

CSCI-1200 Computer Science II — Fall 2008

Lecture 24 — Garbage Collection

Review from Lecture 23

- Inheritance is a relationship among classes. Examples: bank accounts, polygons, stack & list.
- Class Hierarchy & Is-A/Has-A/As-A relationships among classes
- Polymorphism allows containers storing objects of different derived types

Today's Class: Garbage Collection

- **What is *garbage*?** Not everything sitting in memory is useful. Garbage is anything that cannot have any influence on the future computation.
- With C++, the programmer is expected to perform *explicit memory management*. You must use `delete` when you are done with dynamically allocated memory (which was created with `new`).
- In Java, and other languages with “garbage collection”, you are not required to explicitly de-allocate the memory. The system automatically determines what is garbage and returns it to the available pool of memory. Certainly this makes it easier to learn to program in these languages, but *automatic memory management* does have performance and memory usage disadvantages.
- Today we'll overview 3 basic techniques for automatic memory management.

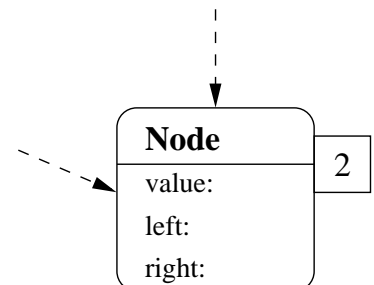
24.1 The Node class

- For our discussion today, we'll assume that all program data is stored in dynamically-allocated instances of the following simple class. This class can be used to build linked lists, trees, and graphs with cycles:

```
class Node {
public:
    Node(char v, Node* l, Node* r) : value(v), left(l), right(r) {}
    char value;
    Node* left;
    Node* right;
};
```

24.2 Garbage Collection Technique #1: Reference Counting

1. Attach a *counter* to each Node in memory.
2. When a new pointer is connected to that Node, increment the counter.
3. When a pointer is removed, decrement the counter.
4. Any Node with `counter == 0` is garbage and is available for reuse.



24.3 Reference Counting Exercise

- Draw a “box and pointer” diagram for the following example, keeping a “reference counter” with each Node.

```
Node *a = new Node('a', NULL, NULL);
Node *b = new Node('b', NULL, NULL);
Node *c = new Node('c', a, b);
a = NULL;
b = NULL;
c->left = c;
c = NULL;
```

- Is there any garbage?

24.4 Memory Model Exercise

- In memory, we pack the `Node` instances into a big array. In the toy example below, we have only enough room in memory to store 8 `Nodes`, which are addressed 100 → 107. 0 is a NULL address.
- For simplicity, we'll assume that the program uses only one variable, `root`, through which it accesses all of the data. Draw the box-and-pointer diagram for the data accessible from `root = 105`.

address	100	101	102	103	104	105	106	107
value	a	b	c	d	e	f	g	h
left	0	0	100	100	0	102	105	104
right	0	100	103	0	105	106	0	0

root: 105

- What memory is garbage?

24.5 Garbage Collection Technique #2: Stop and Copy

1. Split memory in half (*working memory* and *copy memory*).
2. When out of working memory, stop computation and begin garbage collection.
 - (a) Place `scan` and `free` pointers at the start of the copy memory.
 - (b) Copy the `root` to copy memory, incrementing `free`. Whenever a node is copied from working memory, leave a *forwarding address* to its new location in copy memory in the left address slot of its old location.
 - (c) Starting at the `scan` pointer, recursively copy the left and right pointers. Look for their locations in working memory. If the node has already been copied (i.e., it has a forwarding address), update the reference. Otherwise, copy the location (as before) and update the reference.
 - (d) Repeat until `scan == free`.
 - (e) Swap the roles of the working and copy memory.

