CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Autumn 2007

Lecture 29— Function Pointers and Objects; Course Wrap-Up

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.

What is an Object?

First Aproximation

- An object consists of data and methods
  - Provides the correct model
  - Easy to explain
- But...
  - Doesn't make engineering sense — we don't want to replicate the (same) method bodies (code) in every object

What is an Object?

Second Aproximation

- An object consists of data and pointers to methods
- Each method has an implicit `this` parameter added that provides a reference to the receiving object
  - Gives method a way to refer to the instance variables of the correct receiver object
- Avoids code duplication
- But...
  - Still wastes space, particularly if there is relatively little instance data, or if the class has a large number of methods

What is an Object?

How it's really done

- There is a single "virtual function" table (vtable) for each class containing pointers to the methods belonging to that class.
  - This is static — does not change during execution
- An object consists of data and a pointer to its class vtable
- Method calls are indirect through the vtable
- Each method still has an implicit `this` parameter that refers to the receiving object
- Avoids code duplication
- Avoids method pointer duplication
- Costs an indirect pointer lookup for each function call

Inheritance and Overriding

We don't have time to talk about this in depth. Basic ideas:

- We still have a vtable for every class and subclass
- The vtable for a subclass points to the correct methods — either ones belonging to the base class that are inherited, or ones belonging to the subclass (added or overriding)
- *Key idea*: The initial part of the vtable for a subclass points to the methods that are inherited or overridden from the base class in *exactly* the same order they appear in the base class vtable
  - So compiled code can find a method at the *same* offset in the vtable whether it is overridden or not
- Use casts as needed to adjust references up and down the inheritance chain

Course Summary

A lot of comfort and de-magicalizing

- 90% of you thought 90% of your classmates knew more of "that practical stuff"

A lot of pragmatic application of computer science

1. There are concepts behind `gdb`, but there's also value in knowing a couple handfuls of commands
2. The best computer scientists are not the folks who have the largest number of utilities, options, and features memorized
   - But the best computer scientists also don't know the least
   - And they're definitely not afraid to learn new ones

Some concepts too

We did *not* just pick up one strange set of rules after another!

- Shell-scripting is about automating program-invocation
- C programming is about a lower level of computing where:
  - all data is just bits and pointers are explicit
  - memory is managed manually
- Debuggers and profilers are about how to characterize and measure running programs
- Linking, make, etc. are about how code is created and combined to make running programs
- Assertions and specifications are about assumptions and the robustness of larger programs

Societal implications

And we talked about ethics too

- We talked about it like computer scientists
- (Overly) logical, rational, understanding limitations of technology and humans
- "We" should not be making all the decisions ourselves, but we should be actively engaged in the process

The Ultimate Goal

5 years from now, look at the course webpage and think this whole course was a waste of time.

After all, you won't remember not knowing this stuff.

And you won't understand why you need a teacher to help you pick up a new tool or automate a repetitive task.

From lecture 1: *There is an amorphous set of things computer scientists know about and novice programmers don't. Knowing them empowers you in computing, lessens the "friction" of learning in other classes, and makes you a mature programmer.*
"There is more to programming than Java methods"
"There is more to software development than programming"
"There is more to computer science than software development"
"There is more to computing's effects than computer science"