Algebraic Data Types: Optional

Section 4

In this section

- What an `optional<T>` is
- `std::optional<T>`'s interface and semantics
- Example use cases

What is an optional?

Part 4.1

In this part

- What an optional represents
- Possible implementation
- *Optional* vs *pointers*

Meaning of `optional<T>`

- An `optional<T>` can fundamentally be in two states
  - **Set**: contains an instance of `T`
  - **Unset**: does not contain any instance of `T`
- It represents a "*value that may or may not be present*"
- Has *value semantics* and doesn't use dynamic allocations
- `sizeof(optional<T>)` is slightly bigger than `sizeof(T)`

Example: `std::optional` usage

```cpp
std::optional<int> o; // <== initially unset
assert(o.has_value() == false);

o = 15; // <== `o` is now set
assert(o.has_value() == true);
assert(o.value() == 15);

o = std::nullopt; // <== `o` is now unset
assert(o.has_value() == false);
```

Meaning of `optional<T>`

- *Intuition:* `optional<T>` is similar to `variant<T, nothing>`
- `optional<T>` can properly model:
  - Functions that can fail, where no extra error information is required
  - Absence of a result *(e.g. searching in a container)*
  - Non-mandatory *data members* and *arguments*
  - Deferred construction of an object

Meaning of `optional<T>`

```
optional<int> parse_string_to_int(const string&);
```

- If the string cannot be parsed, an *unset optional* is returned

```
optional<int> find_substr(const string&, const string&);
```

- If a substring match is found, its index is returned
- If there is no such match, an *unset optional* is returned
- Similarly to `variant`, the caller is forced to check the status of the returned `optional`

Meaning of `optional<T>`

```
struct person
{
    std::string _name;
    std::optional<employer> _employer;
};
```

- A `person` may or may not have an employer

Meaning of `optional<T>`

```cpp
struct service
{
    std::optional<request_processor> _rp;

    service()
    {
        // ... initialization steps ...
        _rp.emplace(); // ⟸ construct instance in `_rp`
    }
};
```

- Complicated classes can have their construction deferred, while still retaining safety and convenience of **RAII**

Memory layout of `optional<T>`

- Conceptually, `optional<T>` is just:
  - Storage for a `T` instance
  - `bool` that keeps track of whether or not the optional is set

```
template <typename T>
class optional
{
    std::aligned_storage_t<sizeof(T), alignof(T)> _data;
    bool is_set = false;
    // ...
};
```

Memory layout of `optional<T>`

- This means that **no dynamic allocation** is required, and that `optional<T>` **has value semantics**
- Inexpensive abstraction compared to a *smart pointer*, potentially cache-friendly
- `optional<T>` should be your **first choice** when you want to:
  - Represent possible absence of a value/parameter
  - Model a function that can fail/return nothing
  - Manually control the lifetime of an object

`optional<T>` vs *smart pointers*

- Both `optional<T>` and `std::unique_ptr<T>` can be used to control the lifetime of an object or represent absence of a `T` instance
- Using a *smart pointer* has several drawbacks:
  - **Loss of value semantics**
  - Overhead due to **dynamic allocation**
  - Overhead due to **indirection**

`optional<T>` vs *raw pointers*

- `T*` can be used to represent a "*potentially-null reference*" to an existing `T` instance
- Not all optional implementations support `optional<T&>`
- It is therefore *idiomatic* and *recommended* to use `T*` to:
    - Return/accept a reference that might be null
- `T*` should **never** own the memory it points to
    - Manual memory management is **unsafe** and superseded by *smart pointers*

`optional<T>` vs *raw pointers*

- If your implementation supports `optional<T&>`, use it to represent **non-owning nullable** references to `T`
  - It can be "more type-safe" than `T*`, as long as proper abstractions are used
  - Otherwise, `T*` is fine - but you must remember to explicitly check for `nullptr`

`std::optional` - basic interface

Part 4.2

In this part

- `std::optional`'s basic interface

std::optional

std::optional

Defined in header <optional>

```
template< class T >
class optional;
```

(since C++17)

