Basic Garbage Collection

Garbage Collection (GC) is the automatic **reclamation** of heap records that will never again be accessed by the program. **GC** is universally used for languages with closures and complex data structures that are **implicitly** heap-allocated. **GC** may be useful for any language that supports heap allocation, because it obviates the need for **explicit deallocation**, which is tedious, error-prone, and often non-modular.
GC technology is increasingly interesting for **"conventional"** language implementation, especially as users discover that free isn't free. i.e., **explicit memory management can be costly too**. We view **GC** as part of an **allocation service** provided by the **runtime environment** to the **user program**, usually called the **mutator(user program)**. When the **mutator** needs heap space, it **calls** an allocation routine, which in turn performs garbage collection activities if needed.

Many **high-level** programming languages remove the **burden of manual memory** management from the programmer by offering automatic garbage collection, which deallocates unreachable data. Garbage collection dates back to the initial implementation of Lisp in 1958. Other significant languages that offer garbage collection include Java, Perl, ML, Modula-3, Prolog, and Smalltalk.

Principles

The basic principles of garbage collection are:

1. Find data objects in a program that cannot be accessed in the future
2. Reclaim the resources used by those objects

Many computer languages require garbage collection, **either** as part of the language specification (e.g., Java, C#, and most scripting languages) **or** effectively for practical implementation (e.g., formal languages like lambda calculus); these are said to be **garbage collected languages**. Other languages were designed for use with **manual memory management**, but have garbage collected implementations available (e.g., C, C++). Some languages, like Ada, Modula-3, and C++/CLI allow both garbage collection and manual memory management to co-exist in the same application by using separate heaps for collected and manually managed objects.

Why Garbage Collection?

- **Today's programs consume storage freely**
  - 1GB laptops, 1-4GB deskops, 8-512GB servers
  - 64-bit address spaces (SPARC, Itanium, Opteron)
- **… and mismanage it**
  - Memory leaks, dangling references, double free, misaligned addresses, null pointer dereference, heap fragmentation
  - Poor use of reference locality, resulting in high cache miss rates and/or excessive demand paging
- **Explicit memory management breaks high-level programming abstraction**

Kinds of Memory Allocation

Example

```
static int i;
void foo(void) {
  int j;
  int* p = (int*) malloc(…);
}
```

static int i;

- By compiler (in text area)
- Available through entire runtime
- Fixed size

int j;

- Upon procedure call (on stack)
- Available during execution of call
- Fixed size

int* p = (int*) malloc(…);

- Dynamically allocated at runtime (on heap)
- Available until explicitly deallocated
- Dynamically varying size

