CSE 303:
Concepts and Tools for Software Development

Dan Grossman

Spring 2005

Lecture 13— C: post-overview, function pointers, coding up objects

Where are We

"Official Notice":

- Homework 4 will be posted today, due a week from Thursday
- Midterm Friday, "first page" will be posted today or tomorrow
  - Covers through today
- Homework 2 will be available tomorrow (thanks Ben!)

Today:

- Top-down view of C
- Function pointers
- Coding up objects (more later?)

Top-down post-overview

Now that we have seen most of C, let's summarize/organize:

- Preprocessing
  - `#include` for declarations defined elsewhere
  - `#ifdef` for conditional compilation
  - `#define` for *token-based* textual substitution
- Compiling (type-checking and code-generating)
  - A sequence of *declarations*
  - Each C file becomes a `.o` file
- Linking
  - Take `.o` and `.a` files and make a program
  - `libc.a` in by default, has printf, malloc, ...
  - More later

- Executing
  - O/S maintains the "big array" address-space illusion
  - Execution starts at `main`
  - Library manages the heap via `malloc`/`free`.

C, the language

- A file is a sequence of *declarations*:
  - Global variables (`t x;` or `t x = e;`)
  - `struct` (and `union` and `enum` definitions)
  - Function *prototypes* (`t f(t1,...,tn)`)
  - Function definitions
  - `typedefs`
- A function body is a *statement*
  - Statements are similar to in Java (+ `goto`, – exception-handling, ints for bools)
  - Local declarations have local scope.
- Left-expressions (locations) and right-expressions (values, including pointers-to-locations)
  - * for pointer dereference, & for address-of, . for field access

C language continued

"Convenient" expression forms:

- `e->f` means `(*e).f`
- `e1[e2]` means `*(e1 + e2)`
  - But + for pointer arithmetic takes the size of the pointed to element into account!
  - That is, if `e1` has type `t*` and `e2` has type `int`, then , then `(e1 + c) == (((int)e1) + (sizeof(t) * c))`
  - The compiler "does the sizeof for you" – don't double-do it!

"Size is exposed": In Java, "(just about) everything is 32 bits". In C, pointers are usually the same size as other pointers, but not everything is a pointer.

C is unsafe

The following is allowed to set your computer on fire:

array-bounds violation (bad pointer arithmetic), dangling-pointer dereferences, dereferencing NULL, using results of wrong casts, using contents of uninitialized locations, linking errors (inconsistent assumptions), ...

Casts are not checked (no secret fields at run-time; all bits look the same)

Function pointers

“Pointers to code” are almost as useful as “pointers to data”.

(But the syntax is more painful.)

(Somewhat silly) example:

```
void app_arr(int len, int * arr, int (*f)(int)) {
  for(; len > 0; --len)
    arr[len-1] = (*f)(arr[len-1]);
}
int twoX(int i) { return 2*i; }
int sq(int i) { return i*i; }
void twoXarr(int len, int* arr) { app_arr(len,arr,&twoX); }
void sq_arr(int len, int* arr) { app_arr(len,arr,&sq); }
```

CSE 341 spends a week on *why* function pointers are so useful; today is mostly just *how* in C.

Function pointers, cont'd

Key computer-science idea: You can pass what code to execute as an argument, just like you pass what data to process as an argument.

Java: An object is (a pointer to) code *and* data, so you're doing both all the time.

```
// Java
interface I { int m(int i); }
void f(int arr[], I obj) {
  for(int len=arr.length; len > 0; --len)
    arr[len-1] = obj.m(arr[len-1]);
}
```

C separates the *concepts* of code, data, and pointers.

C function-pointer syntax

C syntax: painful and confusing. Rough idea: The compiler "knows" what is code and what is a pointer to code, so you can write less than we did on the last slide:

```
arr[len-1] = (*f)(arr[len-1]);
  → arr[len-1] = f(arr[len-1]);
app_arr(len,arr,&twoX);
  → app_arr(len,arr,twoX);
```

For types, let's pretend you always have to write the "pointer to code" part (i.e., `t0 (*)(t1,t2,...,tn)`) and for declarations the variable or field name goes after the `*`.

Sigh.

Toward objects

If you want a pointer to code and data, like in Java, then DIY:

```
struct MyPoint {
  // data
  int x;
  int y;
  // code
  int (*getX)(struct MyPoint*);
  void (*setX)(struct MyPoint*,int);
  int (*getY)(struct MyPoint*);
  void (*setY)(struct MyPoint*,int);
  double (*distance2origin)(struct MyPoint*);
};
```

"extra argument" is Java's `this`, else code has no access to the other (data *and* code) fields.

When this "coding pattern" became common, C++ was born (sorta).

A much bigger story

We have just scratched the surface of "C-level OOP".

Food for thought (not on the exam!):

- How could a subclass override methods, add methods, or add fields? (you need casts in a couple different places!)
- What is the difference between calling `this->getX` and calling `MyPoint_getX`?
- Aren't `struct MyPoint` objects awfully large – how could we save space?