## 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.

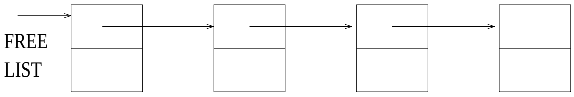
We view GC as part of an **allocation service** provided by the runtime environment to the user program, usually called the **mutator**. When the mutator needs heap space, it calls an allocation routine, which in turn performs garbage collection activities if needed.

## Reference Counting

The most straightforward way to recognize garbage and make its space reusable for new cells is to use **reference counts**.

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**.



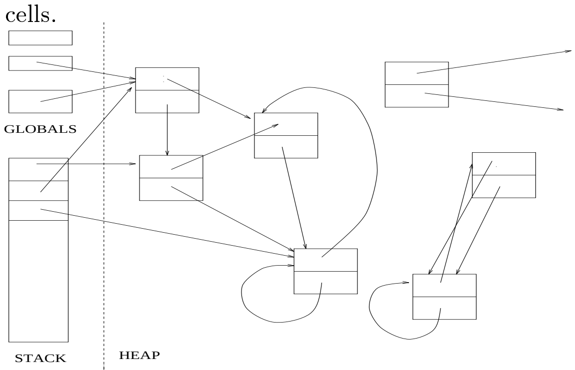

When allocating a new cell, we first try the free list (before extending the heap).

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"

## 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. (We'll deal with distinguishing pointers from non-pointers later.)

There may also be pointers into the heap from the stack and global variables; these constitute the **root set**.

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 non-live cells. (Note that this is a slightly conservative definition.)

## 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).

```
struct cell {
  int mark:1;
  struct cell *c[2];}

struct cell *free;

struct cell heap[HEAPSIZE];

struct cell *roots[ROOTS];
```

```
/* Initially all cells are on free list.
   Use c[0] to link members of free list. */
void init_heap() {
  for (i=0; i < HEAPSIZE-1; i++)
    heap[i].c[0] = &(heap[i+1]);
  heap[HEAPSIZE-1].c[0] = 0;
  free = &(heap[0]);
}

struct cell *allocate() {
  struct cell *a;
  if (!free) { /* no more room => */
    gc();      /*   try gc */
    if (!free) /* still no more room */
      die();
  };
  a = free;
  free = free->c[0];
  return a;
}
```

```
void gc() {
  for (i = 0; i < ROOTS; i++)
    mark(roots[i]);
  sweep();
}

void mark(struct cell *cell) {
  if (!cell->mark) {
    cell->mark = 1;
    mark(cell->c[0]);
    mark(cell->c[1]);
   }
}

void sweep() {
  for (i = 0; i < HEAPSIZE; i++)
    if (heap[i].mark)
      heap[i].mark = 0;
    else {
      heap[i].c[0] = free;
      free = &(heap[i]);
    }
}
```

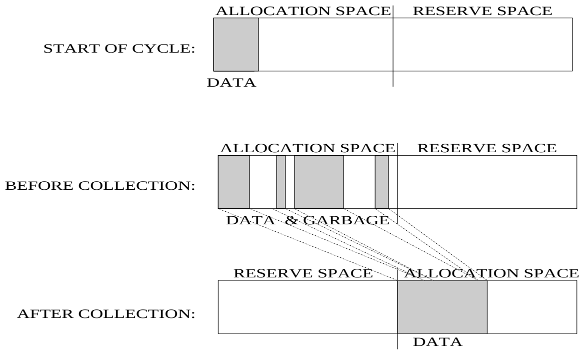
Here `mark` traverses the live data graph in depth-first order, and potentially uses **lots** of stack! A standard trick called **pointer reversal** can be used to avoid needing extra space during the traversal.

**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.

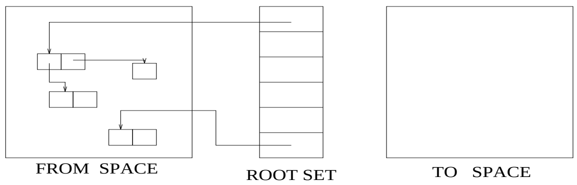
An alternative that solves these problems is **copying collection**. The idea is to divide the available heap into 2 **semi-spaces**. Initially, the allocator uses just one space; when it fills up, the collector **copies** the live data (only) into the other space, and reverses the role of the spaces.



