CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Garbage Collection

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.

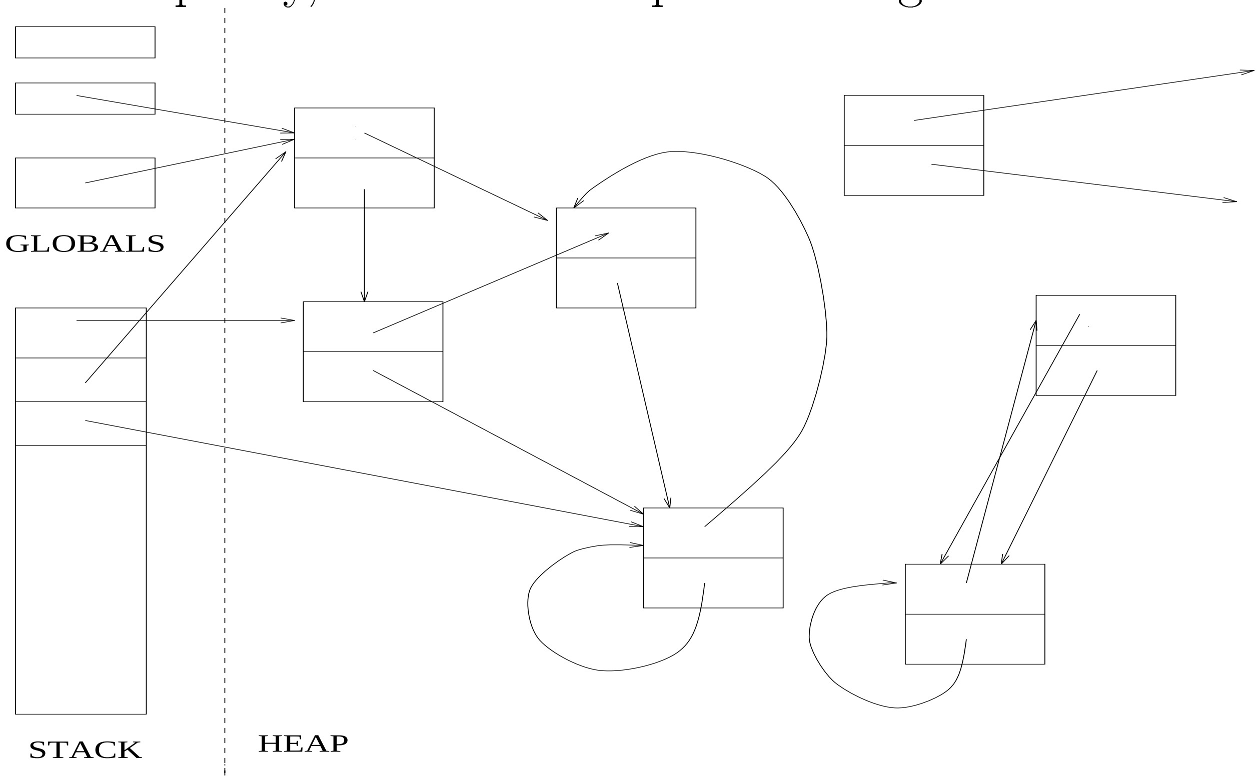
GC may be useful for any language that supports heap allocation, because it obviates the need for explicit deallocation, which is tedious, error-prone, and often non-modular.

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.

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.

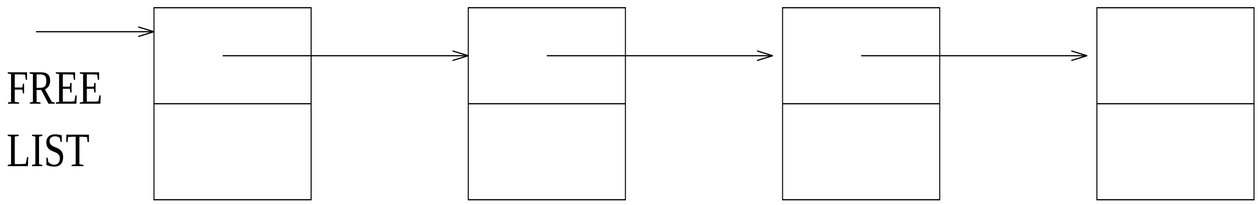
Garbage is the heap memory containing non-live cells. (Note that this is a slightly conservative definition.)

