Programming Languages

Garbage collection

CSCI.GA-2110-001
Summer 2011
For most languages, the amount of memory used by a program cannot be determined at compile time

- earlier versions of FORTRAN are exceptions!

Some features that require dynamic memory allocation:

- recursion
- pointers, explicit allocation (e.g., `new`)
- higher order functions
Types of Allocation

- **Static** – absolute address retained throughout program’s execution.
  - Static variables
  - Global variables
  - Certain fixed data (e.g., string literals)

- **Stack** – last-in, first-out ordering.
  - Subroutine arguments
  - Local variables
  - Runtime system data structures (displays, etc.)

- **Heap** – general storage, for allocation at arbitrary times.
  - Explicitly or automatically allocated
  - Resizable types (e.g., String)
  - Java class instances
  - All objects and data structures in Python
In imperative languages, space for local variables and parameters is allocated in activation records, on the *stack*.

The lifetime of such values follows a LIFO discipline – when the routine returns, we don't need its locals or arguments any more.

The lifetime (aka *extent*) of local variables may be longer than the lifetime of the procedure in which they were created.

These are allocated on the *heap*. 
The heap is finite – if we allocate too much space, we will run out.

Solution: deallocate space when it is no longer necessary.

Methods:

- Manual deallocation, with e.g., `free, delete` (C, Pascal)
- Automatic deallocation via garbage collection (Java, C#, Scheme, ML, Perl)
- Semi-automatic deallocation, using destructors (C++, Ada)
  - Automatic because the destructor is called at certain points automatically
  - Manual because the programmer writes the code for the destructor

Manual deallocation is dangerous (because not all current references to an object may be visible).
Allocation

Two basic methods:

- free list – typically for manual and semi-automatic deallocation
- heap pointer – typically for automatic deallocation

**Free list method:**

- a linked list of unused blocks of memory is maintained (the *free list*)
- **Allocation**: a search is done to find a free block of adequate size; it’s removed from the free list
  - first-fit, best-fit
- **Deallocation**: the block is placed on the free list

Problems:

- may take some time to find a free block of the right size
- memory eventually becomes fragmented
Allocation: Heap pointer

Heap pointer method:

- Initially, the heap pointer is set to bottom of heap
- **Allocation**: the heap pointer is incremented an appropriate amount
- **Deallocation**: defragmentation eventually required

Problems:

- requires moving of live objects in memory
Automatic deallocation

Basic garbage collection algorithms:

- reference counting – usually done by programmer
- mark/sweep – needs run-time support
  - variant: compacting
  - variant: non-recursive
- copying – needs run-time support
  - variant: incremental
  - variant: generational
An object $x$ is live (i.e., can be referenced) if:

- $x$ is pointed to by some variable located
  - on the stack (e.g., in an activation record)
  - in static memory
- there is a register (containing a temporary or intermediate value) that points to $x$
- there is another object on the heap (e.g., $y$) that is live and points to $x$

All live objects in the heap can be found by a graph traversal:

- start at the roots – local variables on the stack, static memory, registers.
- any object not reachable from the roots is dead and can be reclaimed
- each object has an extra bit called the \textit{mark bit}
- \textbf{mark phase}: the collector traverses the heap and sets the mark bit of each object encountered
- \textbf{sweep phase}: each object whose mark bit is not set goes on the free list

<table>
<thead>
<tr>
<th>name</th>
<th>definition</th>
</tr>
</thead>
</table>
| GC()    | for each root pointer $p$ do  
          | mark($p$);  
          | sweep();  |
| mark($p$)| if $p->\text{mark} \neq 1$ then  
          | $p->\text{mark} = 1$;  
          | for each pointer field $p->x$ do  
          | mark($p->x$);  |
| sweep() | for each object $x$ in heap do  
          | if $x.\text{mark} = 0$ then insert($x$, free_list);  
          | else $x.\text{mark} = 0$;  |
- heap is split into 2 parts: FROM space, and TO space
- objects allocated in FROM space
- when FROM space is full, garbage collection begins
- during traversal, each encountered object is copied to TO space
- when traversal is done, all live objects are in TO space
- now we flip the spaces – FROM space becomes TO space and vice versa
- Note: since we are moving objects, any pointers to them must be updated
  This is done by leaving a forwarding address

