TODO: this needs more work from some more technical sources

Overview

- mathematical functions think of a variable, x, as a set value at time of input to a function and variable with respect to what is input
- in general programming languages, a variable, x, stores a value that is changed over time
- in C++, in general, these are objects (or primitive types)
- the distinction between a value type and an object is similar to the distinction between and lvalue and an rvalue
- lvalue requires an address
- rvalue needs to be movable

Value Semantics

- focus on values in rather than objects and an object's identity
- reference semantics considers objects as locations that are modified
- C++ allows value and reference semantics

Uses

- pass and return from functions by value by default
- no memory management issues (dangling references, nonexistent objects, free store usage, memory leaks, pointers (smart or regular))
- no reference aliasing problems
- helps with multithreaded code
- referential transparency (avoids possibility of side effects)

Reasons to use reference semantics

- ensures an object passed always refers to the same object
- limit to the common analogy between pass by reference and pass pointers
    - passing pointers actually passes the pointers by value
    - NOTE: Java is a actually pass by value rather than commonly thought of as pass by reference (uses pointers behind the scenes)

- heap is used to keep commonly used objects around for longer and operate on them
- also, pointers are used for polymorphism
- NOTE: risk slicing
- review type erasure

Performance of value semantics

- value semantics are not quite as non-performant as commonly stated
- the copy constructor is often elided out by the compiler
- move semantics are added
- passing by value is not copying, compiler techniques:
  - copy elision
  - move construction
  - copy construction
  - the 'as-if' rule enables the compiler to add optimizations as long as the code performs the expected operations

NOTE: Reference Aliasing

- programmers default to using Object&
- passing by reference (const reference) causes at least two problems:
  - passing by reference to a container and another reference that is a points to an element of that container can modify the intended parameter within the function
  - compiler may still have to copy when it is unexpected
  - compiler has to cater to the possibility that an argument is an aliased reference