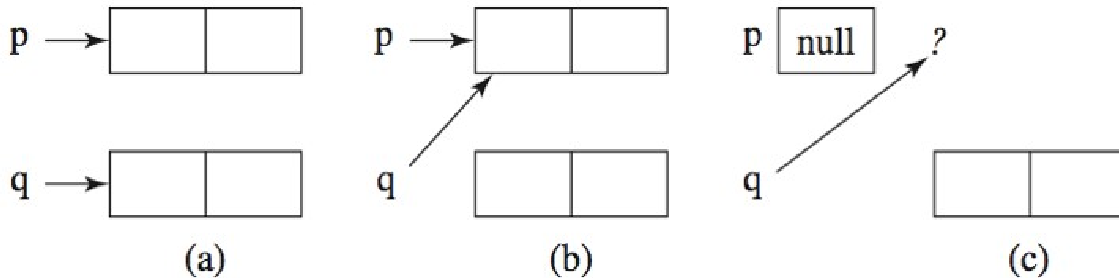
Cell = data item in the heap

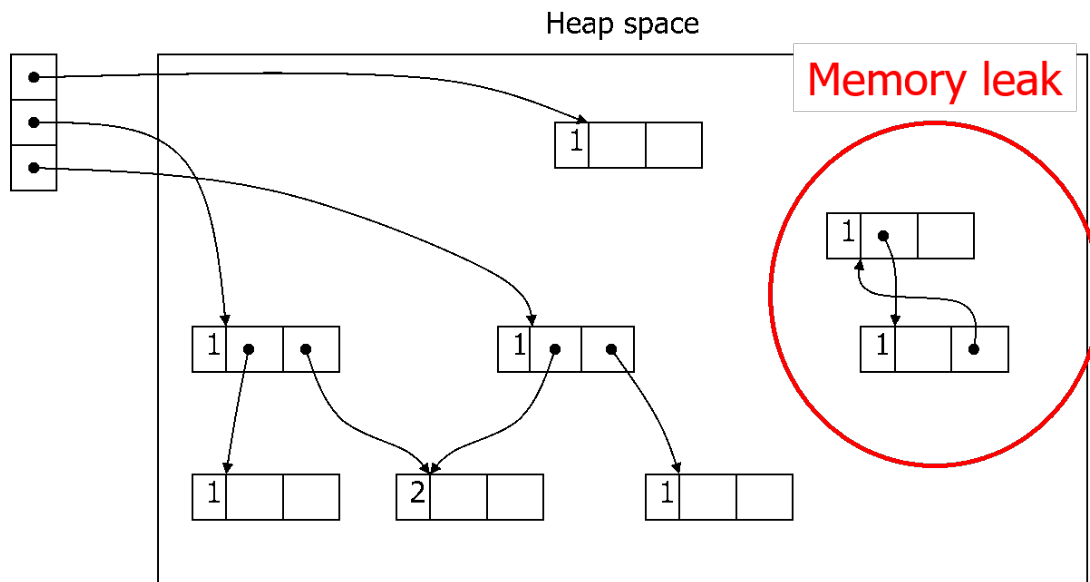
Cells are "pointed to" by pointers held in registers, stack, global/static memory, or in other heap cells

- Roots: registers, stack locations, global/static variables
- A cell is live if its address is held in a root or held by another live cell in the heap

Example of Garbage

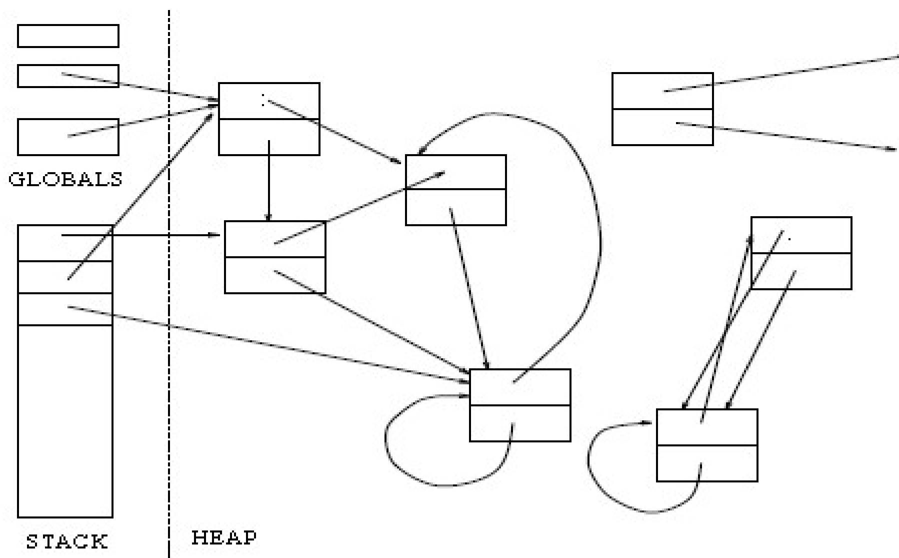
```
class node {
        int value;
        node next;
}
node p, q;
p = new node();
q = new node();
q = p;
delete p;
```





Simple Heap Model

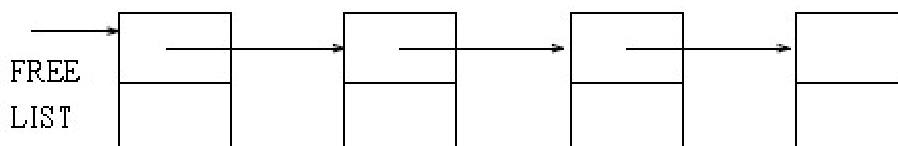
For simplicity, consider a heap containing “cons” cells.



Heap consists of **two-word cells** and each element of a cell is a **pointer** to another cell. There may also be pointers into the heap from the stack and global variables; these constitute the **root set (root set** is used as the starting point in determining all reachable data). At any given moment, the system’s **live data** are the **heap cells that can be reached** by **some** series of pointer traversals starting from a member of the root set. **Garbage** is the **heap memory containing <u>non-live cells</u>**.

