CSE 374
Programming Concepts & Tools

Hal Perkins

Spring 2010

Lecture 12 –C: structs, linked lists, and casts

Where we are

- We’ve seen most of the basic stuff about C, but we still need to look at structs (aka records or objects without methods) and linked data structures
  – Understand the code posted with today’s lecture – we won’t have time to walk through all the details
- Next: Rest of the C preprocessor (# stuff, macros), building multi-file programs
- Then: more programming tools (make)
- That will set us up for the next programming project
- Meanwhile: Midterm exam this Friday
  – Closed book, optional page of handwritten notes

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 is x not a pointer – 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 parameters - revisited

- C has a uniform rule for parameters (almost): When a function is called, each parameter is *initialized* with a *copy* of the corresponding argument (int, char, ptr,…)
  - This holds even for structs! – a copy is created
  - There is no further connection between the argument and the parameter value in the function
    - But they can point to the same thing, of course
- ***But***: if the argument is an array name, the function parameter is initialized with a pointer to the array argument instead of a copy of the entire array
  - Implicit array promotion

struct parameters

- A struct argument is copied (call-by-value)
- It is far more common to use a pointer to a struct as an argument instead of copying an entire struct
  - Gives same semantics as Java object references
  - Usually what you want – pointer to data that lives outside the function
    - Also avoids cost of copying a possibly large object
  - But occasionally you want call-by value (small things like complex numbers, points, …)
- Puzzle: if an argument is an array containing a single struct, is it copied or is it promoted to a pointer?
  - What if it's a struct containing only a single array?

Linked lists, trees, and friends

- Very, very common data structures
- Building them in C
  - Use malloc to create nodes
  - Need to use casts for “generic” types
  - Memory management issues if shared nodes
  - Usually need to explicitly free entire thing when done
  - Shows tradeoffs between lists and arrays

- Look at the sample code and understand what it does/how it does it

C types

- There are an infinite number of types in C, but only a few ways to make them:
  - char, int, double, etc. (many variations like unsigned int, long, short, …; mostly "implementation-defined")
  - void (placeholder type; 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; type synonym)

Typedef

- Defines a synonym for a type – does not declare a new type
- Syntax

  typedef *type name*;

  After this declaration, writing *name* is the same as writing *type*
- Examples:

```
typedef int int32;              // use int32 for portability
typedef struct point {          // type tag optional (sortof)
    int32 x, y;
} Point2d;                      // Point2d is synonym for struct
typedef Point2d * ptptr;        // pointer to Point2D

Point2d p;                      // var declaration
ptptr ptlist;                   // declares pointer
```

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.

C pointer casts, continued

Questions worth answering:

- How does this compare to Java's casts?
  - Unsafe, unchecked (no "type fields" in objects)
  - Otherwise more similar than it seems
- When should you use pointer casts in C?
  - For "generic" libraries (malloc, linked lists, operations on arbitrary (generic) pointers, etc.)
  - For "subtyping" (later)
- What about other casts?
  - Casts to/from struct types (*not* struct pointer casts) are compile-time errors.