CSE 303: Concepts and Tools for Software Development

Dan Grossman

Spring 2005

Lecture 9— C: structs, heap-allocation, initialization, memory management

Where are We

We are learning C, a lower-level and less-safe language than Java So far:

- Basic control constructs
- Left vs. right expressions
- The address-of (&) and dereference (*) operators
- Some strange rules for arrays

Today:

- structs
- memory management
- initialization

Later: header files, the preprocessor, printf, casts, function pointers, gotchas

From last time...

A couple points that got lost in the shuffle:

- Dangling pointers and *lifetime* vs. *scope*
- Declarations should precede uses

Dangling Pointers

```
int* f(int x) {
  int *p;
  if(x) {
    int y = 3;
    p = &x; /* ok */
  } /* ok, but p now dangling */
  /* y = 4 does not compile */
  *p = 7; /* YPMSTCOF, but probably not */
  return p; /* uh-oh */
}
void g(int *p) { *p = 123; }
void h() {
  g(f(7)); /* YPMSTCOF, and likely a problem */
}
```

No forward references

- A function must be defined and/or declared before it is used. (Lying: “implicit declaration” warnings, return type assumed to be `int`, ...)
- You get a *linker error* if something is declared but never defined (or `main` is not defined).
- You can still write mutually recursive functions, you just need a declaration.

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.

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

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

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

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

Again, right-expressions evaluate to values (which can be whole structs or just fields).

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 $\mathbf{0}$).

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 = malloc(sizeof(int));
int * p = NULL; /* LEAK! */
int * q = malloc(sizeof(int));
free(q);
free(q); /* YPMSTCOF */
int * r = malloc(sizeof(int));
free(r);
int * s = malloc(sizeof(int));
*s = 19;
*r = 17; /* YPMSTCOF, 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?