CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Spring 2008

Lecture 9— C: structs, the heap and manual memory management

Where are we

- structs (records with fields) and more on pointers
- heap-allocated objects; space leaks and more dangling pointers
- Next week: our first "societal implications" class (suggestions for topics?)

Structs

A `struct` is a record.

A pointer to a `struct` is like a Java object with no methods.

`x.f` is for field access. (if `x` not a pointer — something new!)

`(*x).f` in C is like `x.f` in Java. (if `x` is a pointer)

`x->f` is an abbreviation for `(*x).f`.

There is a huge difference between passing a `struct` and passing a pointer to a `struct`.

Again, left-expressions evaluate to locations (which can be whole struct locations or just a field's location).

Again, right-expressions evaluate to values (which can be whole structs or just a field's contents).

Pointer syntax

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.

(And there's multiplication too.)

Heap-Allocation

So far, all of our ints, pointers, arrays, and structs 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 C(...)` in Java:

- Allocate space for a `C` (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 `(struct Foo*)(malloc(8))`!

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 $\mathbf{0}$).

Note: `x[i]` and `*(x+i)` actually get you the `sizeof(x)` bytes at address "x plus (i times sizeof(x))" – do *not* do `*(x+i*sizeof(x))` (that's too big an index).

Half the Battle

We can now allocate memory of any size and have it "live" forever.

For example, we can allocate an array and return it.

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.

- Later we will learn idioms that help. For now just learn the rules of the game.

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?

`g` takes two pointers and frees one pointed-to object. Can the other pointer be dereferenced?

The rules of malloc/free

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).