Reference Counting

The most **straightforward way** to recognize garbage and make its **space reusable** for new cells is to use **<u>reference counting</u>**. We **augment each heap cell** with a **count field** that records the total number of pointers in the system that point to the cell. **Each time we create or copy a pointer to the cell**, **we increment the count**; each time we **destroy a pointer**, we **decrement the count**. If the reference count ever **goes to 0**, we can **reuse the cell by placing it on a free list**.



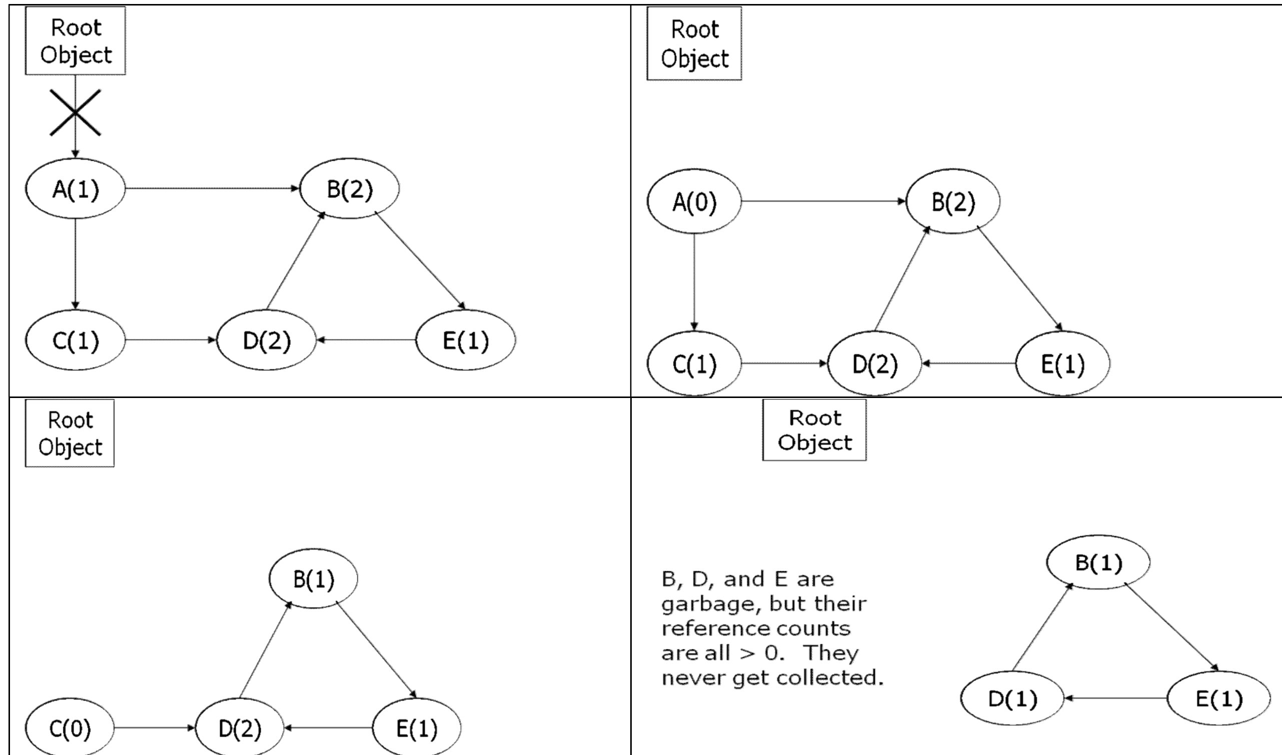
When allocating a new cell, we first try the free list (before extending the heap).

Pros:

Conceptually simple; Immediate reclamation of storage

Cons:
Extra space; Extra time (every pointer assignment has to change/check count)
Can't collect "cyclic garbage"

Example: Reference Counting



<u>**The Trace-Based Collection**</u>
Instead of collecting garbage as it is created, trace-based collectors **run periodically** to find unreachable objects and reclaim their space. Typically, we run the trace based collector whenever the **free space is exhausted** or **its amount drops below some threshold**.
All trace-based algorithms compute the set of reachable objects and then take the complement of this set. **Memory is therefore recycled as follows:**
a) The program or mutator runs and makes allocation requests.
b) The garbage collector discovers reachability by tracing.
c) The garbage collector reclaims the storage for unreachable objects.

