Memory Management

- Stack: Data on stack (local variables on activation records) have lifetime that coincides with the life of a procedure call.
  Memory for stack data is allocated on entry to procedures . . .
  . . . and de-allocated on return.
- Heap: Data on heap have lifetimes that may differ from the life of a procedure call.
  Memory for heap data is allocated on demand (e.g. `malloc`, `new`, etc.) . . .
  . . . and released
  - Manually: e.g. using `free`
  - Automatically: e.g. using a garbage collector

Memory Allocation

- Heap memory is divided into **free** and **used**.
- Free memory is kept in a data structure, usually a free list.
- When a new chunk of memory is needed, a chunk from the free list is returned (after marking it as **used**).
- When a chunk of memory is freed, it is added to the free list (after marking it as **free**)

Fragmentation

- Free space is said to be fragmented when free chunks are not contiguous.
- Fragmentation is reduced by:
  - Maintaining different-sized free lists (e.g. free 8-byte cells, free 16-byte cells etc.) and allocating out of the appropriate list.
  - If a small chunk is not available (e.g. no free 8-byte cells), grab a larger chunk (say, a 32-byte chunk), subdivide it (into 4 smaller chunks) and allocate.
  - When a small chunk is freed, check if it can be merged with adjacent areas to make a larger chunk.

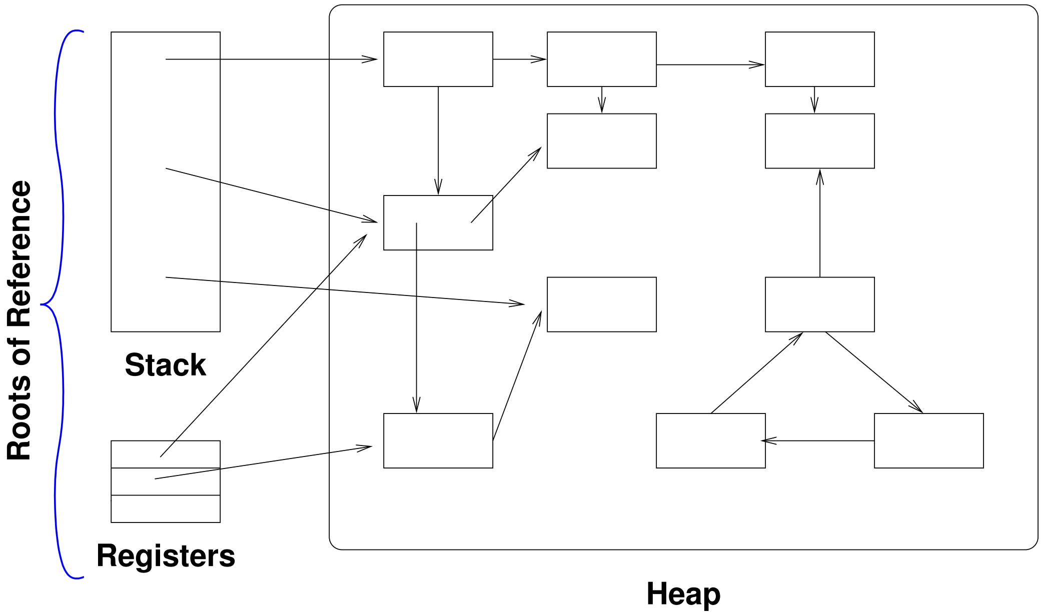
Manual Memory Management

- Programmer has full control over memory
- . . . with the responsibility to manage it well
- Premature `free`'s lead to dangling references
- Overly conservative `free`'s lead to *memory leaks*
- With manual `free`'s it is virtually impossible to ensure that a program is correct and secure.
- Even with manual memory management, the system maintains bookkeeping data and does non-trival memory-related processing (e.g. search for appropriate chunk to allocate, avoid fragmentation, etc.)

Automatic Memory Management

- An object in the heap is *garbage* if it will never be subsequently used by the program.
- Automatic memory management techniques detect which objects in the heap are garbage
- . . . and release garbage objects to the free list.
- Determining whether or not an object is garbage is undecidable in general.
- Hence we use conservative techniques to detect garbage.

Object Graph



Any object unreachable from the roots of reference is *garbage*.

