Lecture 10 – C: the heap and manual memory management
Pointer syntax

• A review (for completeness)
• Declare a variable to have a pointer type:
  
  t * x; or t* x; or t *x; or t*x;
  
  (where t is a type and x is a variable)

• An expression to dereference a pointer:
  
  *x (or more generally *e)
  
  where e is an expression.

• C’s designers used the same character on purpose, but declarations (create space) and expressions (compute a value) are totally different things.
Heap allocation

• So far, all of our ints, pointers, and arrays, have been stack-allocated, which in C has two huge limitations:
  – The space is reclaimed when the allocating function returns
  – The space required must be a constant (only an issue for arrays)
• Heap-allocation has neither limitation.
• Comparison: new T(...) in Java does all this:
  – Allocate space for a T (exception if out-of-memory)
  – Initialize the fields to null or 0
  – Call the user-written constructor function
  – Return a reference (hey, a pointer!) to the new object
• In C, these steps are almost all separated
malloc, part 1

• malloc is “just” a library function: it takes a number, heap-allocates that many bytes and returns a pointer to the newly-allocated memory.
  – Returns NULL on failure.
  – Does not initialize the memory.
  – You must cast the result to the pointer type you want.
  – You do not know how much space different values need!
    • Do not do things like malloc(17)!
malloc, part 2

- malloc is “always” used in a specific way:
  
  
  (t*)malloc(e * sizeof(t))

- Returns a pointer to memory large enough to hold an array of length e with elements of type t

- It is still not initialized (use a loop)!
  
  - Underused friend: calloc (takes e and sizeof(t) as separate arguments, initializes everything to 0)

- malloc returns an untyped pointer (void*); the cast (t*) tells C to treat it as a pointer to a block of type t
Half the battle

• We can now allocate memory of any size and have it “live” forever.
• For example, we can allocate an array and use it indefinitely.
• Unfortunately, computers do not have infinite memory so “living forever” could be a problem.
• Java solution: Conceptually objects live forever, but the system has a garbage collector that finds unreachable objects and reclaims their space.
• C solution: You explicitly free an object’s space by passing a pointer to it to the library function free.
• Freeing heap memory correctly is very hard in complex software and is the disadvantage of C-style heap-allocation.
Everybody wants to be free(d once)

```c
int * p = (int*)malloc(sizeof(int));
p = NULL; /* LEAK! */
int * q = (int*)malloc(sizeof(int));
free(q);
free(q); /* HYCSBWK */
int * r = (int*)malloc(sizeof(int));
free(r);
int * s = (int*)malloc(sizeof(int));
*s = 19;
*r = 17; /* HYCSBWK, but maybe *s==17 ?! */
```

- Problems much worse with functions:
  - f returns a pointer; (when) should f's caller free the pointed-to object?
  - g takes two pointers and frees one pointed-to object. Can the other pointer be dereferenced?
The Rules

• For every run-time call to malloc there should be one run-time call to free.
• If you “lose all pointers” to an object, you can’t ever call free (a leak)!
• If you “use an object after it’s freed” (or free it twice), you used a dangling pointer!
• Note: It’s possible but rare to use up too much memory without creating “leaks via no more pointers to an object”.
• Interesting side-note: The standard-library must “remember” how big the object is (but it won’t tell you).
  – We will explore this further…
    later …. 