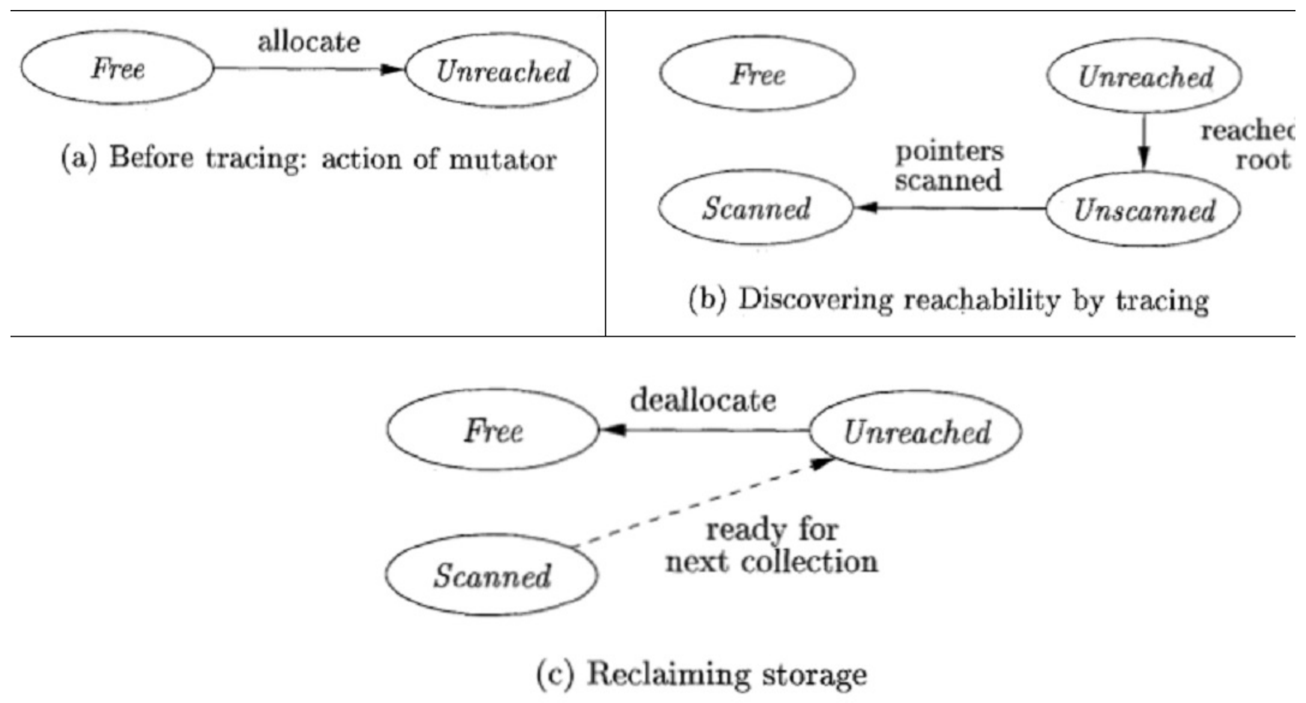
This cycle is illustrated in Figure below in terms of four states for chunks of memory:

Four States of Memory Chunks

1. **Free** = not holding an object; available for allocation (**a list of free space**).
2. **Unreached** = Holds an object, but has not yet been reached from the root set (**a work list**).
3. **Unscanned** = Reached from the root set, but its references not yet followed (**a list of allocated objects**).
4. **Scanned** = Reached and references followed (**a list of scanned objects**).

Baker's Algorithm

- Scanned = ∅
- Move objects in root set (program vars) from Unreached to Unscanned
- While Unscanned ≠ ∅
- move object **o** from Unscanned to Scanned
- scan **o**, move newly reached objects from Unreached to Unscanned
- Free = Free ∪ Unreached
- Unreached = Scanned



States of memory in a garbage collection cycle

Free, Unreached, Unscanned, and Scanned. The state of a chunk might be stored in the chunk itself, or it might be implicit in the data structures used by the garbage-collection algorithm.
While trace- based algorithms may differ in their implementation, they can all be described in terms of the following states:
1. **Free**. A chunk is in the Free state if it is ready to be allocated. Thus, a Free chunk must not hold a reachable object.
2. **Unreached**. **Chunks are presumed unreachable**, unless proven reachable by tracing. A chunk is in the Unreached state at any point during garbage collection if its reachability has not yet been established. Whenever a chunk is allocated by the memory manager, its state is set to **Unreached as illustrated in Fig (a)**.
3. **Unscanned**, **Chunks that are known to be reachable are either in state Unscanned or state Scanned**. A chunk is in the **Unscanned state** if it is known to be reachable, but its pointers have not yet been scanned, see **Fig (b)**.
4. **Scanned**. **Every Unscanned object will eventually be scanned and transition to the Scanned state, see Fig. (b)**.

