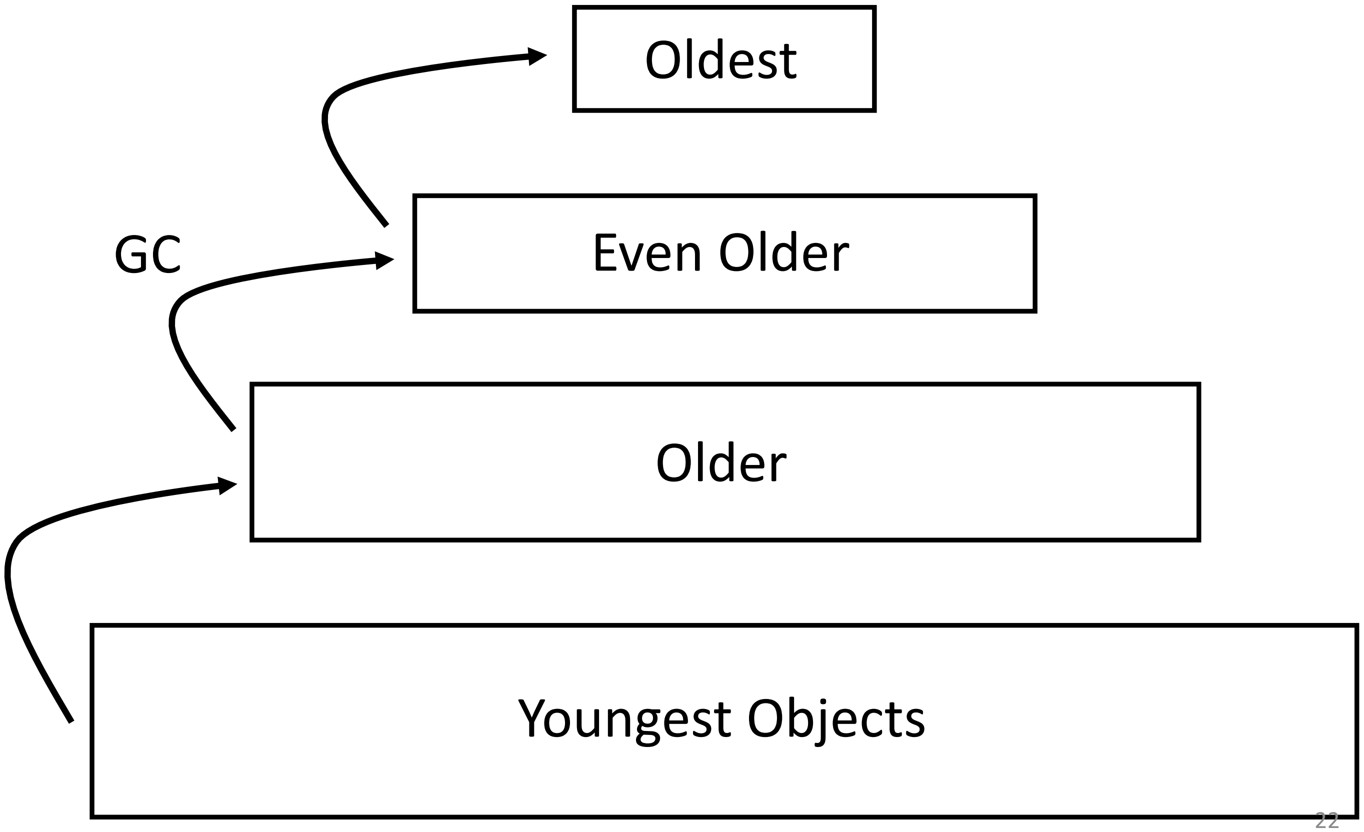
Generational Copying GC exploits this property in the following way:

- Instead of two heaps, there are many heaps.
- Each heap contains objects of similar age.
- These heaps are called generations, for obvious reasons.
- When a heap fills up, the live objects in that heap is copied to the next (older) heap.

Generational Copying GC

When a heap fills up, GC copies the live objects in only that heap to the next older heap.

No other heap is touched during GC.



Generational Copying GC

The advantages of Generational Copying GC over the simple Copying GC are:

- Since the youngest heap doesn’t contain all the older objects, rather only the youngest objects, GC will be triggered less frequently.
- Since young objects tend to die at a higher rate than older objects, and only live objects are copied to the older heap, GC will take less time on the youngest heap.
- GC will rarely happen on the older heaps which contain most of the live objects.

Reference Counting

Reference counting collection is based on a completely different idea.

Each object contains an extra field called a *reference count* .

During execution, the reference count for an object contains the count of all pointers pointing to the object.

- When a pointer to an object is copied, the object's reference count is incremented.
- When a pointer to an object is destroyed, the object's reference count is decremented.
- When an object's reference count becomes 0, the object can be reclaimed.
  - Any pointer contained in the object is "destroyed", causing more reference count activity
  - The object is placed on the free list.

Reference Counting

The code for copying and destroying pointers is:

```
copy(p1,p2) /* assume pass by reference */
{ p2 := p1;
  p1->ref_count := p1->ref_count + 1;
}


destroy(p)
{ p->refcount := p->refcount - 1;
  if p->refcount = 0 then {
     for each pointer field p->x do
        destroy(p->x);
     insert(p-> , free_list);
     }
}
```

Cost of Reference Counting

It is hard to compare the cost of reference counting to mark/sweep or copying GC.

Cost_{RC} = O(# pointers created) + O(# pointers destroyed)

but this cost is spread over the whole execution.

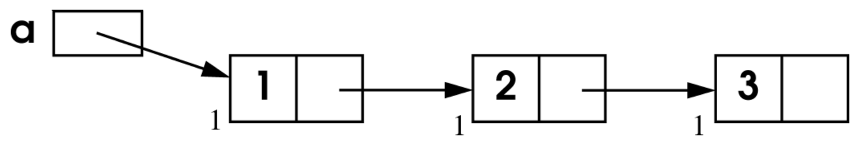
Advantages of Reference Counting:

- Storage reclamation is incremental,
- happens a little bit at a time.
- The program will not be interrupted for long. (No guarantee, though).
- Suitable for real-time programs (?)

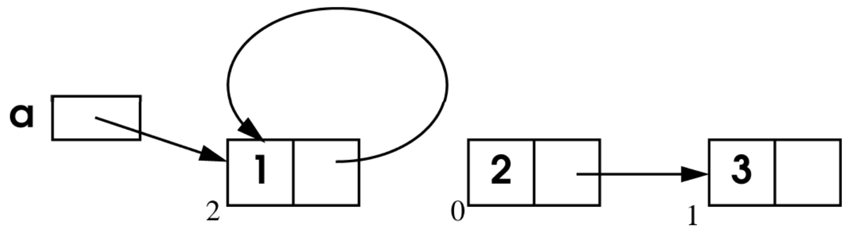
Reference Counting

Disadvantage: Cycles are a problem!

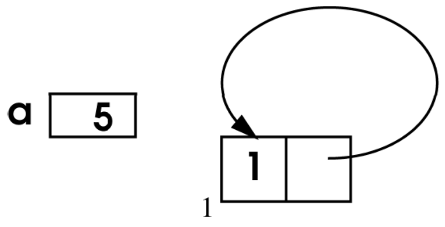
(define a '(1 2 3))



(set-cdr! a a)



(set! a 5)



The the first cons cell is dead but will never be reclaimed. Might need a backup mark/sweep collector for when the heap fills up with cycles.