CSCI-1200 Data Structures — Fall 2015
Lecture 27 — Garbage Collection & Smart Pointers

Announcements

- Please fill out your course evaluations!
- Those of you interested in becoming an undergraduate mentor for Data Structures, or another CSCI course:
  - Speak to your graduate lab TA and ask him/her to recommend you for the position.
- Turn in your class inheritance diagram for HW10 tomorrow during your normal lab section.
  - It will be graded for accuracy & neatness (no scribbles or edge crossings).
  - Make a copy of your diagram so you can use it as you finish your HW10 implementation.
- The final exam pratice problems are posted on the calendar.
  - If we get at least 85% response to the course evaluations, we will post the solutions early.

Review from Lecture 26

- Error handling strategies
- Basic exception mechanisms: `try/throw/catch`
- Functions & exceptions, constructors & exceptions

Today's Lecture

- What is Garbage?
- 3 Garbage Collection Techniques
- Smart Pointers

27.1 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.

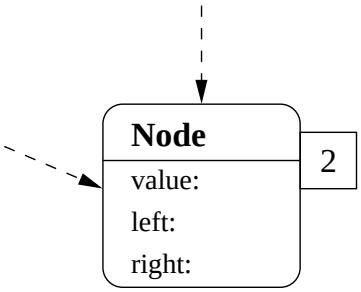
  Certainly this makes it easier to learn to program in these languages, but *automatic memory management* does have performance and memory usage disadvantages.
- Today we'll overview 3 basic techniques for automatic memory management.

27.2 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;
};
```

27.3 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.

27.4 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?

27.5 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 `Node`s, 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?

27.6 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, process the left and right pointers of each node. 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.

27.7 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:
```

27.8 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.

27.9 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:
```

- **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.

27.10 Practical Garbage Collection Methodology in C++: Smart Pointers

- Garbage collection looks like an attractive option both when we are quickly drafting a prototype system and also when we are developing big complex programs that process and rearrange lots of data.
- Unfortunately, general-purpose, invisible garbage collection isn't something we can just tack onto C++, an enormous beast of a programming language (but that doesn't stop people from trying!). So is there anything we can do? Yes, we can use *Smart Pointers* to gain some of the features of garbage collection.
- Some examples below are modified from these nice online references:
  http://ootips.org/yonat/4dev/smart-pointers.html
  http://www.codeproject.com/KB/stl/boostsmartptr.aspx
  http://en.wikipedia.org/wiki/Smart_pointer
  http://www.boost.org/doc/libs/1_48_0/libs/smart_ptr/smart_ptr.htm

27.11 What's a Smart Pointer?

- The goal is to create a widget that works just like a regular pointer most of the time, except at the beginning and end of its lifetime. The syntax of how we construct smart pointers is a bit different and we don't need to obsess about how & when it will get deleted (it happens automatically).
- Here's one flavor of a smart pointer (much simplified from STL):

```
template <class T>
class auto_ptr {
public:
  explicit auto_ptr(T* p = NULL) : ptr(p) {}  /* prevents cast/conversion */
  ~auto_ptr()      { delete ptr; }
  T& operator*()   { return *ptr; }
  T* operator->()  { return ptr; }            /* fakes being a pointer */
private:
  T* ptr;
};
```

- And let's start with some example code without smart pointers:

```
void foo() {
  Polygon* p(new Polygon(/* stuff */));
  p->DoSomething();
  delete p;
}
```

- Here's how we can re-write the same example with our `auto_ptr`:

```
void foo() {
  auto_ptr<Polygon> p(new Polygon(/* stuff */);
  p->DoSomething();
}
```

- We don't have to call delete! There's no memory leak or memory error in this code. Awesome!

27.12 So, What are the Advantages of Smart Pointers?

- Smart pointers are magical. They allow us to be lazy! All the time we spent learning about dynamically allocated memory, copy constructors, destructors, memory leaks, and segmentation faults this semester was unnecessary. *Whoa... that's overstating things more than slightly!!*
- With practice, smart pointers can result in code that is more concise and elegant with fewer errors. *Why? ...*
- With thoughtful use, smart pointers make it easier to follow the principles of RAII and make code *exception safe*. In the `auto_ptr` example above, if `DoSomething` throws an exception, the memory for object `p` will be properly deallocated when we leave the scope of the `foo` function! This is *not* the case with the original version.
- The STL `shared_ptr` flavor implements reference counting garbage collection. Awesome[2]!
- They play nice with STL containers. Say you make an `std::vector` (or `std::list`, or `std::map`, etc.) of regular pointers to Polygon objects, `Polygon*` (especially handy if this is a polymorphic collection of objects!). You allocate them all with `new`, and when you are all finished you must remember to *explicitly* deallocate each of the objects.

```
class Polygon { /*...*/ };
class Triangle : public Polygon { /*...*/ };
class Quad : public Polygon { /*...*/ };

std::vector<Polygon*> polys;
polys.push_back(new Triangle(/*...*/));
polys.push_back(new Quad(/*...*/));

for (unsigned int i = 0; i < polys.size(); i++) {
  delete polys[i];
}
polys.clear();
```

  In contrast with smart pointers they will be deallocated automagically!

```
std::vector<shared_ptr<Polygon> > polys;

polys.push_back(shared_ptr<Polygon>(new Triangle(/*...*/)));
polys.push_back(shared_ptr<Polygon>(new Quad(/*...*/)));

polys.clear();  // cleanup is automatic!
```

27.13 Why are Smart Pointers Tricky?

- Smart pointers **do not alleviate the need to master pointers, basic memory allocation & deallocation, copy constructors, destructors, assignment operators, and reference variables**.
- You can still make mistakes in your smart pointer code that yield the same types of memory corruption, segmentation faults, and memory leaks as regular pointers.
- There are several different flavors of smart pointers to choose from (developed for different uses, for common *design patterns*). You need to understand your application *and* the different pitfalls when you select the appropriate implementation.

27.14 What are the Different Types of Smart Pointers?

Like other parts of the C++ standard, these tools are still evolving. The different choices reflect different *ownership semantics* and different *design patterns*. There are some smart pointers in STL, and also some in Boost (a C++ library that further extends the current STL). A quick overview:

- `auto_ptr`
  When "copied" (copy constructor), the new object takes ownership and the old object is now empty. *Deprecated in new C++ standard.*
- `unique_ptr`
  Cannot be copied (copy constructor not public). Can only be "moved" to transfer ownership. Explicit ownership transfer. *Intended to replace* `auto_ptr`. std::unique_ptr has memory overhead only if you provide it with some non-trivial deleter. It has time overhead only during constructor (if it has to copy the provided deleter) and during destructor (to destroy the owned object).

- `scoped_ptr` (Boost)
  "Remembers" to delete things when they go out of scope. Alternate to `auto_ptr`. Cannot be copied.

- `shared_ptr`
  Reference counted ownership of pointer. Unfortunately, circular references are still a problem. Different subflavors based on where the counter is stored in memory relative to the object, e.g., `intrusive_ptr`, which is more memory efficient. std::unique_ptr has memory overhead only if you provide it with some non-trivial deleter. It has time overhead in constructor (to create the reference counter), in destructor (to decrement the reference counter and possibly destroy the object) and in assignment operator (to increment the reference counter).

- `weak_ptr`
  Use with `shared_ptr`. Memory is destroyed when no more `shared_ptr`s are pointing to object. So each time a `weak_ptr` is used you should first "lock" the data by creating a `shared_ptr`.

- `scoped_array` and `shared_array` (Boost)