ECE 590.01
C++ Programming, Data structures, and Algorithms

Garbage Collection
Admin

- **Reading**
  - On class website

- **Homework 4**
  - Due this Friday (April 5)

- **Project**
  - Due April 12
  - In class: 10 min demos to everyone last week
  - Show off what you did
  - "Normal" demos w/ Tas: same week
Talk this week: you should go!

- Todd Austin is giving a talk next week
  - Distinguished lecture series
  - Thurs 3/4 @ 3:15
  - Hudson 125

- Attend if you can
What have we been talking about?

- What did we talk about last time?
What have we been talking about?

• What did we talk about last time?
  • Multiple Inheritance
  • Object Layout
  • Mixins
Now: Garbage Collection

- Some languages (e.g., Java, SML, Scheme) have GC
  - C and C++ do not

- Today:
  - How does GC work?
    - Advantages and dis-advantages
  - Why is it harder in C/C++?
Garbage

- Garbage: any data which cannot affect the outcome of the program
  - Impossible to determine if data is garbage (undecidability)

- Conservative approximation:
  - Data which cannot be reach (no way to name it)
Garbage Creation

- Objects becomes un-reachable when pointers change
  - No pointer points at them anymore

- “Object” in GC terms means “thing that is allocated” may or may not be “Object” in the OOP sense
Garbage Creation

- Objects becomes un-reachable when pointers change
  - No pointer points at them anymore
  - Or only pointers are from un-reachable objects

- "Object" in GC terms means "thing that is allocated" may or may not be "Object" in the OOP sense
Garbage Creation

- Objects become un-reachable when pointers change
  - No pointer points at them anymore
  - Or only pointers are from un-reachable objects
- GC has two tasks
  - Identify “garbage”
  - Re-claim that memory for re-allocation
Garbage Creation

- Reachability has to start somewhere
  - Root set: registers
    - Program can always access pointers in its registers
    - One points at the stack (so all things on the stack are reachable)
  - Not pictured here (off left side)
Reference Counting

- Conceptually simplest approach: Reference Counting
  - Each object tracks how many pointers point at it
  - Update a pointer? Update reference counts
    - Decrement old reference count
    - Increment new reference count
    - Needs to include pointers in registers
Reference Counting

- Reference count goes to 0?
  - Object is un-reachable
  - Re-claim it
Reference Counting

- Reference count goes to 0?
  - Object is un-reachable
  - Re-claim it
  - May make other objects un-reachable: repeat
Reference Counting

- Reference count goes to 0?
  - Object is un-reachable
  - Re-claim it
  - May make other objects un-reachable: repeat
Reference Counting Overheads

