Arrays (with functions)
Ch 7

POINTS OF VIEW

OPTIMIST
"The glass is half-full."

PESSIMIST
"The glass is half-empty."

REALIST
"Yep. That's a glass, alright."

IDEALIST
"One day, cold-fusion from a glass of water will provide unlimited energy and end war."

CAPITALIST
"If I bottled this and gave it a New Agey sounding name, I could make a fortune."

COMMUNIST
"This drink belongs to every single one of us in equal measure."

CONSPIRACIST
"The government is fluoridating the water for mind-control purposes."

SEXIST
"This glass isn't gonna refill itself, honeybun..."

NIHILIST
"The glass does not exist, and neither do I."

OPPORTUNIST:
"There's a funny t-shirt in here somewhere."
Highlights

- partially filled arrays

```c
int arr[20000]; // more than you need!
int arrSize = 4; // how much are you actually using?
```

- arrays in functions

```c
int x[3];
foo(x);
```
Array review

You can ask for multiple boxes in memory with only one variable name for them all:

```c
int x[5]; // 5 ints
```

To access a single box, you use [ ] and the index of where that box is

```c
x[2] = 4;
```

(Remember index starts at zero, so x[2] is 3rd box out of 5 (i.e. the middle))
Arrays can be initialized by the following: (must be done on declaration line!)

```c
int x[] = {1, 4, 5, 2};
```

If you access outside of your array you will either crash or get a random value.

You can also use a constant variable to set the size:

```c
const int size = 8;
int x[size];
```

(See: average.cpp)
Arrays

When you make an array, the computer reserves space in memory for the size

The array variable is then just a reference to the first element's memory location

The computer simply converts the index into an offset from this initial location (see arrayAddress.cpp)
Memory

Memory:

Code:

CAUTION OFF LIMITS  CAUTION OFF LIMITS
Memory (declaration)

Memory:
#0 (int) x

Code:
int x;
Memory (declaration)

Memory:

#0 (int) x
#1 (int) y[0]
#2 (int) y[1]
#3 (int) y[2]

y is the address of y[0]

Code:

```c
int x;
int y[3];
```
Arrays - looping

As arrays store multiple elements, we very often loop over those elements.

There is a special loop that goes over all elements (for each):

```cpp
int x[] = {1, 4, 5, 2};
for(int a : x) {
    // a has the value of x[i] for each i
}
```

(See: forEach.cpp)
Partially filled arrays

Arrays are annoying since you cannot change their size.

You can get around this by making the array much larger than you need.

If you do this you need to keep track of how much of the array you are actually using.

(See: partiallyFilled.cpp)
Each element of an array is the same as an object of that type

For example: 

\[
\text{int[]} \ x = \{1, 2\}; \\
\text{int} \ x0 = 1;
\]

(See: maxPassInt.cpp)
Arrays are references (memory addresses)

This means we can pass the reference as an argument in a method

Then the method can see the whole array, but it won't know the size

(See: maxPassArray.cpp)
Array - array passing

But wait! This means the function can change the data since we share the memory address

(See: reverse.cpp)
Array - array passing

If we want to prevent a function from modifying an array, we can use `const` in the function header:

```cpp
void reverse(const int word[]);
```

This also means any function called inside `reverse` must also use `const` on this array

(See: reverseFail.cpp)
Array - returning arrays

However, we do not know how to return arrays from functions (yet)

```c
int[] foo(){
    int x[] = {1,2};
    return x;
}
```

// x dies here, what are you returning?

For now, you will have to pass in an array to be changed, much like call-by-reference