The garbage collector reclaims the space they occupy and places the chunks in the Free state, as illustrated by the solid transition in **Fig. (c)**.

Mark and Sweep

There's no real need to remove garbage as long as unused memory is available. **So GC is typically deferred until the allocator fails due to lack of memory**. The **collector then takes control** of the processor, performs a collection—hopefully freeing enough memory to satisfy the allocation request—and returns control to the mutator. This approach is known generically as **"stop and collect"**. There are several options for the collection algorithm. Perhaps the simplest is called **mark and sweep**, which operates in two phases:

- First, **mark** each live data cell by tracing all pointers starting with the root set.
- Then, **sweep** all unmarked cells onto the free list (also unmarking the marked cells).

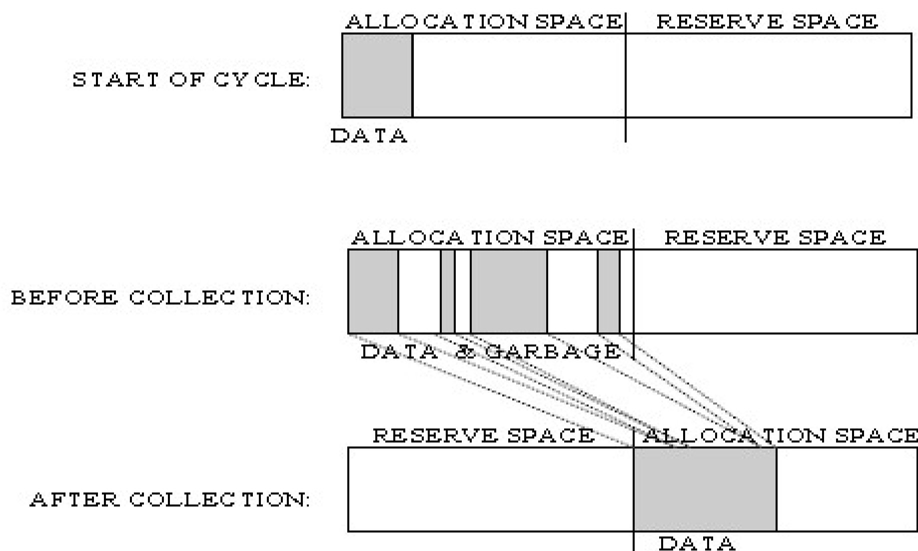
Marking	**Sweeping**
1. Assume all objects in **Unreached** state. **2.** Start with the **root set**. Put them in state **Unscanned**. **3.** while **Unscanned** objects remain do examine one of these objects; make its state be **Scanned**; add all referenced objects to **Unscanned** if they have not been there; end;	• Place all objects still in the **Unreached** state into the **Free state**. • Place all objects in **Scanned** state into the **Unreached** state. • To prepare for the next mark-and-sweep.

Copying Collection

Mark and sweep has several problems:

- It does work proportional to the size of the entire heap.
- It leaves memory fragmented.
- It doesn't cope well with non-uniform cell sizes.

An alternative that solves these problems is **copying(Compacting) collection**.
The idea is to **divide the available heap into 2 semi-spaces**. Initially, the allocator uses just **one space**; **when it fills up, the collector copies the live data** (only) into the other space, and reverses the role of the spaces.



Copying collection must fix up **all** pointers to copied data. To do this, it leaves a **forwarding pointer** in the **“from”** space after the copy is made. A copying collector typically traverses the live data graph **breadth first**, using **“to”** space itself as the search **“queue”**. Copying **compacts** live data, which improves locality and may be good for **virtual memory and caches**. Compacting collectors tend to cause caches to become more effective, improving run-time performance after collection. Compacting collectors are **difficult to implement** because they change the locations of the objects in the heap. This means that all pointers to moved objects must also be updated. This extra work can be expensive in time and storage.