- `std::optional<T>` is a *template class* that may or may not contain an instance of `T`
- The only requirement for `T` is `Destructible`

```
using maybe_int = std::optional<int>;
using maybe_str = std::optional<std::string>;
```

`std::optional` - default constructor

- The *default constructor* of `std::optional` will create an **unset** optional

```cpp
std::optional<int> o0;
// `o0` does not contain an instance of `int`

std::optional<float> o1;
// `o1` does not contain an instance of `float`
```

std::optional - nullopt

The Standard Library provides:

```
namespace std
{
    struct nullopt_t;
    inline constexpr nullopt_t nullopt{};
}
```

- `nullopt` can be used during `optional` *construction* or *assignment* to conveniently represent the "unset state"

`std::optional` - `nullopt` constructor

- Idential behavior to the *default constructor*, but takes a `std::nullopt_t` argument
- Useful in generic contexts and/or when we want to be *explicit* to the reader

```cpp
using foo = std::optional<int>;

// ...

foo f{std::nullopt}; // unset `optional<int>`
```

std::optional - U&& constructor

```
template <typename T>
template <typename U = T>
std::optional<T>::optional(U&& value);
```

- Initializes a **set** optional by **perfectly-forwarding** `value` in the optional's data storage
- `T` must be *constructible* from `U&&`

```
std::optional<int> o0{10};
std::optional<float> o1{42};
```

`std::optional` - copy/move constructors

Optionals of the same type can be copy/move-constructed

- The *target optional* will be in the same state as the *source optional*
- If the *source optional* contained a value, it will be copied/moved

```
std::optional<int> s0;
std::optional<int> s1{42};

auto d0 = s0; // ⟸ unset `optional<int>`
auto d1 = s1; // ⟸ set `optional<int>`, value `42`
```

std::optional - in-place constructor

```
template< class... Args >
constexpr explicit optional( std::in_place_t, Args&&... args );
```

```
template< class U, class... Args >
constexpr explicit optional( std::in_place_t,
                             std::initializer_list<U> ilist,
                             Args&&... args );
```

- `args ...` are perfectly-forwarded to construct the value in-place *(i.e. no unnecessary temporaries are created)*

```
std::optional<std::string> o5(std::in_place, 100, 'a');
// contains string with 100 'a' characters
```

`std::optional` assignment

`std::optional<T>` supports:

- Copy/move assignment
- Assigment from any `U` or `optional<U>`, where `U` can be used to construct `T`

```
std::optional<int> o0;
o0 = 42;

std::optional<int> o1;
o1 = o0;
```

`std::optional` - checking status

The current status of an `optional` can be checked with:

- `optional<T>::has_value()`
- `optional<T>::operator bool()`

```
std::optional<int> o0{42};
assert(o0.has_value());
assert(o0);

o0 = std::nullopt;
assert(o0.has_value() == false);
assert(!o0);
```

`std::optional` - accessing contained value

The value in a set `std::optional` can be accessed with:

- Unchecked access:
  - `std::optional<T>::operator*`
  - `std::optional<T>::operator->`
- Checked access:
  - `std::optional<T>::value`
- `std::optional<T>::value_or`

`std::optional` - unchecked access

- **Undefined behavior** if `has_value() == false`
- Pointer-line interface
- Useful when you are sure the optional contains a value

```
std::optional<std::string> o0{"hello"};
assert(*o0 == "hello");
assert(o0->size() == 5);

std::optional<int> o1;
foo(*o1); // Undefined behavior!
```

`std::optional` - checked access

- `.value()` returns a reference to the stored object, if any
- Otherwise, `std::bad_optional_access` is thrown

```cpp
std::optional<int> o0{42};
assert(o0.value() == 42);

std::optional<int> o1;
foo(o1.value()); // Throws `std::bad_optional_access`
```

std::optional - value_or

- Returns the contained value, if any
- Otherwise, returns the passed default value
- Useful to model choices with a predefined value

```cpp
std::optional<int> o0{42};
assert(o0.value_or(1000) == 42);

o0 = std::nullopt;
assert(o0.value_or(1000) == 1000);
```

std::optional - reset and emplace

- `.reset()` destroys the contained value, if any - otherwise it has no effects
- `.emplace( ... )` takes any number of arguments constructs an object in-place by perfectly-forwarding them

```cpp
std::optional<std::vector<char>> o;
o.emplace(20, 'a'); // `o` is a vector containing 20
                    // characters

o.reset();
assert(o.has_value() == false);
```