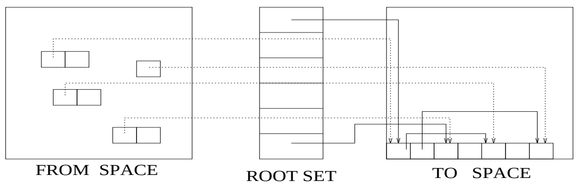

Copying collection must fix up **all** pointers to copied data. To do this, it leaves a **forwarding pointer** in the "from" space after the copy is made.

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.

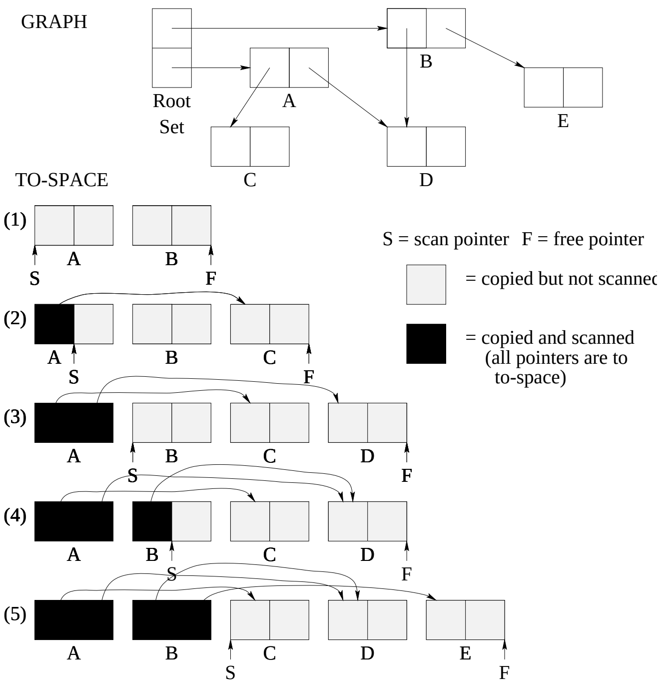



BEFORE COLLECTION




AFTER COLLECTION

## Copying Collection Details




```
gc() {
  int i;
  struct cell *scan = &(space[to_space][0]);
  free = scan;
  for (i = 0 ; i < ROOTS; i++)
    roots[i] = forward(roots[i]);
  while (scan < free) {
    scan->c[0] = forward(scan->c[0]);
    scan->c[1] = forward(scan->c[1]);
    scan++;
  };
  from_space = 1-from_space;
  to_space = 1-to_space;
  end = *(space[from_space][HALFSIZE]);
}

struct cell *forward(struct cell *p) {
  if (p >=&(space[from_space][0]) &&
       p < &(space[from_space][HALFSIZE]))
     if (p->c[0] >= &(space[to_space][0]) &&
         p->c[0] < &(space[to_space][HALFSIZE]))
        return p->c[0];
     else {
       *free = *p;
       p->c[0] = free++;
       return p->c[0];
     }
  else return p;
}
```

```
struct cell {
  struct cell *c[2];
}

struct cell space[2][HALFSIZE];

struct cell *roots[ROOTS];

struct cell *free = &(space[0][0]);
struct cell *end = &(space[0][HALFSIZE]);

int from_space = 0;
int to_space = 1;

struct cell *allocate() {
  if (free == end) { /* no room */
    gc();
    if (free == end) /* still no room */
      die();
  };
  return free++;
}
```

## Comparison

Copying collector does work proportional to amount of **live** data. Asymptotically, this means it does less work than mark and sweep. Let

$A$ = amount of live data
$M$ = total memory size before a collection.