Reference Counting

- Reference counting is a conservative technique for detecting garbage.
- Each object has a *reference count*: the no. of references made to it. (in-degree of the node in object graph)
- When an object is allocated, we set its reference count to **0**.
- When a new reference to an object is *created* (e.g. the object referred by `q` in `p=q`) then its reference count is *incremented*.
- When an existing reference to an object is *removed* (e.g. the object referred by `p` in `p=q`, then its reference count is *decremented*.
- When the reference count of an object falls to *0*, then the object is garbage.

Pluses and Minuses of Reference Counting

+ Simple. Can be employed by programmers to ensure there are no dangling references.
+ Needs no elaborate system support. (e.g. used in OS Kernel data structures)
– Has high overheads.
  e.g. `p = q` will need manipulation of two counts: one move instruction will now need 6 or more extra instructions.
– Cyclic structures cannot be detected as garbage.

Trace-Based Garbage Collection

- Techniques for "behind the scenes" garbage detection and disposal
  Need no programmer intervention (may need compiler support).
- Garbage collector is called at some program point (usually when memory allocation fails or memory is deemed too "full")
- When called, computes the set of reachable nodes in the object graph.
- Every node that is unreachable is garbage, and is released.
- Needs to know
  - the set of all objects in heap (free or not)
  - the set of all pointers in each object

Types of Trace-Based Garbage Collectors

- *Mark-and-Sweep:* Marks each reachable node in first phase
  Releases every unmarked node in the second phase.
- *Copying Collector:* Heap is divided into two spaces, only one of which is active
  Copies reachable objects from active to inactive space
  Switches active and inactive spaces when copying is done
- *Generational GC:* Objects are divided into old and new generations, and new generations are collected more often
- *Concurrent GC:* Garbage collector runs concurrently (e.g. in a separate thread) with the program; the program is not interrupted for collection

Mark-and-Sweep Collector

Two-phase collector

- *Mark Phase:* Does a depth-first traversal of the object graph starting from the roots.
  Marks all objects visited (note reachable nodes represent live data)
- *Sweep Phase:* Does a sweep over the *entire heap*,
  adding any unmarked node to the free list, and
  removing marks from nodes (preparing for next round)

Needs extra bookkeeping space in each object for storing the marks.

Pragmatics of Mark-and-Sweep

- Keeps objects in place after collection
  - + Can be used for conservative collection in languages such as C.
  - - Memory may be fragmented
- Cost of collection is proportional to the entire heap size (since sweep traverses the whole heap).
- Mark phase does a depth-first search and needs a stack.
  - Explicit stack consumes extra space!
  - Stack can be simulated by reversing pointers during traversal (Deutsch-Schorr-Waite algorithm)

Copying Collector

Two-Space Collector (Cheney's Algorithm)

- Heap is divided into two spaces:
  - *From Space:* The currently active heap
  - *To Space:* Space to which objects will be copied (currently inactive)
- Objects reached are copied from the *From Space* to *To Space*
- References to copied objects are modified during the traversal.
- *From* and *To* spaces are swapped at the end of copying

Cheney's Copying Collector

- Traversal of the object graph is done *breadth-first*, starting from the roots.
- When an object is copied, a *forwarding pointer* is kept in place of the old copy.
- From and To spaces are contiguous chunks of memory.
- Two pointers are kept in the *To Space*:
  - *Scan Pointer:* All objects behind it (i.e. to its left) have been fully processed; objects in front of it have been copied but not processed.
  - *Free Pointer:* All copied objects are behind it; Space to its right is free.

Cheney's Algorithm

Initially, *Scan Pointer* and *Free Pointer* are 0.

- For each heap reference R in the root set
  1. If $*R$ is not yet copied
     Copy object $*R$ to *To Space* (starting at *Free Pointer*)
     $R = *R =$ *Free Pointer*
     Free Pointer $+=$ *sizeof(*R)*
  2. If $*R$ has been copied
     $R = *R$
- While *Scan Pointer* != *Free Pointer*
  - Let *Obj* be the object at *Scan Pointer*
  - For each reference R in *Obj*
    Do steps 1 and 2 (whichever is applicable) above.
  - *Scan Pointer* $+=$ *sizeof(Obj)*

Pragmatics of Copying Collection

- Needs more heap space than is currently used
  + Memory is compacted during copy, and hence no fragmentation
  - Can be used for conservative collection in languages such as C.
- Cost of collection is proportional to size of live objects in heap (unreachable objects are not touched).
- Objects that survive a collection may get copied repeatedly, which is expensive.
- Often used as a part of a generational garbage collector.