24.6 Stop and Copy Exercise

Perform stop-and-copy on the following with `root = 105`:

	WORKING MEMORY							
address	100	101	102	103	104	105	106	107
value	a	b	c	d	e	f	g	h
left	0	0	100	100	0	102	105	104
right	0	100	103	0	105	106	0	0

	COPY MEMORY							
address	108	109	110	111	112	113	114	115
value								
left								
right								

root: 105
scan:
free:

24.7 Garbage Collection Technique #3: Mark-Sweep

1. Add a mark bit to each location in memory.
2. Keep a free pointer to the head of the free list.
3. When memory runs out, stop computation, clear the mark bits and begin garbage collection.
4. Mark
 - (a) Start at the root and follow the accessible structure (keeping a *stack* of where you still need to go).
 - (b) Mark every node you visit.
 - (c) Stop when you see a marked node, so you don't go into a cycle.
5. Sweep
 - (a) Start at the end of memory, and build a new free list.
 - (b) If a node is unmarked, then it's garbage, so hook it into the free list by chaining the left pointers.

24.8 Mark-Sweep Exercise

Let's perform Mark-Sweep on the following with `root = 105`:

address	100	101	102	103	104	105	106	107
value	a	b	c	d	e	f	g	h
left	0	0	100	100	0	102	105	104
right	0	100	103	0	105	106	0	0
marks								

root: 105
free:
stack:

24.9 Garbage Collection Comparison

• Reference Counting:

- + fast and incremental
- can't handle cyclical data structures!
- ? requires ~33% extra memory (1 integer per node)

• Stop & Copy:

- requires a long pause in program execution
- + can handle cyclical data structures!
- requires 100% extra memory (you can only use half the memory)
- + runs fast if most of the memory is garbage (it only touches the nodes reachable from the root)
- + data is clustered together and memory is "de-fragmented"

• Mark-Sweep:

- requires a long pause in program execution
- + can handle cyclical data structures!
- + requires ~1% extra memory (just one bit per node)
- runs the same speed regardless of how much of memory is garbage. It must touch all nodes in the mark phase, and must link together all garbage nodes into a free list.

```

#include <iostream>
using namespace std;
#define CAPACITY 16 // size of memory available for this process
#define OFFSET 100 // first valid address for this process
#define MY_NULL 0
typedef int Address;

// =====
class Node {
public:
    Node() { value='?'; left=-1; right=-1; } // initialized with "garbage" values
    char value;
    Address left;
    Address right;
};

// =====
class Memory {
public:
    Memory() { root = MY_NULL; }

    // Return the node corresponding to a particular address
    Node& operator[](Address addr);
    // allocate a new node
    virtual Address my_new(char value, Address l, Address r) = 0;
    // print the contents of memory
    friend ostream& operator<<(ostream &ostr, Memory &m);
    virtual void print_availability(ostream &ostr) = 0;

    // the user must set this value such that all useful memory is
    // reachable starting from root (NOTE: publicly accessible)
    Address root;
protected:
    Node memory[CAPACITY]; // total machine memory
};

// =====
class CPP_Memory : public Memory {
public:
    CPP_Memory() {
        for (int i=0; i < CAPACITY; i++)
            available[i] = true;
        next = 0; }
    Address my_new(char v, Address l, Address r);
    void my_delete(Address addr); // explicit memory management
    void print_availability(ostream &ostr);
protected:
    bool available[CAPACITY]; // which cells have been allocated
    int next; // where to start looking for the next available node
};

// =====
class StopAndCopy_Memory : public Memory {
public:
    StopAndCopy_Memory() {
        partition_offset = 0;
        next = 0; }
    Address my_new(char v, Address l, Address r);
    void collect_garbage(); // automatic memory management
    void print_availability(ostream &ostr);
protected:
    void copy_help(Address &old_address);
    int partition_offset; // which half of the memory is active
    int next; // next available node
};