heap pointer method used for allocation – fast
<table>
<thead>
<tr>
<th>name</th>
<th>definition</th>
</tr>
</thead>
</table>
| GC()   | for each root pointer p do  
|        | p := traverse(p);          |
|        | if *p contains forwarding address then  
|        | p := *p; // follow forwarding address  
|        | return p;                   |
|        | else {                      |
|        | new_p := copy (p, TO_SPACE);|
|        | *p := new_p; // write forwarding address  
|        | for each pointer field p->x do  
|        | new_p->x := traverse(p->x);   |
|        | return new_p;               |
| traverse(p) |                     |
Generational GC

- a variant of a copying garbage collector
- Observation: the older an object gets, the longer it is expected to stay around.
  Why?
  - many objects are very short-lived (e.g., intermediate values)
  - objects that live for a long time tend to make up central data structures in the program, and will probably be live until the end of the program

- Idea: instead of 2 heaps, use many heaps, one for each “generation”
  - younger generations collected more frequently than older generations (because younger generations will have more garbage to collect)
  - when a generation is traversed, live objects are copied to the next-older generation
  - when a generation fills up, we garbage collect it
Reference Counting

The problem:

- we have several references to some data on the heap
- we want to release the memory when there are no more references to it
- may not have “built-in” garbage collection

Idea: Keep track of how many references point to the data, and free it when there are no more.

- set reference count to 1 for newly created objects
- increment reference count whenever we make a copy of a pointer to the object
- decrement reference count whenever a pointer to the object goes out of scope or stops pointing to the object
- when an object’s reference count becomes 0, we can free it
Reference Counting

Advantages:

- Memory can be reclaimed as soon as no longer needed.
- Simple, can be done by the programmer for languages not supporting GC.

Disadvantages:

- Additional space needed for the reference count.
- Will not reclaim circular references.
- Can be inefficient (e.g., if many objects are reclaimed at once).
Comparison

Costs of various methods:

\[ L = \text{amount of storage occupied by live data} \]
\[ M = \text{size of heap} \]

- Mark/sweep: \( O(L) + O(M) = O(M) \) since \( M > L \)
- Copying: \( O(L) \)

Experimental data for LISP: \( L \approx 0.3 \times M \)

Harder to compare with reference counting, but mark/sweep and copying are generally \textit{faster}. 
C++: Important events in the lifetime of an object

<table>
<thead>
<tr>
<th>Event</th>
<th>what gets called (declaration)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation</td>
<td><code>C (...)</code> // constructors</td>
</tr>
<tr>
<td>Pass by value</td>
<td><code>C (const C&amp;)</code> // copy constructor</td>
</tr>
<tr>
<td>Assignment</td>
<td><code>C&amp; operator= (const C&amp;)</code></td>
</tr>
<tr>
<td>Destruction</td>
<td><code>~C ()</code> // destructor</td>
</tr>
</tbody>
</table>

A chief reason C++ has destructors is to enable implementation of reference counting.
class C {
public:
    C() : p(NULL) {} 
    C(const C& c) : p(c.p) { if (p) p->refCount++ ; } 
    ~C() { if (p && --p->refCount == 0) delete p ; } 
    C& operator=(const C&); 
    ... 
private:
    struct RefCounted {
        int refCount;
        ...
        RefCounted(...) : refCount(1), ... { ... }
    };
    RefCounted *p;
}
const C& C::operator= (const C& c) {
  if (c.p)
    c.p->refCount++;

  if (p)
    p->refCount--;

  p = c.p;

  return *this;
}
What about weakly typed languages?
What about languages not designed for GC? (hostile environments)

It turns out that strong typing is not necessary for garbage collection.

**Approach:** traverse the stack, static memory, heap and *guess* whether bit patterns “look like” a pointer.

- If memory beginning at address $x$ was previously allocated and there is no pointer-like memory address pointing to $x$, then deallocate the block at $x$.
- If some bit pattern in memory points to $x$, do not deallocate $x$.
- Worst case: some objects may not be deallocated.