`std::optional` - use cases

Part 4.3

In this part

- When to use `std::optional`
- Simple failure cases
- Modeling optional data
- Controlling construction/destruction

When to use `std::optional`

- `optional<T>` is recommended for situation where there is a **single** and **obvious** reason to model the absence of a `T` value
- If there can be multiple reasons for the absence of a `T` value, choices such as `std::variant` or *exceptions* might be more appropriate

When to use `std::optional`

```cpp
std::optional<double> safe_sqrt(double x);
```

- One failure case: $x < 0$

```cpp
using connection_result =
    std::variant<success, timeout, invalid_address>;

connection_result connect_to(ip_address x);
```

- Multiple failure cases, with possible additional state

Example: parsing a string to `int`

```cpp
std::optional<int> parse_to_int(const std::string& s);
```

- Good use case for `optional`, unless more information about the failure is required
- Signature makes it clear that `nullopt` will be returned if `s` cannot be parsed as a valid `int`
- No need for special values / extra booleans / output parameters

Example: modeling optional data - `person`

```
struct person
{
    std::string _name;
    int _age;
    std::optional<phone_number> _home_number;
    std::optional<phone_number> _work_number;
};
```

- Some data might be inherently "optional"
- Instead of implementing "empty value" semantics for types like `phone_number`, `std::optional` is the appropriate choice

Example: modeling optional data - reading configuration

```cpp
std::optional<int> get_config_arg(const std::string& key);
```

- Possible scenario: reading from a `.ini` configuration file
- Some key-value pairs are not mandatory (or might have been forgotten)

```cpp
const auto speed = get_config_arg("speed").value_or(5);
```

- Elegant way of falling back to a *default value*

Controlling construction/destruction

- `std::optional<T>` can be useful to provide enough storage for a `T` instance, which has not yet been constructed
- Construction/destruction can be controlled manually with `.emplace( ... )` and `.reset()`
- Useful for controlling the lifetime of *active* objects

Example: delaying construction of an active object

- `async_port_listener` creates a new thread that listens to a port on construction, and joins the thread on destruction
- The user must have control over `async_port_listener`

Example: delaying construction of an active object

```cpp
struct state
{
    std::optional<async_port_listener> listener;
    // ...
};
```

```cpp
button_start.on_click([&] {
    _state.listener.emplace(port_entry.value());
});

button_stop.on_click([&] {
    _state.listener.reset();
});
```

Section recap

- `std::optional<T>` can be in two states: **set** or **unset**
- When *set*, it contains an instance of `T` in its internal storage - no indirection or dynamic allocation is used, has **value semantics**
- Usual implementation: `storage_for<T>` + `bool`
- `std::optional<T>` supports construction/assignment from types convertible to `T` and `std::nullopt`

Section recap

- `std::optional<T>` exposes a pointer-like interface (*`operator*`* *and* *`operator->`*) to access the internal value
  - The **behavior is undefined** if `.has_value() == false`
- It also exposes `.value()`, which **throws** if the optional is unset
- `o.value_or(default)` will return `o`'s value if set, `default` otherwise

Section recap

- `std::optional<T>` can be used to model:
  - Simple failure cases
  - Optional data or data that might not exist
  - Delayed construction of a `T` instance
- It is superior to alternatives such as *return codes/output parameters* or *dynamic allocation* both in terms of performance and type-safety
- When more information is needed, consider using `std::variant` instead

Exercise

- Implement a message protocol using algebraic data types
  - `exercise2.cpp`
    - on Wandbox
    - on Godbolt

Q&A

Break

5 minutes