The UNIVERSITY of VERMONT

Benjamin Fenster
CS-100 (Damon)
19 March 2003

Notes – Memory Allocation

I. Objects
  a. Everything in the representation defines what storage is needed for the object.
  b. C++ guarantees fields will be in the order they're declared.
  c. Java rearranges fields so they'll fit better.
    i. Variables will need to align with 8 byte boundaries.
    ii. With a one-byte variable followed by some other storage could waste seven bytes.
    iii. Generally the most efficient storage is from largest to smallest.
  d. Subclasses append the additional representation to the end of the superclass representation.
  e. Since everything is a subclass of Object, there's always some stuff at the top that allows access to Object's class code.

II. Memory
  a. Divided into three pieces.
  b. Stack
    i. Local variables.
    ii. Grows down.
    iii. A new record is added for each function call.
    iv. Memory goes away when the function ends.
    v. Objects can be stored directly here, so they'll go away after the function is done running.
    vi.
  c. Pre-allocated Storage
    i. Global variables, static variables.
    ii. Size of everything is fixed as soon as the program is compiled.
  d. Heap
    i. The interesting piece.
    ii. Grows upward.
    iii. Object variables typically go here.
    iv. In Java, *all* objects go here.
    v. All heap space is dynamically allocated and released.

III. Garbage Collection
  a. Java uses garbage collection.
  b. The system automatically does all the deletes it can.
  c. That means you cannot explicitly say "delete."
  d. The result is a slightly slower runtime.
    i. Objects get deleted in batches.
    ii. One pass used to involve scanning the entire heap, and would take several seconds to execute.
    iii. Now it takes a few 10s of milliseconds.
    iv. That's still problematic if you want something done in real time.
  e. The upside is it's much more reliable.
    i. Deleting objects is one of the most error-prone parts of programming.
    ii. Letting the system handle it saves a lot of hassle.
  f. Any object that can't possibly be used any more gets deleted.
    i. If an object can be reached by any means, it's still in use.
    ii. It's possible to have references to an object from another object that's unreachable.
    iii. Thus, reference counting doesn't quite work.
  g. When you want to make an object garbage, just set it to null – that removes the link and renders the space unusable.
  h. The Exception
    i. Suppose you've got a set of employees that's stored as a static class field.

    ii. There’s no way to get rid of the storage when you’re done with it (for a single employee)
    iii. Weak References
        1. Says that a particular reference doesn’t count for garbage collection purposes.
        2. If an object has only a weak reference, count it as garbage.
    iv. Use the finalize() method
        1. Called just before the object gets garbage collected.
        2. Guaranteed to be called exactly once on any object.
        3. Exceptions that are thrown are ignored, it just doesn’t finish the execution.
        4. This is good for things like removing an object from a set if it was in one.
        5. Compares to a C++ destructor, but is MUCH more rare.

IV. C++ “Garbage Collection”
- a. Doesn’t exist!
- b. Explicit deletion is “the norm.”
- c. Implicit Deletion
    - i. If the object is on the stack, it gets deleted when the function ends.
    - ii. If it’s a global, it gets deleted when the program ends.
    - iii. Otherwise there is no implicit deletion of objects.
- d. It’s up to the programmer to delete every object exactly once.
- e. Dangling References
    - i. Something has been deleted that’s still in use.
    - ii. Best Case
        1. Operating system yells, and the program crashes.
        2. It’s the best case because it’s clear what the problem is.
    - iii. Worst Case
        1. The OS lets the program use the data stored in that memory as if it were still valid.
        2. That can take days to weeks to debug.
        3. The worst kind of bug to face.
    - iv. It may be tempting to just not delete anything for fear of this, but…
- f. Memory Leaks.
    - i. …if you decide never to delete an object, but then remove all references to it you’ve left garbage.
    - ii. More and more memory is consumed.
    - iii. It’s not worth the pain here either.
- g. Destructors
    - i. Called when the object is deleted.
    - ii. The complement of a constructor.
    - iii. The same as Java’s finalize() method.
    - iv. Very, very common.
    - v. Responsible for deleting all representation too.
    - vi. Need to delete everything that’s not exposed, or you’ve got a memory leak.
- h. Gives much more control over memory management
    - i. operator new
        1. Global
        2. Default just returns some memory.
        3. Comparable to C’s malloc()
        4. Returns a void*
    - ii. operator delete
        1. Makes memory free again.
        2. Complement of operator new
    - iii. What if you run out of memory?
        1. If new_handler() is defined, it gets called.

    2. Otherwise new will return 0 (NULL).

iv. Class-specific operator new, operator delete

    1. Very handy
    2. You'd usually do this for better performance.
    3. Take advantage of the fact that every instance of an object will be exactly the same size.
    4. It's especially great for small classes.
    5. Normally the size is stored with each instance of an object, but if the size is always the same there's no point in that.
    6. Normally freed memory should be coalesced as more blocks are freed so it will be easier to reallocate. If all allocations will be the same size there's no point in that.
    7. Just keep a pointer to the first free space, and from there keep a linked list of all free spaces.