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"

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]);
    }
}
```

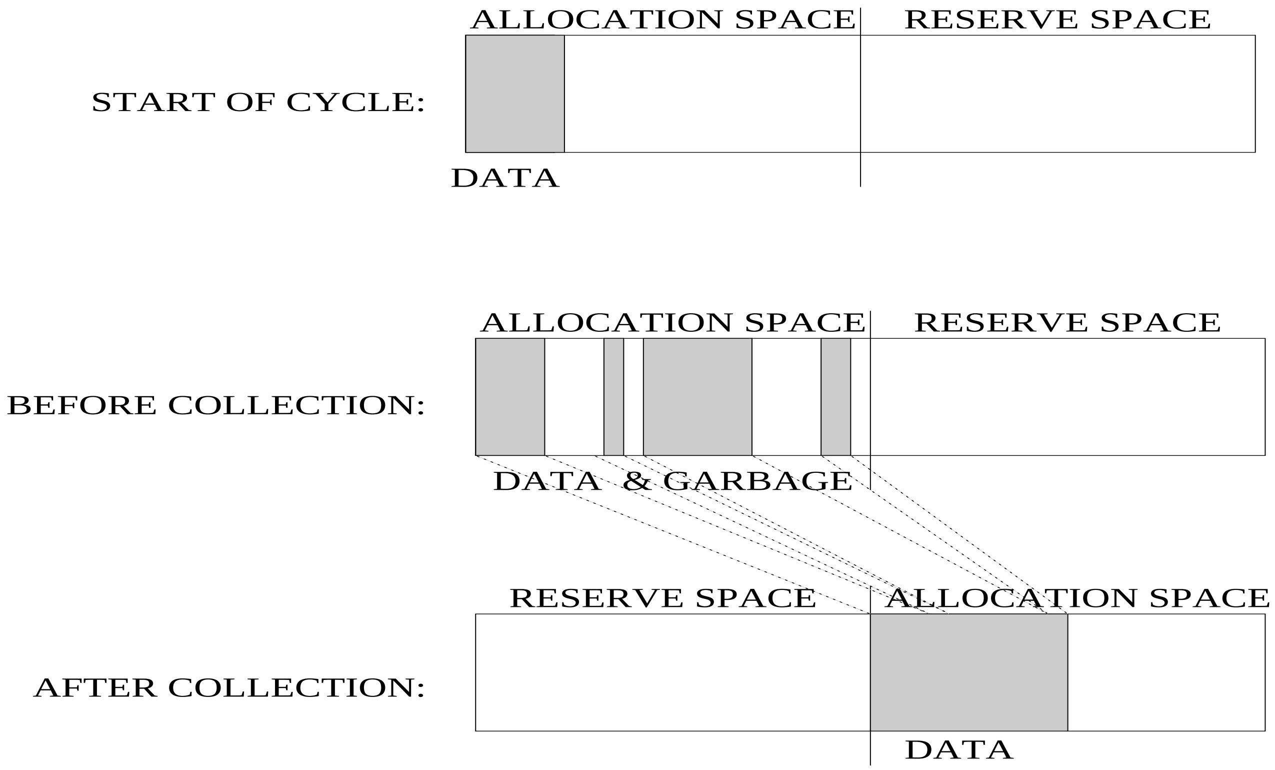
Here `mark` traverses the live data graph in depth-first order, and potentially uses **lots** of stack! A standard trick called **pointer reversal** can be used to avoid needing extra space during the traversal.

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.

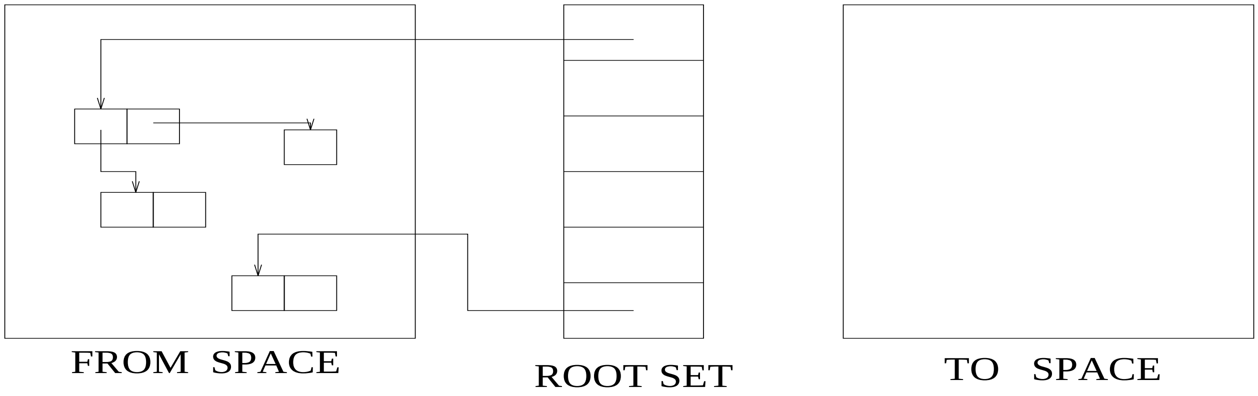
An alternative that solves these problems is **copying collection**. The idea is to divide the