```

```

#include <assert.h>
#include "memory.h"

// =====
Node& Memory::operator[](Address addr) {
    // Return the node corresponding to a particular address
    if (addr == MY_NULL) {
        std::cerr << "ERROR: NULL POINTER EXCEPTION!" << endl; exit(1); }
    if (addr < OFFSET || addr >= OFFSET+CAPACITY) {
        cerr << "ERROR: SEGMENTATION FAULT!" << endl; exit(1); }
    return memory[addr-OFFSET];
}

ostream& operator<<(ostream &ostr, Memory &m) {
    ostr << "root-> " << m.root << endl;
    for (int i = 0; i < CAPACITY; i++) {
        ostr.width(4); ostr << i+OFFSET << " "; }
    ostr << endl;
    for (int i = 0; i < CAPACITY; i++) {
        ostr << " "; ostr.width(1); ostr << m.memory[i].value << " "; }
    ostr << endl;
    for (int i = 0; i < CAPACITY; i++) {
        ostr.width(4); ostr << m.memory[i].left << " "; }
    ostr << endl;
    for (int i = 0; i < CAPACITY; i++) {
        ostr.width(4); ostr << m.memory[i].right << " "; }
    ostr << endl;
    m.print_availability(ostr);
    return ostr;
}

// =====
Address CPP_Memory::my_new(char v, Address l, Address r) {
    // starting at next, walk through the memory to find the first
    // available address. If we walk in a circle, we're out of memory.
    for (int i=0; i < CAPACITY; i++, next++) {
        next %= CAPACITY;
        if (available[next] == true) {
            available[next] = false;
            memory[next].value = v;
            memory[next].left = l;
            memory[next].right = r;
            return OFFSET + next++;
        }
    }
    cerr << "ERROR: OUT OF MEMORY!" << endl; exit(1);
}

void CPP_Memory::my_delete(Address addr) {
    // makes a node available for re-use
    if (addr == MY_NULL) return; // deleting a NULL pointer is not an error in C++
    if (addr < OFFSET || addr >= OFFSET+CAPACITY) {
        cerr << "ERROR: SEGMENTATION FAULT!" << endl; exit(1); }
    if (available[addr-OFFSET] == true) {
        cerr << "ERROR: CANNOT DELETE MEMORY THAT IS NOT ALLOCATED!" << endl; exit(1); }
    available[addr-OFFSET] = true;
}

void CPP_Memory::print_availability(ostream &ostr) {
    // print "FREE" or "used" for each node
    for (int i = 0; i < CAPACITY; i++) {
        (available[i]) ? ostr << "FREE " : ostr << "used "; }
    ostr << endl;
}

```

```

// =====
Address StopAndCopy_Memory::my_new(char v, Address l, Address r) {
    // if we are out of memory, collect garbage
    if (next == partition_offset+CAPACITY/2) {
        collect_garbage();
        // update the addresses (since memory has been shuffled!)
        if (l != MY_NULL) l = memory[l-OFFSET].left;
        if (r != MY_NULL) r = memory[r-OFFSET].left;
    }
    // if we are still out of memory, we can't continue
    if (next == partition_offset+CAPACITY/2) {
        cerr << "ERROR: OUT OF MEMORY!" << endl; exit(1); }
    // assign the next available node
    memory[next].value = v;
    memory[next].left = l;
    memory[next].right = r;
    return OFFSET + next++;
}

void StopAndCopy_Memory::print_availability(ostream &ostr) {
    // print "FREE" or "used" for each node in the current partition
    for (int i = 0; i < CAPACITY; i++) {
        if (i >= next && i < partition_offset+CAPACITY/2)
            ostr << "FREE ";
        else if (i >= partition_offset && i < partition_offset+CAPACITY/2)
            ostr << "used ";
        else // print nothing for the other half of memory
            ostr << " "; }
    ostr << endl;
}

void StopAndCopy_Memory::collect_garbage() {
    // switch to the other partition
    partition_offset = (partition_offset == 0) ? CAPACITY/2 : 0;
    // scan & next start at the beginning of the new partition
    Address scan;
    next = scan = partition_offset;
    // copy the root
    copy_help(root);
    // scan through the newly copied nodes
    while (scan != next) {
        // copy the left & right pointers
        copy_help(memory[scan].left);
        copy_help(memory[scan].right);
        scan++;
    }
}

void StopAndCopy_Memory::copy_help(Address &old) {
    // do nothing for NULL Address
    if (old == MY_NULL) return;
    // look for a valid forwarding address to the new partition
    int forward = memory[old-OFFSET].left;
    if (forward-OFFSET >= partition_offset &&
        forward-OFFSET < partition_offset+CAPACITY/2) {
        // if already copied, change pointer to new address
        old = forward;
        return;
    }
    // otherwise copy it to a free slot and leave a forwarding address
    memory[next] = memory[old-OFFSET];
    memory[old-OFFSET].left = next+OFFSET;
    old = next+OFFSET;
    next++;
}

```