**Memory Management & the Heap**

* E.g. dangling pointer

int \*f() {

int x = 0; ← should be int \*x = new int[0];

return &x; //BAD ← return x;

}

* Explicit memory management
* E.g. delete/free etc.
* How do new/delete work?
* Method 1 – list of free blocks
  + Maintain a linked list of pointers to free areas of memory
  + Initially – entire heap is free, list contains one entry
  + Ex: suppose heap contains 1024 bytes
    - Free list has pointer to block of length 1024
    - Suppose 16 bytes is allocated
      * Actually allocate 20 bytes = 16 bytes + 1 int
      * Return a pointer to the 2nd word
      * 1st word contains the size of allocated memory (20)
    - Free list now points to block of length 1004
    - Suppose 28 bytes is allocated
      * Actually allocate 32 bytes
    - Free list now points to block [972]
    - Suppose the first block is freed
      * Free list now points to block [20], which has a pointer to a block [972] (i.e. linked list)
    - Suppose the second block is freed
      * Free list now points to block [20] → block [32] → block [972]
      * Order the blocks by address – because now for every p = pointer in free list to a block, p + \*p == \*(p + 1)
      * i.e. since \*p contains the size of block pointed to by p, p + \*p is the address of the next block
      * This implies that heap blocks are contiguous – they can be merged
  + As allocation/deallocation occur repeatedly, holes are created in the heap
    - i.e. fragmentation – if even n bytes are free, a block of n bytes may not be able to be allocated
    - To reduce fragmentation – don’t always use the first block of RAM big enough for the requested size
  + E.g. 3 separated free blocks – [20], [15], [100]
    - 10 bytes is requested
    - First fit – use [20] block
    - Best fit – use [15] block; try to conserve the largest free blocks
  + Searching for available RAM is costly – linear time
* Method 2 – binary buddy system
  + Assume size of heap is a power of 2
  + Memory is also allocated in powers of 2
  + Ex: suppose heap contains 4k = 1024 words
    - Suppose 20 words is requested
      * i.e. 21 words is needed
      * Actually allocate 32 words
      * Split [1024] heap into 2 heaps of [512] a.k.a. “buddies”
      * Split one of the [512] heaps into 2 heaps [256]
      * Etc. …
      * Split down to heaps of size 32, then return a pointer to one of them
    - Heap is now [32] → [64] → … → [512]
    - Suppose 63 words is requested
      * i.e. 64 words is needed
      * Use the [64] block already available
    - Heap is now [32] → [128] → [256] → [512]
* Implicit memory management
* E.g. garbage collection – runtime reclaims memory that is no longer accessible
* Method 1 – mark and sweep
  + Scan the entire stack and look for pointers
  + For each pointer found, mark the block it points to
  + If the heap object contains pointers, follow those as well
  + Then all accessible heap memory will have been marked
  + Scan the heap and reclaim all unmarked blocks & clear the marks
  + Con: this technique requires the program execution to be paused
* Method 2 – reference counting
  + For each heap block, keep track of the # of pointers to point to it (its reference count)
  + Update reference counts whenever a pointer is reassigned
  + When a block’s reference count reaches 0, reclaim it
  + Con: circular references
    - i.e. a → b → c → a
    - a, b, c all have reference count = 1, but they are collectively inaccessible – they should be reclaimed
* Method 3 – copying garbage collection
  + Split heap in 2 halves – “from” & “to”
  + Allocate only from “from”
  + When “from” fills up, all reachable data is copied from “from” to “to”
    - The roles of “from” & “to” are reversed
  + This technique has built-in compaction
    - Guaranteed that after the swap, all reachable data occupies contiguous memory – reduces fragmentation
  + Cons:
    - Can only use half of the heap
    - Program execution needs to be paused