CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Winter 2009

Lecture 11— C: structs, casts, lists

Where are We

We have learned most of the important stuff with C, so now we will more toward idioms and larger programs.

- Today: structs, casts, linked lists
- Next: Rest of the C pre-processor (stuff starting with #)
- Then: Post-overview, more programming tools (make)

Next Wednesday: Midterm in class

Later: 2 lectures on C++ (48 less than necessary)

I highly recommend *understanding* the code posted with this lecture; there is far too much to do that "on-the-fly" in the time we have.

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 a `struct` (value) parameter and a pointer to a `struct`.

There is a *huge* difference between local variables that are `structs` and those that are pointers to `structs`.

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).

C Parameter Rules Revisited

C (almost) uniform rule for parameters: A function parameter is initialized with a *copy* of the actual argument value (`int`, `char`, pointer, struct, ...)

- This holds even for structs(!): a complete copy is created.
- It is far more common to use a pointer to a struct as an argument instead of copying an entire struct (i.e., same semantics as Java object references). But for small things like complex numbers, copying the struct can make good sense.

But: For arguments that are array names, a pointer to the array is passed as the value instead of a copy of the entire array (implicit array promotion).

- Puzzle: if a parameter is an array containing a single struct, is a copy made of the struct/array? What if the parameter is a struct containing an array?

The C types

There are an infinite number of types in C, but only a few ways to make them:

- `char`, `int`, `double`, etc. (many more such as `unsigned int`)
- `void` (a type no expression can have)
- `struct T` where there is already a declaration for that struct type.
- Array types (basically only for stack arrays and struct fields, every use is automatically converted to a pointer type)
- `t*` where `t` is a type
- `union T`, `enum E` (later, maybe)
- function-pointer types (later)
- *typedefs* (just expand to their definition)

Casts, part 1

Syntax: `(t)e` where `t` is a type and `e` is an expression (same as Java).

Semantics: It depends.

- If `e` is a numeric type and `t` is a numeric type, this is a *conversion*.
  - To *wider* type, get same value
  - To *narrower* type, may not (will get *mod*)
  - From floating-point to integral, will *round* (may overflow)
  - From integral to floating-point, may round (but `int` to `double` won't round on most machines)

Note: Java is the same without the "most machines" part.

Note: Lots of *implicit* conversions such as in function calls.

Bottom Line: Conversions involve "real" operations; `(double)3` is a very different bit pattern than `(int)3`.

Casts, part 2

- If `e` has type `t1*`, then `(t2*)e` is a (pointer) cast.
  - You still have the same pointer (index into the address space).
  - Nothing "happens" at run-time.
  - You are just "getting around" the type system, making it easy to write any bits anywhere you want.
  - Old example: `malloc` has return type `void*`.

```
void evil(int **p, int x) {
  int * q = (int*)p;
  *q = x;
}
void f(int **p) {
  evil(p,345);
  **p = 17; // writes 17 to address 345 (HYCSBWK)
}
```

Note: The C standard is more picky than we suggest, but few people know that and little code obeys the official rules.

Pointer casts continued

Questions worth answering:

- How does this compare to Java's casts?
  - Unsafe, unchecked
  - Otherwise more similar than it seems
- When *should* you use pointer casts in C?
  - For "generic" libraries (`malloc`, linked lists, swapping any two pointers, etc.)
  - For "subtyping" (later)
- What about other casts?
  - Casts to/from struct types are compile-time errors.

Java casts

Java casts (e.g., `(Foo)e`) explained to C programmers:

- `e` evaluates to a pointer to an object.
- Objects have "secret fields" at *run-time* indicating their class.
- If the object's secret field is `Foo` or a (transitive) subclass of `Foo` "succeed". Else raise an exception. (Called a downcast)
- If `e`'s (*compile-time*) type is a (transitive) subtype of `Foo`, then the compiler can "omit the check". (Called an upcast)
- If `e`'s (*compile-time*) type is neither a (transitive) subtype nor supertype of `Foo`, it is a compile-time error. (The cast could never succeed.)

Linked lists

Linked lists are a very common data structure.

Building them in C:

- Gives practice with pointers, structs, malloc, etc.
- Leads to using casts for "generic" types.
- Shows memory management problems if lists "share tails".
- Shows the trade-offs between lists and arrays.

See the code! Understand the code!