- Changing any pointer requires updating ref counts

```c
p = q;

Turns into something like:
if (q != NULL) {q->refc++;}
if (p != NULL) {
    p->refc --;
    if (p->refc == 0) {
        collect(p); //recursively gc’s p
    }
}

p = q;

Smart compiler: may be able to mitigate this some...
**Reference Counting difficulty**

- Consider the above structure
  - I.e., some sort of graph
- Reference from root set is lost
Reference Counting difficulty

Consider the above structure
  • I.e., some sort of graph

Reference from root set is lost
  • Every item is referenced by some other item...
  • But none of those items are reachable!
Mark and Sweep

• Different approach: Mark and Sweep

• Step 0:
  • GC runs when allocation fails (heap is full)
    • Cleanup heap, then retry allocation

• Step 1: Mark phase
  • Traverse heap (i.e., DFS) starting from root set
  • Encounter an object? It is reachable: mark it

• Step 2: Sweep phase
  • Iterate over all objects in heap (i.e., address order)
  • Un-marked objects? Un-reachable: re-claim.
  • Marked objects? Un-mark for next time.
Mark and Sweep

- Start: all space allocated, not marked
  - Allocated, not marked = orange
Mark and Sweep

- Start: all space allocated, not marked
  - Allocated, not marked = orange

- Mark phase: do DFS from root set
  - Allocated+ Marked = Green
Mark and Sweep

- **Start**: all space allocated, not marked
  - Allocated, not marked = orange

- **Mark phase**: do DFS from root set
  - Allocated+ Marked = Green
Mark and Sweep

- **Sweep phase:**
  - Iterate through all objects
    - Free un-marked objects
    - Un-mark marked objects
Mark and Sweep

• **Sweep phase:**
  • Iterate through all objects
  • Free un-marked objects
  • Un-mark marked objects
Mark and Sweep

- **Sweep phase:**
  - Iterate through all objects
  - Free un-marked objects
  - Un-mark marked objects
Mark and Sweep

- **Sweep phase:**
  - Iterate through all objects
    - Free un-marked objects
    - Un-mark marked objects
Mark and Sweep

- Now we have free spaces in our heap again
  - Re-try failed allocation, and proceed
  - Allocation still fails:
    - Grow heap
    - Or throw exception
M&S Pros and Cons

- **Pros:**
  - Finds/reclaims all un-reachable objects
  - Lower overhead than ref counting

- **Cons:**
  - May have “long pause” for GC of large heap
    - Mark time: $O(L)$ // $L = $ live objects
    - Assume sparse edges in heap
    - Sweep time: $O(H)$ // $H = $ total heap objects
  - Need to add space to objects for mark bit

- **Other:**
  - Must be able to identify start + size of objects
  - Must be able to identify which fields of objects are ptrs
    - Typically add header, which includes mark bit
  - Need space for DFS stack (or clever pointer tricks)
M&S Time

• O(L+H) may not seem bad...
  • Except that we cannot amortize it effectively over more “useful work” by making a larger heap

• Do H+L work
  • Get H-L allocations

• GC work per allocation: \( \frac{H+L}{H-L} \)
Another approach: Stop and Copy

- Setup: Split heap into two halves (semi-spaces)
  - One half is active, the other is idle
- Allocation proceeds sequentially
  - Very fast “malloc” operation: increment pointer by size
Stop and Copy

- Semi-space is full when allocation fails
  - Stop + Copy
    - Trace heap, copying live objects to idle semi-space
Stop and Copy

• Semi-space is full when allocation fails
  • Stop + Copy
    • Trace heap, copying live objects to idle semi-space
Stop and Copy

- Semi-space is full when allocation fails
  - Stop + Copy
    - Trace heap, copying live objects to idle semi-space
  - Put Allocation Pointer at start of idle semi-space
  - Do DFS, copying from active to idle
Stop and Copy

• Semi-space is full when allocation fails
  • Stop + Copy
    • Trace heap, copying live objects to idle semi-space
  • Put Allocation Pointer at start of idle semi-space
  • Do DFS, copying from active to idle
    • Copied objects leave “forwarding pointers”
      • Used to update pointers in objects found later
Stop and Copy

Semi-space is full when allocation fails

• Stop + Copy
  • Trace heap, copying live objects to idle semi-space
• Put Allocation Pointer at start of idle semi-space
• Do DFS, copying from active to idle
  •Copied objects leave “forwarding pointers”
  • Used to update pointers in objects found later
Stop and Copy

- Semi-space is full when allocation fails
  - Stop + Copy
    - Trace heap, copying live objects to idle semi-space
  - Put Allocation Pointer at start of idle semi-space
  - Do DFS, copying from active to idle
    - Copied objects leave “forwarding pointers”
    - Used to update pointers in objects found later
Stop and Copy

- Semi-space is full when allocation fails
  - Stop + Copy
  - Trace heap, copying live objects to idle semi-space
  - Put Allocation Pointer at start of idle semi-space
  - Do DFS, copying from active to idle
    - Copied objects leave “forwarding pointers”
    - Used to update pointers in objects found later
Stop and Copy

- Semi-space is full when allocation fails
  - Stop + Copy
    - Trace heap, copying live objects to idle semi-space
  - Put Allocation Pointer at start of idle semi-space
  - Do DFS, copying from active to idle
    - Copied objects leave “forwarding pointers”
      - Used to update pointers in objects found later
Stop and Copy

- Semi-space is full when allocation fails
