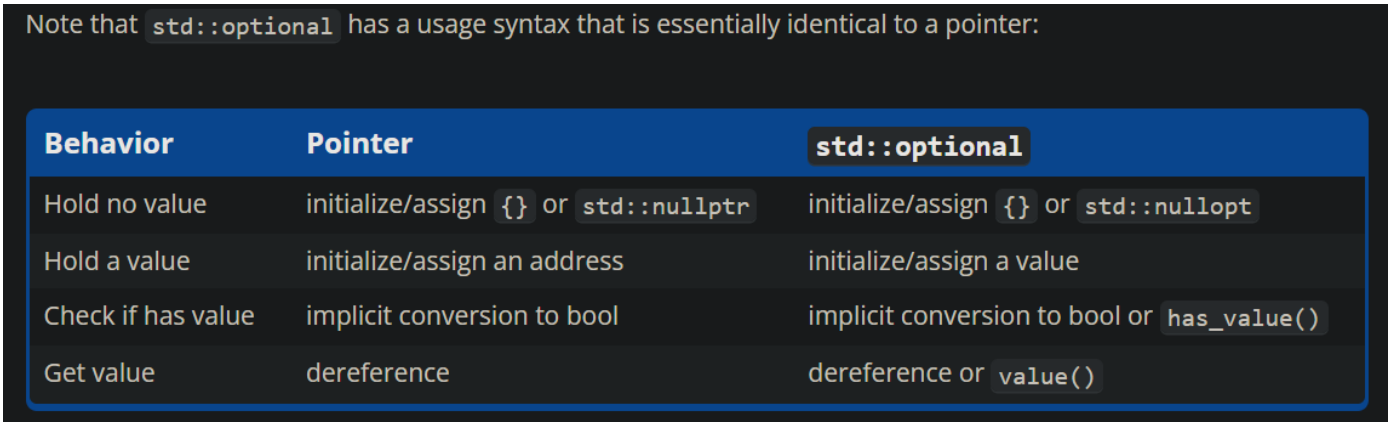
Compound Tyhpes: References and poiners

std::optional


Note that `std::optional` has a usage syntax that is essentially identical to a pointer:

Behavior	Pointer	`std::optional`
Hold no value	initialize/assign `{}` or `std::nullptr`	initialize/assign `{}` or `std::nullopt`
Hold a value	initialize/assign an address	initialize/assign a value
Check if has value	implicit conversion to bool	implicit conversion to bool or `has_value()`
Get value	dereference	dereference or `value()`

- A pointer has reference semantics, meaning it references some other object, and assignment copies the pointer, not the object. If we return a pointer by address, the pointer is copied back to the caller, not the object being pointed to. This means we can't return a local object by address, as we'll copy that object's address back to the caller, and then the object will be destroyed, leaving the returned pointer dangling.
- A std::optional has value semantics, meaning it actually contains its value, and assignment copies the value. If we return a std::optional by value, the std::optional (including the contained value) is copied back to the caller. This means we can return a value from the function back to the caller using std::optional.

Pros and cons of returning a std::optional

Returning a std::optional is nice for a number of reasons:

- Using std::optional effectively documents that a function may return a value or not.
- We don't have to remember which value is being returned as a sentinel.
- The syntax for using std::optional is convenient and intuitive.

Returning a std::optional does come with a few downsides:

- We have to make sure the std::optional contains a value before getting the value. If we dereference a std::optional that does not contain a value, we get undefined behavior.
- std::optional does not provide a way to pass back information about why the function failed.