

Garbage Collection

Andrew Tolmach

(Slides prepared by Marius Nita.)

1

Motivation

Problems with manual memory management:

- ◆ It is extremely tedious and error-prone.
- ◆ It introduces software engineering issues (Who is supposed to free the memory?)
- ◆ It is by no means intrinsically efficient. (`free` is not free.)
- ◆ In many cases, the costs outweigh the benefits.

3

Hoping to cover

- ◆ Motivation & basics
- ◆ Introduction to three families of collection algorithms:
 - ◆ Reference Counting
 - ◆ Mark & Sweep
 - ◆ Copying Collection
- ◆ Advanced issues and topics

2

Garbage Collection (GC)

- ◆ The automatic reclamation of unreachable memory (aka `garbage`).
- ◆ Universally used for high level languages with closures and complex data structures that are allocated **implicitly**.
- ◆ Useful for any language that supports heap allocation.
- ◆ It removes the need for explicit deallocation (no more `delete` and `free`).
- ◆ Let the GC implementor deal with memory corruption issues once and for all.

4

How does it work?

Typically:

- ◆ The user program (**mutator**) is linked against a library known as the **runtime system (RTS)** (e.g. `libc`).
- ◆ In the RTS resides the **memory allocation service**, which exposes an allocation routine to the user program.
- ◆ When the user program desires more memory, it invokes the allocation routine (e.g. `malloc`).
- ◆ The **allocation service** may then perform a collection to free unused memory before the allocation routine returns.

5

Terminology

Heap: A directed graph whose nodes are dynamically allocated records and whose edges are pointers between nodes. Typically laid out in a contiguous memory space.

Root set: The set of pointers **into** the heap from an external source (e.g. the stack, global variables).

Live data: The set of heap records that are **reachable** by following paths starting at members of the root set.

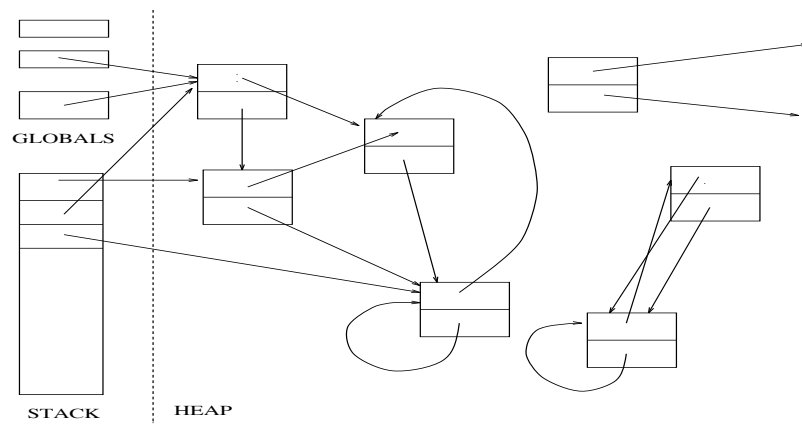
Garbage: The set of heap records that are **not** live.

Note: **reachability** is a conservative liveness estimate.

6

Simple heap model

For simplicity, consider a simple heap of “cons cells:” two-field records, both fields are pointers to other records.



7

Reference counting

Most straightforward collection strategy.

- ◆ Add a **counter** field to each record.
- ◆ Increment a record's counter when taking a new pointer to it.
- ◆ Decrement it when releasing a pointer to it.
- ◆ When it reaches **0**, put the record on a free list.
- ◆ When allocating a new record, check the free list first.

8

Reference counting

- ◆ simple, easy to understand and implement
- ◆ immediate reclamation of storage: no extended periods of time in which the collector might be running while the mutator waits

9

Reference counting

- ◆ simple, easy to understand and implement
- ◆ immediate reclamation of storage: no extended periods of time in which the collector might be running while the mutator waits

However,

9-a

Reference counting

- ◆ simple, easy to understand and implement
- ◆ immediate reclamation of storage: no extended periods of time in which the collector might be running while the mutator waits

However,

- ◆ space overhead (extra field per record)
- ◆ speed overhead (every pointer assignment is wrapped in counter operations and checks)
- ◆ too simple-minded (can't collect cyclic garbage)

9-b

Stop & Collect

There is no need to get rid of garbage if you do not need more space.