After the collection, there is M-A space left for allocation before the next collection. We can calculate the **amortized cost** per allocated byte as follows:

$C_{M\&S} = \frac{c_1 A + c_2 M}{M - A}$ for some $c_1, c_2$

$C_{COPY} = \frac{c_3 A}{\frac{M}{2} - A}$ for some $c_3$

As $M \to \infty$, $C_{COPY} \to 0$, while $C_{M\&S} \to c_2$.

Of course, real memories aren't infinite, so the values of $c_1, c_2, c_3$ matter, especially if a significant percentage of data are live at collection (since generally $c_3 > c_1$).

**Further Issues**

- Distinguishing pointers from integers.
- Handling records of variable size.
- Finding the root set.
- Avoiding repeated copying of permanently live data.
- Avoiding nasty pauses during collection.

These concerns lead to the study of three important varieties of collectors:

- Conservative collectors.
- Generational collectors.
- Incremental and concurrent collectors.

**Issues in Conservative Collection**

- Some bit patterns that are actually integers, reals, chars, etc. will be mistaken for pointers, so the "records" they "point" to will be treated as live data.
- Accidental pointer identifications can be greatly decreased by careful tests, e.g., must be on a page known to be in the heap, at an appropriate alignment for objects on that page; data at "pointed-to" location must look like a heap header.
- Can further reduce false id's by not allocating on pages whose addresses correspond to data values known to be in use.

Major problems:

- Collector must be able to find registers and stack frames.
- Pointers must not be kept in "hidden" form by mutator code.

**Conservative Collection**

Standard GC algorithms rely on precise identification of pointers.

This is hard in "uncooperative" environments, i.e., when the mutator (and its compiler) are not aware that GC will be performed. This is the normal case for C/C++ programs.

(Hence issue for portable Java implementations based on C, and for native functions.)

Basic problem: the mutator and collector can no longer communicate a root set.

Idea: for any scanning collector to be correct, it's essential that every pointer be found. But for non-moving collectors, it's ok to mistake a non-pointer for a pointer – the worst that happens is that some garbage doesn't get collected.

**Conservative collectors** scan the entire register set and stack of the mutator, and **assume** that anything that **might** be a pointer really is a pointer.

**Object Lifetimes**

Major problem with tracing GC: long-lived data get traced (scanned and/or copied) repeatedly, without producing free space.

**(Weak) Generational Hypothesis:** "Most data die young."

I.e., most records become garbage a short time after they are allocated.

If we equate "age" of an object O is equated with amount of heap allocated since O was allocated, this says that most records become garbage after a small number of other records have been allocated.

Moreover, the longer an object stays live, the more likely it is to remain live in the future.

These are **empirical** properties of many (not necessarily all) languages/programs.

Implication : if you're looking for garbage, it's more likely to be found in recently-allocated data, e.g., in data allocated since the last garbage collection.

**Generational Collection**

Idea: Segregate data by age into **generations**.

• Arrange that the younger generations can be collected independently of the older ones.

• When space is needed, collect the youngest generation first.

• Only collect older generation(s) if space is still needed.

• Should make GC more efficient overall, since less total tracing is performed.

• Should shorten pause times (at least for young generation GCs).

Some variant of generational collection is almost universally used in serious implementations of heavily-allocating languages (LISP, functional languages, Smalltalk, ...)

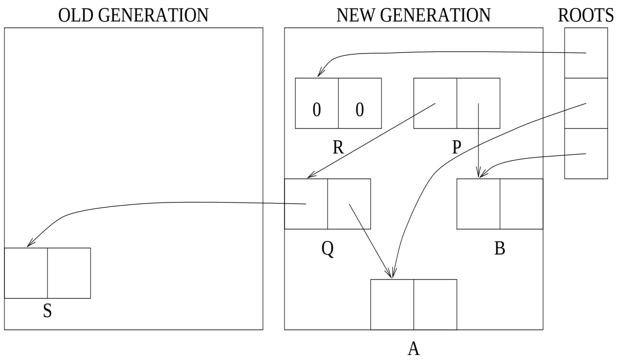
Most generational systems are copying collectors, although mark and sweep variants are possible.

