CSE 303: Concepts and Tools for Software Development

Dan Grossman

Winter 2006

Lecture 13— C: post-overview, function pointers

Where are We

Today:

- Top-down view of C
- Function pointers
- Connection to objects

Top-down post-overview

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

- Preprocessing
  - `#include` for declarations defined elsewhere
    - Definition is textual; idioms *very* formulaic
  - `#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 (more later)
  - Take `.o` and `.a` files and make a program
  - `libc.a` in by default, has printf, malloc, ...

- 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 (stack space).
- 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.

New side point: padding, alignment may mean "bigger than expected"

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), ...

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

Crashing is a "good thing" compared to continuing silenty with meaningless data.

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]);
}
```

The `m` method of an `I` can have access to data (in fields).

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.

Code pointers often want data

Taking a code pointer can make a function more general because different callers pass different functions.

But *usually* to be more useful you should also pass the function a `void*` also provided by the caller.

- Else code-pointer can read only its arguments and globals.
- Using globals does not work well with this idiom.
- The `void*` is like the fields in an OO object.
  - Especially if you put it in a `struct` with the function pointer!
- See list-find example.

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*);
};
```

1st arg is Java's `this`, else code can't get other (data & code) fields.

Subclassing a *slightly* more complicated story.

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