CSCI-1200 Computer Science II — Spring 2006
Lecture 25 — Garbage Collection & Course Summary

Final Exam General Information

- The final exam will be held **Thursday, May 11th, 2006, 6:30-9:30pm, DCC 308**. A makeup exam will be offered if required by the RPI rules regarding final exam conflicts -*OR*- if a written excuse from the Dean of Students office is provided. Contact Professor Cutler ASAP if you require a makeup exam.
- Coverage: Lectures 1-25, Labs 1-14, HW 1-8.
- Closed-book and closed-notes *except for 2 sheets of 8.5x11 inch paper (front & back) that may be handwritten or printed*. Computers, cell-phones, palm pilots, calculators, PDAs, etc. are not permitted and must be turned off.
- **All students must bring their Rensselaer photo ID card.**
- The best thing you can do to prepare for the final is practice. Try the review problems with pencil & paper first. Then practice programming (with a computer) the exercises and other exercises from lecture, lab, homework and the textbook.

Review from Lecture 24

- 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 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. 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.

25.1 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.

25.2 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;
```

25.3 Memory Model Exercise

In memory, we pack the nodes into a big array. Draw the box-and-pointer diagram starting 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

25.4 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`. For each node copied, put the *forwarding address* in the left address slot.
   (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.

25.5 Stop and Copy Exercise

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

	WORKING MEMORY								COPY MEMORY							
address	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115
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								

25.6 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.

25.7 Mark-Sweep Exercise

Let's perform Mark-Sweep on the following with → 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								

```
#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;
  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;
};

// ============================================================================
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"

// Return the node corresponding to a particular address
Node& Memory::operator[](Address addr) {
  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(); }
  // 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++;
}
```

25.8 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.

Course Summary

- Approach any problem by studying the requirements carefully, playing with hand-generated examples to understand them, and then looking for analogous problems that you already know how to solve.
- The standard library offers container classes and algorithms that simplify the programming process and raise your conceptual level of thinking in designing solutions to programming problems. Just think how much harder some of the homework problems would have been without generic container classes.
- When choosing between algorithms and between container classes (data structures) you should consider:
  - efficiency,
  - naturalness of use, and
  - ease of programming.
- Use classes with well-designed public and private member functions to encapsulate sections of code.
- Writing your own container classes and data structures usually requires building linked structures and managing memory through the big three:
  - copy constructor,
  - assignment operator, and
  - destructor.
- When testing and debugging:
  - Test one function and one class at a time,
  - Figure out what your program actually does, not what you wanted it to do,
  - Use small examples and boundary conditions when testing, and
  - Find and fix the first mistake in the flow of your program before considering other apparent mistakes.
- Above all, remember the excitement and satisfaction of developing a deep, computational understanding of a problem and turning it into a program that realizes your understanding flawlessly.