In generational copying collector, data in generation $n$ that are still live after a certain number of gc's (the **promotion threshold**) are copied into generation $n+1$ (possibly triggering a collection there).

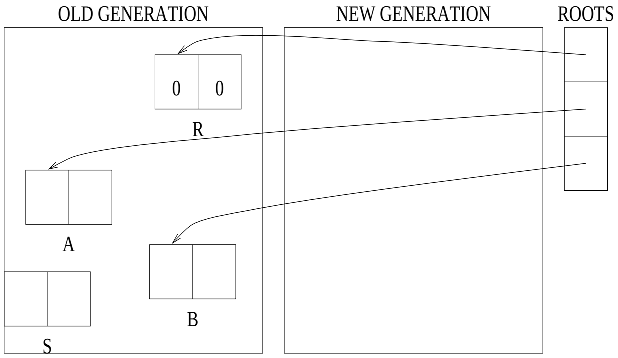
Key problem: finding all the roots that point into generation $n$ without scanning higher generations.

**Example**

Assume 2 generations, promotion threshold = 1. Initial memory configuration after allocation of R:



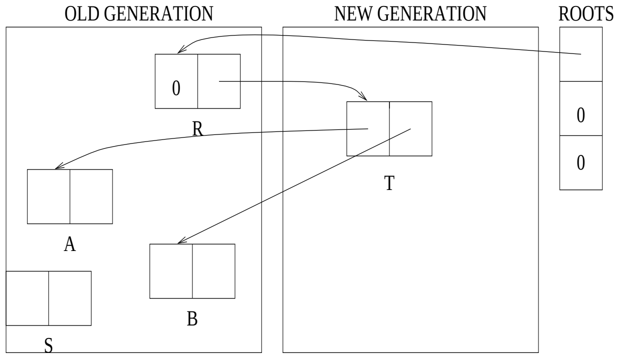

Suppose a GC is now needed:




Note that S is now **tenured** (uncollected garbage).

**Example (continued)**

Now we allocate a new cell T pointed to by R, fill T with pointers to A and B, and zero the root set pointers to A and B.




If a further GC is needed, we **must** follow the inter-generational pointer from R to T.

**Design issues**

Tracking pointers from older generations to younger ones.

• This is primary added cost of generational system.

• Hope there are not too many!

• Maintain **remembered set** of updated memory chunks ("cards"), where chunk size can range from single address to entire page.

• Different tradeoffs in mutator overhead vs. scan time.

Promotion policy?

• Threshold = 1 gives simpler implementation, since no need to record object age, but promotes very young objects.

How many generations?

• Two-generation systems give simpler implementation, but multiple generations are useful if there is a spread of object lifetimes (especially if threshold = 1).

May want separate areas for large, pointer-free, or "immortal" objects.

## Garbage Collection in Java

Sun's JVM uses a "mark and compact" collector

• Compromise between M&S and copying collectors.

• Live data cells are marked.

• Then heap is scanned and live data are slid down to a compact region at the bottom of the heap.

• Extra space costs for forwarding pointers; extra time costs for added traversals.

• Object pointers actually point to a **handle** which in turn points to the real object data. Handles are allocated in their own space, managed by M&S, and never moved. Object data records can be moved just by changing the handle's contents, without altering the object pointer.

In general, copying collection works fine for Java even without handles, since it's easy for interpreter to provide the root set and notice when it's been changed.

## Garbage Collection and Native Code under Java

Interfacing to native code is problematic:

Java objects referenced by native code must not be collected by Java GC until native code is done with them.

Solutions:

• GC can implicitly register all arguments passed to native code (or returned to native code by callbacks into Java), and not collect them until native code finishes.
• References that need to live longer must be explicitly registered (and later unregistered) by native code programmer.

Java objects referenced directly by native code can't be moved.

Solutions:

• Use non-moving collector.
• Pass unmoveable, indirect object references to native code.
• Pin objects passed to native code.
• Make pinned copies of objects passed to native code.

Native code data objects pointed to by Java objects that get GC'ed should be freed.

Solution:

• Use **finalization** routines.