C++ Smart Pointers

- A smart pointer is an *object* that stores a pointer to a heap-allocated object
  - A smart pointer looks and behaves like a regular C++ pointer
    - By overloading `*`, `->`, `[]`, etc.
  - These can help you manage memory
    - The smart pointer will delete the pointed-to object *at the right time* including invoking the object's destructor
      - When that is depends on what kind of smart pointer you use
    - With correct use of smart pointers, you no longer have to remember when to `delete` `new`'d memory!

Introducing: `unique_ptr`

- A `unique_ptr` is the *sole owner* of its pointee
  - It will call `delete` on the pointee when it falls out of scope

Via the unique_ptr destructor

- Guarantees uniqueness by disabling copy and assignment

`std::shared_ptr`

- `shared_ptr` is similar to `unique_ptr` but we allow shared objects to have multiple owners
  - The copy/assign operators are not disabled and *increment* or *decrement* reference counts as needed
    - After a copy/assign, the two `shared_ptr` objects point to the same pointed-to object and the (shared) reference count is 2
  - When a `shared_ptr` is destroyed, the reference count is *decremented*
    - When the reference count hits 0, we `delete` the pointed-to object!

Some Important Smart Pointer Methods

Visit http://www.cplusplus.com/ for more information on these!

- `std::unique_ptr U;`

`U.get()`	Returns the raw pointer U is managing
`U.release()`	U stops managing its raw pointer and returns the raw pointer
`U.reset(q)`	U cleans up its raw pointer and takes ownership of q

- `std::shared_ptr S;`

`S.get()`	Returns the raw pointer S is managing
`S.use_count()`	Returns the reference count
`S.unique()`	Returns true iff S.use_count() == 1

- `std::weak_ptr W;`

`W.lock()`	Constructs a shared pointer based off of W and returns it
`W.use_count()`	Returns the reference count
`W.expired()`	Returns true iff W is expired (W.use_count() == 0)