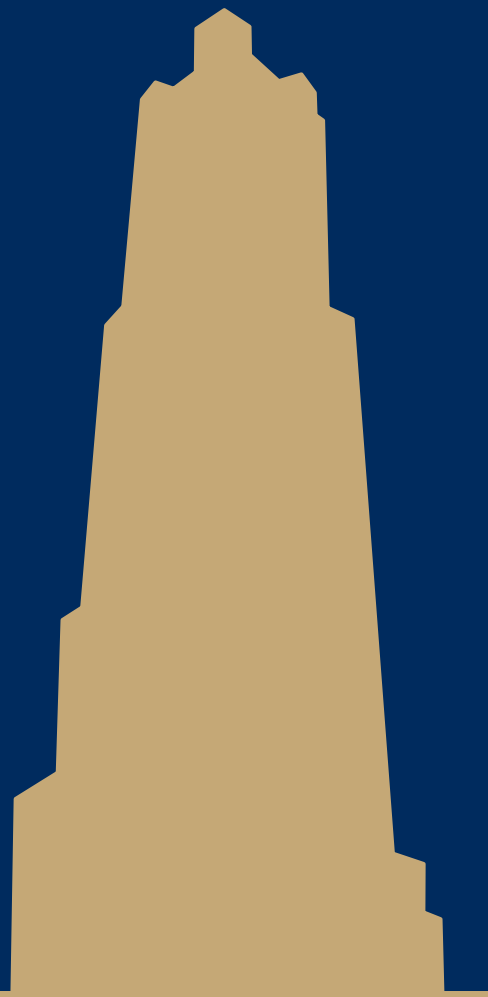
CS/COE 1520

pitt.edu/~ach54/cs1520

Functional programming



A challenge:

- Rewrite the following without using a loop:

```
var data = [1, 2, 3, 4];
for (var i = 0; i < data.length; i++) {
   alert("loopin! " + data[i]);
}
```

A different way of programming

- Functional programming
  - Data stored in the program should be *immutable*
  - Programs should run in a *stateless* manner

Pure functions

- Building blocks of functional programs
- Should be *idempotent*
  - Operate free from their timing relative to other operations
- Should be free of *side-effects*
  - Example side effects:
    - Mutate any shared state or mutable arguments
      - Covered by having immutable data...
    - Don’t produce any observable output, e.g.,
      - Thrown exceptions
      - Triggered events
      - I/O to devices
      - I/O to display
      - Writes to a log
    - Note that this means our programs won't be composed entirely of pure functions

Guidelines for functional programming

- Your functions should never rely on outside values
  - Operate only on data passed in as arguments
- All of your functions must accept at least one argument
- All of your functions must return data or another function
- No loops

Let's consider getting this data from a server

```
[
   {  "name":     "Crosby",
      "age":         28,
      "points":   [0, 0, 0, 1, 1, 0]
   },
   {  "name":     "Malkin",
      "age":         29,
      "points":      [1, 1, 0, 0, 0, 0]
   },
   {  "name":     "Letang",
      "age":         29,
      "points":      [1, 0, 1, 0, 1]
   }
]
```

Grab each player's average points per game

Imperative approach

-vs-

Functional approach

DRYing totalAcrossArray()

- Array.reduce(*callback* [, *initialValue*])
  - *callback* is a reference to a function that will be called for every item in the array being reduced
    - Will be passed 4 arguments:
      - The value previously returned in the last invocation of the callback, or initialValue, if supplied.
      - The current value of the array being processed
      - The index of the current element being processed
      - The array on which reduce was called
    - What should our callback be if we wanted to use reduce() instead of totalAcrossArray()?

DRYing avgAllSubarrays()

- Array.map(*callback* [, *thisArg*])
  - *callback* is, again, a reference to a function that will be called for every item in the array being reduced
    - However, in this case, the result produced by each call is added into a result array instead of being passed to future calls
      - Hence, is only passed 3 arguments:
        - Current value
        - Current index
        - Array
  - *thisArg* allows you to set what should be referenced by the *this* keyword within *callback*

  - What callback could we use to replace avgAllSubarrays()?

DRYing grab()

- How?

Another challenge

- What will this code output?

```
var ex = [];
for (var i = 0; i < 5; i++) {
   ex[i] = function() { document.write(i); };
}
for (var j = 0; j < 5; j++) {
   ex[j]();
}
```

- How can we get it to behave as intended?

Closure example

```
var ex = [];
function funcMaker(i) {
    return function() { document.write(i); };
}
for (var i = 0; i < 5; i++) {
    ex[i] = funcMaker(i);
}
for (var j = 0; j < 5; j++) {
    ex[j]();
}
```

This was only a *brief* intro to functional programming

- For a language designed around functional programming:
  - Haskell