A better approach is to wait until the allocator fails to allocate new memory due to lack of space. Then,

- ◆ The collector takes over and frees enough memory to satisfy the allocation request.
- ◆ The allocation now succeeds (or we're out of memory).
- ◆ Control is returned to the mutator.

This is known as [stop & collect](#); mutator is effectively paused while the collector runs.

10

Mark & Sweep

Two phases:

1. **Mark** each live record by tracing all pointers starting at the root.
2. **Sweep** unmarked records (garbage) onto a free list, making them available for reuse. Unmark marked cells at the same time.

Already marked records are ignored in the marking step, so termination is guaranteed.

11

M&S: implementation

```
struct cell {
    int mark:1;
    struct cell *c[2];
};
struct cell *free, heap[HEAPSIZE], *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]);
}
```

12

M&S: implementation

```
struct cell *allocate() {
    struct cell *newCell;
    if (!free) {
        gc();
        if (!free)
            die();
    };
    newCell = free;
    free = free->c[0];
    return newCell;
}
```

/ no more room => */*
/ try gc */*
/ still no more room */*
/ take the first free cell */*
/ off of the free list */*

13

M&S: implementation

```
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)
            /* unmark live data */
            heap[i].mark = 0;
        else {
            /* sweep garbage */
            heap[i].c[0] = free;
            free = &(heap[i]);
        }
}
```

14

M&S: implementation

```
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)
            /* unmark live data */
            heap[i].mark = 0;
        else {
            /* sweep garbage */
            heap[i].c[0] = free;
            free = &(heap[i]);
        }
}
```

Notice anything “strange” about mark?

14-a

M&S: pointer reversal

It's recursive!

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

It could use a **lot** of stack, hence a lot of memory!

A trick called **pointer reversal** can be used to avoid this problem.

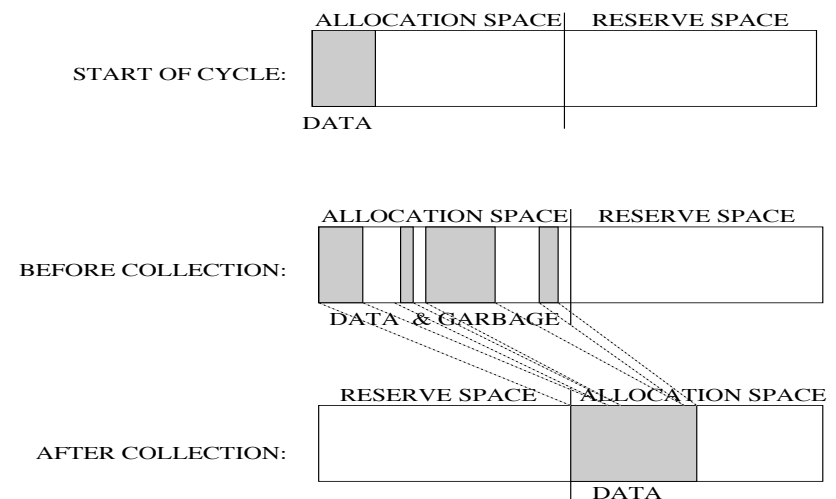
15

Copying collection

- ◆ Divide the heap into **two semi-spaces**.
- ◆ Allocate into one space (the **to-space**).
- ◆ When it fills up, **move** the **live data** to the **from-space** and reverse the roles of the two spaces.
- ◆ Must reassign all pointers as a consequence. (Can't have a copying collector for C!)
- ◆ Inherently compacting – no fragmentation problems, good spatial locality.