available heap into 2 **semi-spaces**. Initially, the allocator uses just one space; when it fills up, the collector **copies** the live data (only) into the other space, and reverses the role of the spaces.



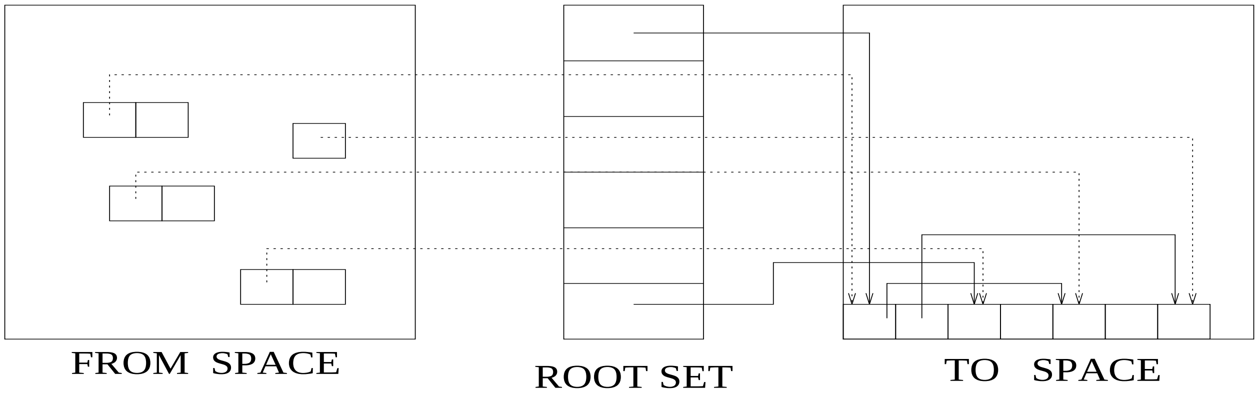
Copying collection must fix up **all** pointers to copied data. To do this, it leaves a **forwarding pointer** in the "from" space after the copy is made.

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.

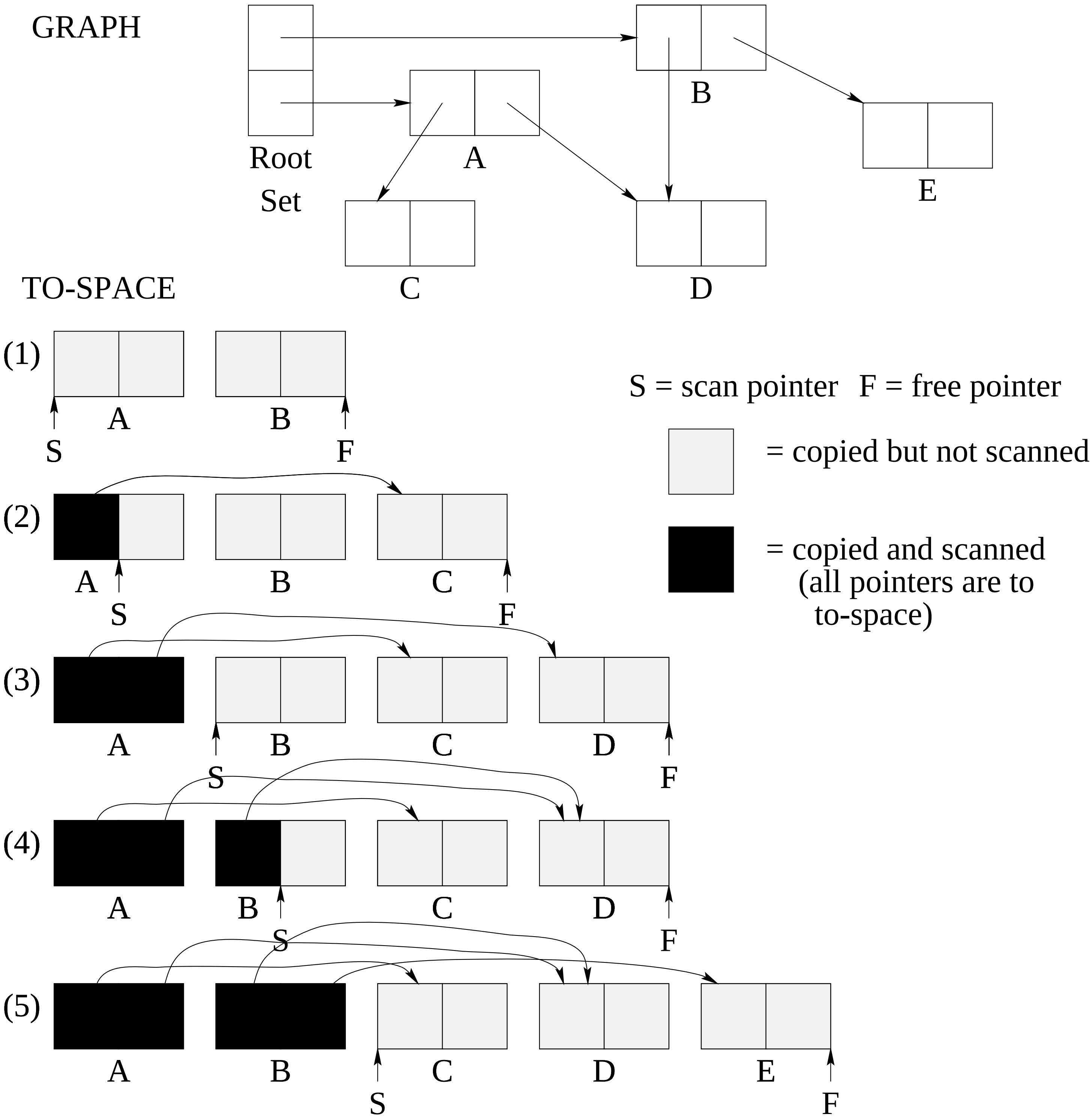


BEFORE COLLECTION



AFTER COLLECTION

Copying Collection Details



