CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Autumn 2008

Lecture 29— Function Pointers and Objects

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
- The compiler adds an additional, implicit `this` parameter to every method to provide a reference to the receiving object
  - Gives the 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 class data — 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

Basic ideas:

- We 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