16

Copying (cont'd)



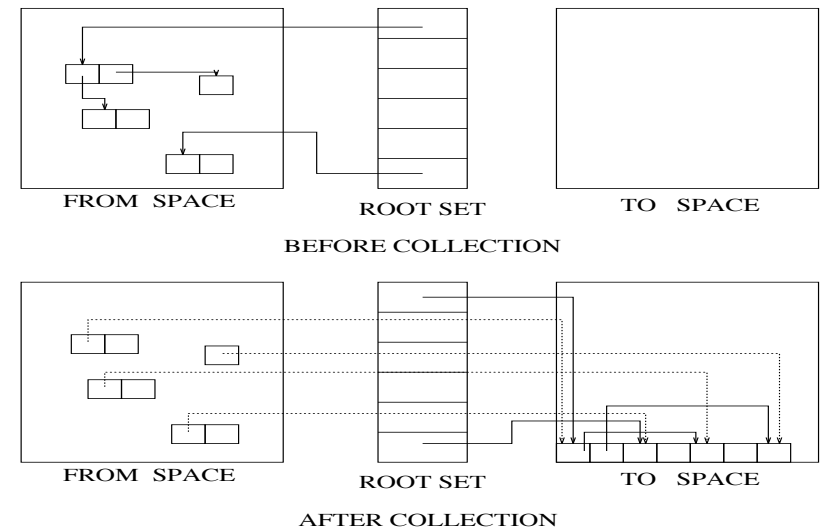
17

Copying (cont'd)

- ◆ The live data is traversed breadth-first using the to-space itself as the queue (Cheney's algorithm).
- ◆ When a record is copied, a **forwarding pointer** pointing to the new location is left in the original.
- ◆ Subsequent attempts to forward that same record will immediately observe the forwarding pointer.

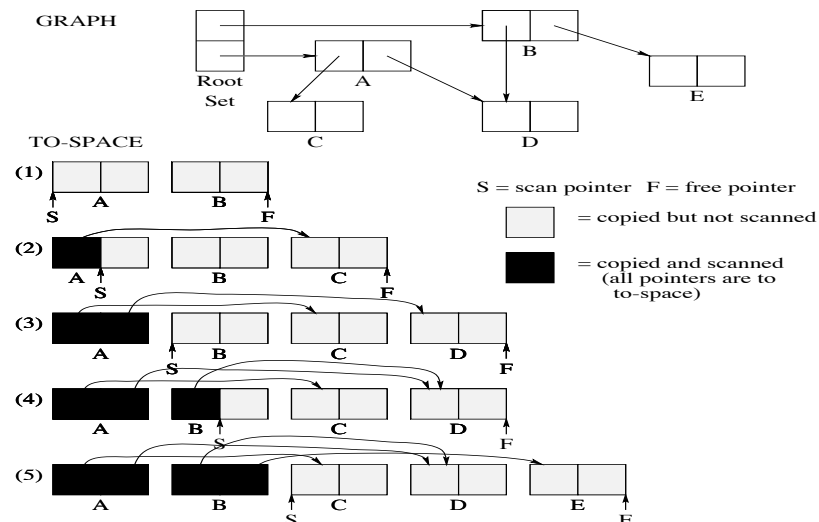
18

Copying (cont'd)



19

Copying: details



20

Copying: implementation

```
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++;
}
```

21

Copying: implementation

```
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]);
}
```

22

Conclusions: M&S

Pros:

- ◆ Big win is that it can use the **whole heap** for allocation.
- ◆ Works well in systems with large amounts of live data – many long lived objects.

Cons:

- ◆ Fragmentation is a real problem.
- ◆ Allocation can be expensive in a heavily fragmented heap.
- ◆ Potential spatial locality issues, bad cache behavior.

24

Copying: implementation

```
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;
}
```

23

Conclusions: Copying

Pros:

- ◆ Simple to understand and implement.
- ◆ Allocation is very fast: contiguous free memory.
- ◆ Good locality, favorable effect on cache behavior.

Cons:

- ◆ It can use only **half** the heap space for allocation – a real concern in systems with limited memory.
- ◆ Poor performance in systems with large amounts of live data.

25

Further Issues

- ◆ Distinguishing pointers from integers.
- ◆ Handling records of various sizes, arrays.
- ◆ Finding and passing the root set.
- ◆ Avoiding unnecessary scanning of long-lived data.
- ◆ Minimizing collection pauses.
- ◆ Improving memory utilization.

These lead to study of other varieties of collectors:

conservative, generational, incremental, compacting, etc.