CSE 303:
Concepts and Tools for Software Development

Dan Grossman

Spring 2005

Lecture 25— Memory-Management Idioms

No tools or rule today

Review: Java and C memory-management rules

Idioms for memory-management:

- Garbage collection
- Unique pointers
- Reference Counting
- Arenas (a.k.a. regions)

Generalization: Same "problems" with file-handles, network-connections, Java-style iterators, ...

Java rules

- Space for local variables lasts until end of method-call, but no problem because cannot get pointer into stack
- All "objects" are in the heap; they conceptually live forever.
  - Really get reclaimed when they are *unreachable* (from a stack variables or global variable).
  - Static fields are global variables.

Consequences:

- You rarely think about memory-management.
- You *can* run out of memory without needing to (e.g., long dead list in a global), but you still get a *safe* exception.
- No dangling-pointer dereferences!
- Extra behind-the-scenes space and time for doing the collection.

C rules

- Space for local variables lasts until end of function-call, may lead to dangling pointers into the stack.
- Objects into the heap live until `free(p)` is called, where `p` points to the beginning of the object.
- Therefore, unreachable objects can never be reclaimed.
- `malloc` returns `NULL` if it cannot find space.
- TFMSTCOF[a]:
  1. Calling `free` with a stack pointer or middle pointer.
  2. Calling `free` twice with the same pointer.
  3. Dereferencing a pointer to an object that has been freed.
- Usually 1–2 screw up the `malloc`/`free` library and 3 screws up an application when the space is being used for another object.

[a] The Following May Set The Computer On Fire

Garbage Collection for C

Yes, there are garbage collectors for C (and C++)!
`http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

- redefines `free` to do nothing
- unlike a Java GC, *conservatively* thinks an `int` might be a pointer.

Questions to ask yourself in any application:

- Why do I want manual memory management?
- Why do I want C?

Good (and rare!) answers against GC: Tight control over performance; even short pauses unacceptable; need to free reachable data.

Good (and rare!) answers for C: Need tight control over data representation and/or pointers into the stack.

Other answer for C: need easy interoperability with lots of existing code

Analogous situations

The manual memory-management challenge boils down to: For each object, you might have multiple pointers but you need to call `free`:

- exactly once
- not too late (space consumption)
- not too early (dangling-pointer derereferences)

Even if you have GC for memory, you'll probably have the same thing for other "interfaces".

Example: Java OutputStream (cannot call `write` after `close`).

Example: `complete_input` in your homework.

In general, a library "wants to know" when you're done with something, and it's up to you to make a timely and accurate report.

Why is it hard?

This is not really the problem:

```
free(p);
...
p->x = 37; // dangling-pointer dereference
```

These are:

```
p = q; // if p was last reference and q!=p, leak!
```

```
lst1 = append(lst1,lst2);
free_list(lst2); // user function, assume it
                 // frees all elements of list
length(lst1); // dangling-pointer dereference
              // if append does not copy!
```

There are an infinite number of *safe idioms*, but only a few are known to be simple enough to get right in large systems...

Idiom 1: Unique Pointers

Ensure there is exactly one pointer to an object. Then you can call `free` on the pointer whenever, and set the pointer's location to `NULL` to be "extra careful".

Furthermore, you *must* free pointers before losing references to them.

Hard parts:

1. May make no sense for the data-structure/algorithm.
2. May lead to extra space because sharing is not allowed.
3. Easy to lose references (e.g., `p=q;`).
4. Easy to duplicate references (e.g., `p=q;`) (must follow with `q=NULL;`).
5. A pain to return unfreed function arguments.

Relaxing Uniqueness

This is just too painful:

```
struct T { int*x; int*y; };
void g(int *p1, int*p2) {
  printf("%d %d'',*p1,*p2);
  struct T ans;
  ans.x = p1;
  ans.y = p2;
  return ans;
}
void f(int *p1, int*p2) {
  struct T ptrs = g(p1,p2);
  p1 = ptrs.x; p2 = ptrs.y;
  ...
  free(p1);
  free(p2);
}
```

Relaxing Uniqueness

Instead, you allow "unconsumed" pointers:

- Callee won't free them
- They will be unique again when function exits

More often what you want, but changes the contract:

- Callee must *not* free
- Callee must not store the pointer anywhere else (in a global, in a field of an object pointed to by another pointer, etc.)

Reference-Counting

Store with an object how many pointers there are to it. When it reaches 0, call `free`.

- Literally a field in each pointed to object.
- `p=q;` becomes `decr_count(p); p=q; incr_count(p);`
- In practice, you can "be careful" and omit ref-count manipulation for temporary variables.

```
struct Example { int count; ... };
void decr_count(struct Example * p) {
  --p->count;
  if(p->count == 0)
    free(p);
}
void incr_count(struct Example * p) { ++p->count; }
```

Reference-Counting Problems

1. Avoids freeing too early, but one lost reference means a leak.
2. Reference-count maintenance expensive and error-prone (C++ tricks can automate to some degree).
3. CYCLES!

Cycle detection looks a lot like GC.

(Actually, there's this cool folk-algorithm for detecting if a list is cyclic.)

Arenas (a.k.a. regions)

Rather than track each object's "liveness", track each object's "region" and deallocate a region "all at once".

Revised memory-management interface:

```
typedef struct RgnHandle * region_t;
region_t create_rgn();
void destroy_rgn(region_t);
void * rgn_malloc(region_t,int);
```

So now you "only" have to keep track of a pointer's region and the region's status. (In theory, no simpler? In practice, much simpler!)

Arena Uses

Examples:

- Scratch space for lots of lists with sharing. When you're done, copy out the one answer and destroy the region.
- Callee chooses size, number of objects, aliasing patterns.
  Caller choose lifetime (and passes in a *handle* as an argument).
- You can track handles and inter-region pointers via other means (e.g., reference-counting) while "ignoring" intra-region pointers.

Conclusions

Memory management is difficult; each "general approach" has plusses and minuses.

As with any "design patterns", knowing vocabulary helps communicate, assess trade-offs, and reuse hard-won wisdome.

Key notions: reachability, aliasing, cycles, "escaping (e.g., storing argument in global)". Each approach restricts one of them to some degree.