
CSE 374

Programming Concepts & Tools

Hal Perkins

Fall 2017

Lecture 10 – C: the heap and
manual memory management

Administrivia

- Midterm exam a week from Wednesday
 - Topics – everything up to hw4 (including gdb concepts)
 - These slides (malloc) are for next hw and final
 - Old exams on web now for review/practice
 - Hint: work problems on blank exam(s) first
 - Review Q&A Tuesday Oct. 31, 4:30 – MOR 230
- HW4 reminders
 - (Re-)read the specifications (assignment) carefully, particularly after you “think” you’re done(!)
 - clint: pay attention to most everything. Questions about edge cases, odd warnings, etc.? Discussion board!
- Watch late days – several people have used up all but 1 already – and a couple of people are out(!)
 - Gradebook entry gives number we think you have left

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 variable) 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 (normally) 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
 - And the reference has a specific type: T
- In C, these steps are almost all separated

malloc, part 1

- malloc is “just” a library function: it takes a number, heap-allocates (at least) 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`
 - If allocation fails (extremely rare, but can happen), returns `NULL`. Programs must *always* check.

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
 - Must be a pointer value returned by malloc
- 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)

```
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? (i.e., who owns the pointed-to space?)
 - 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

valgrind

- Ideally there are no memory leaks, dangling pointers, or other bugs, but how do we check?
- *valgrind program program-arguments*
 - Runs *program* with *program-arguments*
 - Catches pointer errors during execution
 - At end, prints summary of heap usage, including details of any memory leaks at termination
 - Option `--leak-check=full` gives more details, use it
- But it *really* slows down execution
 - But still a fantastic diagnostic, debugging tool
- Valgrind has other options/tools but memory check is the default and most commonly used

Processes and the heap

- Recall: a process (running program) has a single address space (code, static/global, heap, stack)
- When a program terminates the address space is released by the OS
 - So any allocated memory is “reclaimed” since it no longer exists
- Good practices
 - OK to rely on this if appropriate, but...
 - Any data structure package that allocates storage should normally provide routines to free it so client code can release the space if the client wants to