Lecture #38

Next Friday: HKN surveys. Extra points awarded to those who participate!

Today: A little side excursion into nitty-gritty stuff: Storage management.
Scope and Lifetime

- **Scope** of a declaration is portion of program text to which it applies (is visible).
  - Need not be contiguous.
  - In Java, is static: independent of data.
- **Lifetime** or extent of storage is portion of program execution during which it exists.
  - Always contiguous
  - Generally dynamic: depends on data
- **Classes of extent:**
  - **Static**: entire duration of program
  - **Local** or **automatic**: duration of call or block execution (local variable)
  - **Dynamic**: From time of allocation statement (**new**) to deallocation, if any.
Explicit vs. Automatic Freeing

• Java has no means to free dynamic storage.
• However, when no expression in any thread can possibly be influenced by or change an object, it might as well not exist:

```java
IntList wasteful ()
{
    IntList c = new IntList (3, new IntList (4, null));
    return c.tail;
    // variable c now deallocated, so no way
    // to get to first cell of list
}
```

• At this point, Java runtime, like Scheme’s, recycles the object c pointed to: garbage collection.
Under the Hood: Allocation

- Java pointers (references) are represented as integer addresses.
- Corresponds to machine’s own practice.
- In Java, cannot convert integers ↔ pointers,
- But crucial parts of Java runtime implemented in C, or sometimes machine code, where you can.
- Crude allocator in C:

```c
char store[STORAGE_SIZE]; // Allocated array
size_t remainder = STORAGE_SIZE;

/** A pointer to a block of at least N bytes of storage */
void* simpleAlloc (size_t n) { // void*: pointer to anything
    if (n > remainder) ERROR ();
    remainder = (remainder - n) & ~0x7; // Make multiple of 8
    return (void*) (store + remainder);
}
```
Example of Storage Layout: Unix

- OS gives way to turn chunks of unallocated region into heap.
- Happens automatically for stack.
Explicit Deallocating

- C/C++ normally require explicit deallocation, because of
  - Lack of run-time information about what is array
  - Possibility of converting pointers to integers.
  - Lack of run-time information about unions:

```c
union Various {
    int Int;
    char* Pntr;
    double Double;
} X;  // X is either an int, char*, or double
```

- Java avoids all three problems; automatic collection possible.

- Explicit freeing can be somewhat faster, but rather error-prone:
  - Memory corruption
  - Memory leaks
Free Lists

- Explicit allocator grabs chunks of storage from OS and gives to applications.
- Or gives recycled storage, when available.
- When storage is freed, added to a free list data structure to be recycled.
- Used both for explicit freeing and some kinds of automatic garbage collection.
- Problem: free memory fragments.

Variables
(visibile to program)

The Heap

Free List
Garbage Collection: Reference Counting

- Idea: Keep count of number of pointers to each object. Release when count goes to 0.

Y: □
X: □ → 1 □ → 1 □ → 1 □
    1A → 1B → 1C

Y = X.tail;
Y: □
X: □ → 1 □ → 2 □ → 1 □
    1A → 1B → 1C

X = Y;
Y: □
X: □ → 0 □ → 3 □ → 1 □
    1A → 1B → 1C

Y: □
X: □ → 2 □ → 1 □
    0A → 1B → 1C etc.

Last modified: Wed Nov 26 12:56:27 2014
Garbage Collection: Mark and Sweep

Roots (locals + statics)

1. Traverse and mark graph of objects.

2. Sweep through memory, freeing unmarked objects.

Before sweep:

<table>
<thead>
<tr>
<th>A</th>
<th>B*</th>
<th>C</th>
<th>D*</th>
<th>E*</th>
<th>F</th>
<th>G*</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>D</td>
<td>G</td>
<td>F</td>
<td>A</td>
<td>7</td>
<td>G</td>
</tr>
</tbody>
</table>

After sweep:

| B | D | G | D | 7 | G | D | E | G | E |
Copy Garbage Collection

- Mark-and-sweep algorithms don’t move any existing objects—pointers stay the same.

- The total amount of work depends on the amount of memory swept—i.e., the total amount of active (non-garbage) storage + amount of garbage. Not necessarily a big hit: the garbage had to be active at one time, and hence there was always some “good” processing in the past for each byte of garbage scanned.

- Another approach: copying garbage collection takes time proportional to amount of active storage:
  - Traverse the graph of active objects breadth first, copying them into a large contiguous area (called “to-space”).
  - As you copy each object, mark it and put a forwarding pointer into it that points to where you copied it.
  - The next time you have to copy a marked object, just use its forwarding pointer instead.
  - When done, the space you copied from (“from-space”) becomes the next to-space; in effect, all its objects are freed in constant time.
Copying Garbage Collection Illustrated

(a) Roots

from:  

\[
\begin{array}{cccccccc}
A & B & C & D & E & F & G \\
42 & D & G & F & A & 7 & G & D & C & E
\end{array}
\]

to:  

(b) Roots

G

\[
\begin{array}{cccccccc}
A & B' & C & D & E' & F & G \\
42 & B' & G & F & A & 7 & G & E' & C & E
\end{array}
\]

to:  

D

\[
\begin{array}{cccccccc}
B' & E' & D & G & D & E
\end{array}
\]

(c) Roots

\[
\begin{array}{cccccccc}
A & B' & C & D' & E' & F & G' \\
42 & B' & G & F & A & D' & 7 & G & E' & C & G' & E
\end{array}
\]

to:  

\[
\begin{array}{cccccccc}
D' & G' & D & 7 & G & E
\end{array}
\]

(d) Roots

\[
\begin{array}{cccccccc}
A & B' & C & D' & E' & F & G' \\
42 & B' & G & F & A & D' & 7 & G & E' & C & G' & E
\end{array}
\]

to:  

\[
\begin{array}{cccccccc}
D' & G' & D' & 7 & G' & E'
\end{array}
\]

B: Old object
B': New object