CS 32

Lecture 2: objects good?

"It's an object, all right, but is it d'art?"

Double Vision

- This course has two main tracks
  - Unix/shell stuff
  - Object-Oriented Programming
  - Basic C++ familiarity
- Off by one!

Another Troika

- This class has three main texts
  - CS16 Problem Solving, Ch 10-18
  - CS 24 Data Structures, Ch 12-14
  - Reader
  - Lecture notes
- Off by one

Problem Solving

- CS16 Problem Solving, Ch 10-18
  - Step-by-step C++ course
  - So much for you there, if you read deeply
  - But who can?
- We will go through most chapters

Data Structures

- CS 24 Data Structures, Ch 12-14
  - Assumes you understand classes
  - Develops good models of ADTs
    - Stack, queue, graph, tree
    - Next is the hash table
    - Then search & sort
    - Then inheritance

Reader

1. Unix shells
2. OOP stuff
3. Unix processes
4. gcc, gdb, and make
    - Refers to “Unix Shell”
5. Compilers, linkers, and make
6. Memory, pointers, and OOP stuff
7. Libraries and linking

Do the Dance

- The labs are also mixed between Unix stuff and C++ stuff
- We will try to lean toward the OOP/ C++ direction in the first half of the course
- And of course the other direction — the Unix/Shell — in the second

Object vs Class

- Terminology
  - Object — a value (typ. local)
  - Instance — a value (typ. pointer)
  - Class — a type (typ. private)
  - Struct — a type (typ. public)

Defining Classes

- PSCC++ chapter 10
    - Starts out with struct
    - Why struct and not class?
    - What is difference?

struct v. class

- Pretty much the same, except
  - Every member (ivar, method) is public
  - Should be interchangeable otherwise
  - Smaller projects have less access control, and prefer struct

Structured Data

- Not just an **`int`** or **`char*`** but a whole named list

```
struct Date {
    int month;
    int day;
    int year;
};
```

Getting Access

- You expect to get or set these values by referring to them with dot-syntax.
  - Or arrow -> syntax
  - C++ reminds you of the level of indirection

Reading

- Safer operation
  - `x = p->x;`
    - Direct read presumed
  - `x = p.x;`
    - In C++, you declared **p** without a *
    - In Swift, could be an accessor

Writing

- Usually allowed only for small structs that are passed around by value
- Anything more complicated has private sections and possibly filters for setting
  - `p.x = 3` looks like plain assignment
  - Could be a whole filter on that 3

C++ isms

- Book says quite clearly implement the `::output(ostream& outs)` function
- In Swift, implement the `description()` method
- Your language has its equivalent

Constructors

- If raw values are private, how do you create objects with the values you want?
- Constructors
- Universally invoked by name of Class or Struct used like a function call
    - `y = new Thingy()`

Variety

- Provide a variety of constructors
  - Simple ones have default assumptions

```
BankAccount(int dollars, int cents, double rate);
BankAccount(int dollars, double rate);
BankAccount()
```

- May be funneled into one master constructor
- Details vary by language

All in One

- Can sometimes accomplish this with default variables in the declaration
  - `void func(int a, int b=47);`
  - Much the same in most languages

Declaring in C++

- As usual, two levels of indirection to choose from
  - `C myInstance(3);`
  - `C *myInstance = new C(3);`
  - `C* myInstance = new C(3);`
  - WTF?!?

Local Storage

- `C myInstance(3);`
- Stored on the stack
- copied by value
- Access members with dot operator in C++
    - `myInstance.ivar1`

Pointer

type dereferenced variable value? pointer?

- `C *myInstance = new C(3);`
- `myInstance` is a pointer
  - Data is on the heap
  - Being memory-managed by... ?
- `x = myInstance->ivar3;`
- `x = (*myInstance).ivar3;`

Weird Syntax?

type variable pointer

- `C* myInstance = new C(3);`
- Means the same as previous
- Has pitfall
  - `C* myInstance1, myInstance2;`
  - `C *myInstance1, myInstance2;`

Equality

- Most basic comparison between two objects
- Comparing our date objects is something that should be handled by the object itself
    - But which one?

Two Ways

- Several options
    - Provide an `isEqualTo(C *other)`
    - Overload the `==` operator
    - Section 11.2

Bane or Boon?

- Widely derided as cluttering up C++ code (that other people wrote)
- Great topic for flame wars
- Fell out of favor
- Until…

Better

- Two modern languages go all-in
  - Swift
  - Scala
- Even the built-in operators are declared explicitly

```
public protocol Equatable {
   public static func ==(lhs: Self, rhs: Self) -> Bool
```

Important Overloads

- Equality
- Ordering (possibly)
- Streaming << and >>

No Inheritance

- No Trustafarians
  - Not yet anyway
- We haven't even looked much at encapsulation
- Still looking at a <u>structured</u> value

OOP Proper

- OOP, otherwise unqualified, means
    - Structured values
    - Access control
    - Type inheritance

OOP Über Alles?

- As the Reader points out, it was a big fad for a while
- Hierarchical structures express a lot about the relations between nodes
    - But it's hard to start with a full top-down factoring of things
    - How else can we design?

Prototypes

- What's the big deal with top-down analysis anyway?
- Just take what exists, and modify it
    - Self
    - JavaScript
- Prototype-Based Programming

Functions

- That take functions as parameters
- And return functions
- Everything is a function call
- Function heaven
  - But not the solution to all problems

Truce

- Both Scala and Swift make point of allowing multiple kinds of programming
- This will be the trend for general-purpose languages