  - Stop + Copy
    - Trace heap, copying live objects to idle semi-space
  - Put Allocation Pointer at start of idle semi-space
  - Do DFS, copying from active to idle
    - Copied objects leave “forwarding pointers”
      - Used to update pointers in objects found later
Stop and Copy

- Semi-space is full when allocation fails
  - Stop + Copy
    - Trace heap, copying live objects to idle semi-space
  - Put Allocation Pointer at start of idle semi-space
  - Do DFS, copying from active to idle
    - Copied objects leave “forwarding pointers”
      - **Used to update pointers in objects found later**
Stop and Copy

• Semi-space is full when allocation fails
  • Stop + Copy
    • Trace heap, copying live objects to idle semi-space
  • Put Allocation Pointer at start of idle semi-space
  • Do DFS, copying from active to idle
    • Copied objects leave “forwarding pointers”
      • Used to update pointers in objects found later
Stop and Copy

- At this point, all live objects are copied to idle
  - And their pointers are updated
- Switch which half is idle, which half is active
  - Everything left in new idle half is garbage
Stop and Copy

- At this point, all live objects are copied to idle
  - And their pointers are updated
- Switch which half is idle, which half is active
  - Everything left in new idle half is garbage
- Resume allocation from (new) active half
**S&C: Pros**

- **S&C Pros**
  - **Compaction**: Live objects together, free space together
    - Reduces *fragmentation*
    - Improves locality

- Fragmentation:
  - Free memory non-contiguous
  - May have enough total free memory to satisfy request, but not in contiguous block, so cannot use it
S&C: Pros

- **S&C Pros**
  - **Compaction**: Live objects together, free space together
    - Reduces *fragmentation*
    - Improves locality
  - Faster than M&S if much dead data
    - O(L) : only proportional to live objects
    - Now amortization works better:
      - L / (H-L)
        - Larger heap? Clear win
  - Re-claims all un-reachable data (like M&S)
  - Very fast allocation
    - Check if ptr + size < limit, increase pointer
S&C Cons

- S&C Cons
  - Wastes half the heap
    - Need 2x as much space as you want to use
    - Virtual memory is cheap?
  - Changes pointers
    - Absolutely must know what is a pointer and what is not
      - Could have conservative versions of M&S which assume “might be pointer if could be”

ECE 590.01 (Hilton): GC
Conservative Collection

• GC in C/C++ is harder
  • What is a pointer?
    • Java, SML etc: strongly typed
      • Know types of all things, easy to tell
    • C: can do anything you want
      • Even if it’s a “bad idea”—break type safety
  • Can also reach objects that appear un-reachable
    • Add a constant to another pointer...

• Assuming you don’t do the second, can be conservative about the first
  • Maybe pointer: assume is a pointer
  • Boehm-Weiser collector: works for most C programs
    • Conservative M&S
Generational Collection

• Stop & Copy
  • Efficient, especially if most objects are dead
  • Wastes heap space: 1 GB program—do you really want to waste one more GB?

• Mark & Sweep
  • No wasted space
  • Would be fine if done rarely

• Generational GC
  • Exploit “infant mortality of data”
    • Newly created data likely to die quickly
    • Data which has survived for a while likely to live longer
Generational Collection

- Divide heap into Generations
  - Newly allocated data in Generation 0 (smallest)
  - When data survives enough collections, moves up to a larger generation
  - Larger generations: collected more rarely
  - Different generations may use different GC algorithms

<table>
<thead>
<tr>
<th>Generation 0</th>
<th>Generation 1</th>
<th>Generation 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 KB</td>
<td>128 MB</td>
<td>1 GB</td>
</tr>
</tbody>
</table>

ECE 590.01 (Hilton): GC
Helpful principle

• Generational GC works best under this principle
  • New data points to old data
  • Old data (typically) does not point to new data

• Very true in functional languages
  • Why?

• Less true in imperative languages
  • But still often holds to a large extent
Generational Collection

- Old data points to new data?
  - Effectively have to treat such a pointer as part of the root set
  - Need to identify it
    - But don’t want to trace entire older generation (that’s the whole point)
Dealing with this

- How to handle this?
  - Write-barriers and page-marking

- Mark old generations read-only in page tables
  - Write to old generation?
  - Trap-> handle signal
    - Mark page readable
      - Won’t trap on subsequent writes
    - Set a bit indicating that page is dirty

- GC:
  - Use dirty pages in root set of younger generations

- Other approaches, but we won’t go into them
Other fancy GC things

- Incremental GC
  - Avoid long pauses: do a little bit of GC work everywhere?

- Concurrent GC
  - Run GC fully in parallel with main application
  - Tricky: requires ability to detect if object already traversed changes its pointers to point at something new
GC Wrapup

• GC benefits:
  • Avoid errors with free()
    • Memory leaks
    • Or free()ing live data
  • Performance? Depends… done right, could get
    • Little overhead
    • Fast allocation
    • Improved locality

• Saw three algorithms:
  • Ref counting, Mark & Sweep, Stop & Copy
  • Plus generational