```
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++;
}
```

```
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;
}
```

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, espeically if a significant percentage of data is live at collection (since generally $c_3 > c_1$).

Further Issues

- Handling records of variable size.
- Distinguishing pointers from non-pointer data.
- 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 collectors.

Identifying Pointers

A conventional collector needs to distinguish pointers from non-pointers, so that only the former are traced when identifying live data. In most systems, extra **bits** in the data words or headers are used for this.

This implies space and time overhead (to store and access the bits). It also requires that the GC implementation has intimate knowledge of data layout and formats.

Collectors also need to identify the **root set** of pointers into the heap from non-heap locations or registers. Again, this requires intimate knowledge of the layout of stack frames and registers.

If we want to use an existing back-end, e.g., a C compiler, this intimate knowledge may not be available.

But note that if we use a non-moving collector (like M&S), it is not strictly necessary to distinguish pointers from non-pointers precisely: it suffices to treat as pointers any **superset** of the true pointer set.

If the collector accidentally treats a non-pointer as a pointer, it will simply mean that some non-address bit pattern is treated as an address, and the data at that address will be (perhaps erroneously) treated as live.

This may cause a **space leak**, because garbage may be treated as if it were live, but it can never cause live data to be treated as garbage.

Conservative Collectors

A collector that treats **all** bit patterns as potential live addresses in this way is called **conservative**.

We can add conservative GC to an existing system (e.g., a C or C++ compiler) that was not designed with GC in mind. The collector simply treats the **entire** register set and stack as potential roots into the heap, and **every** reachable heap location as a further potential pointer.

In practice, the collector applies a series of simple tests to potential pointers that easily filter out most non-pointer values. To be an actual pointer, a bit pattern must represent an address within the current heap, must point to what looks like the header of a legitimate heap record, etc. With various further tricks, we can get the rate of "false positives" very low.

Still, these filtering tests tend to make the collector slow, and the possibility of leaks can't be ruled out. (Also, the whole scheme can be defeated by use of weird pointer arithmetic.)

So conservative collection is normally used only when GC performance is not a crucial issue, or when the convenience factor is overwhelming. There is a very handy off-the-shelf conservative collector package available from Hans Boehm at

`http://www.hpl.hp.com/personal/Hans_Boehm/gc`

Generational Collectors

In many GCed systems, it is an empirical fact that most data die young. Moreover, the longer a particular heap record has been live, the longer it is likely to stay live.

These points suggest that it is a mistake to recompute the entire live data graph on each garbage collection. Instead, we should concentrate our efforts on looking for garbage in the **newest** data, and look at older data relatively infrequently.

Generational collectors use this idea by dividing data up into two or more distinct **generations** based on age-since-allocation. Each generation lives in a separate heap area. An ordinary GC just cleans up the youngest generation; only if that fails to free sufficient memory is the next older generation collected, and so on. The simplest way to keep track of age is to count the number of collections the record has survived; after a certain threshold has been reached, the records is **tenured** into an older generation.

For generational GC to be worthwhile, we must be able collect younger generations without examining older ones. This cannot be quite true, since it might be that the sole pointer to a young-generation record is from an older-generation one. These "backward" pointers don't happen too often, since it is much more common for new records to point to old, but we must keep track of them somehow, which can be expensive. They can only arise when an older record is **updated** to point to a new one, so they are impossible in a purely functional eager language.