Lecture 15:
Optional & Type Safety

CS106L, Fall 2026

Preston Seay && Emily Ai

Last Time: Move Semantics

- Recall: std::move enables moving—it does not itself move data

```
std::vector<int> a = {1, 2, 3};
std::vector<int> b = std::move(a);

// b can take ownership of a's resources.
// a is still valid, but its state is unspecified.
```

Today's Agenda

- Type Safety
  - How can types help us catch bugs?
- std::optional
  - How can we represent a missing value?
- Enums
  - How do we give a set of choices its own type?

Type Safety

What can the compiler catch before our code runs?

A problem caught at compile time

- ❌ We cannot pass a string where an int is required.

```
int divideByThree(int x) {
    return x / 3;
}

auto result = divideByThree("hello");
```

Key Idea: Types communicate what operations are allowed.

What does this function do?

```
void removeOddsFromEnd(std::vector<int>& v) {
    while (v.back() % 2 == 1) {
        v.pop_back();
    }
}
```

`vector::back()` returns a reference to the last element in the vector

`vector::pop_back()` is like the opposite of `vector::push_back(elem)`. It removes the last element from the vector.

What if the vector is empty?

- Calling std::vector::back() on an empty vector is undefined behavior

```
std::vector<int> a = {5, 7, 9};
removeOddsFromEnd(a);

// All elements removed ... what happens
// when the loop checks a.back() again?
```

One fix: check the precondition

- Short-circuit evaluation checks empty() before calling back()

```
void removeOddsFromEnd(std::vector<int>& v) {
    while (!v.empty() && v.back() % 2 == 1) {
        v.pop_back();
    }
}
```

What questions do you have?



bjarne_about_to_raise_hand

But what if a function might not have a value to return?

Consider a search function

```
// Returns the index of target if found.
int findIndex(const std::vector<int>& v, int target) {
    for (size_t i = 0; i < v.size(); ++i) {
        if (v[i] == target) return i;
    }
    return -1;
}
```

What does -1 really mean?

- A sentinel value is easy to forget and does not appear in the return type.

```
findIndex({2, 4, 6}, 4);  // 1
findIndex({2, 4, 6}, 9);  // -1

// But -1 is not a valid index!
// The caller must remember our convention.
```

What if the return type could say “maybe there is a value”?

std::optional

Definition: An optional either contains a T or contains nothing.

Value present

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

// a contains an int.
// a.has_value() is true.
```

No value

```
std::optional<int> b = std::nullopt;

int* p = std::nullopt; // ERROR - nullopt
   is not a pointer!

// b contains no int.
// b.has_value() is false.
```

What questions do you have?



bjarne_about_to_raise_hand

Creating and resetting optionals

- std::optional<T> is available starting in C++17

```
#include <optional>

std::optional<int> score = std::nullopt;
score = 106;              // now holds 106
score = std::nullopt;   // empty again
```

Checking for a value

```
std::optional<int> a = 5;
std::optional<int> b = std::nullopt;

std::cout << a.has_value();  // 1
std::cout << b.has_value();  // 0
```

Optionals work in conditions

- if (opt) is shorthand for if (opt.has_value()).

```
std::optional<int> result = 5;

if (result) {
    std::cout << "Found something!";
}

if (!result) {
    std::cout << "No value.";
}
```

Getting the value inside

- .value() throws std::bad_optional_access if empty; *opt on an empty optional is undefined behavior

```
std::optional<int> num = 106;

std::cout << num.value(); // 106
std::cout << *num;        // 106

// Always check before using *num!
```

What happens if no value exists?

- An optional makes absence explicit, but does not automatically prevent every mistake

```
std::optional<int> x = std::nullopt;

std::cout << x.value();
// Throws std::bad_optional_access.

std::cout << *x;
// Undefined behavior!
```

Let’s Practice w/ Slide 1!



What questions do you have?



bjarne_about_to_raise_hand

Using a default with value_or()

```
std::optional<int> a = 8;
std::optional<int> b = std::nullopt;

std::cout << a.value_or(99); // 8
std::cout << b.value_or(99); // 99
```

Your turn: predict the output

```
std::optional<int> a = std::nullopt;
std::optional<int> b = 5;

std::cout << a.value_or(10) << " ";
std::cout << b.value_or(10) << " ";
std::cout << b.has_value();
```

Your turn: predict the output

```
std::optional<int> a = std::nullopt;
std::optional<int> b = 5;

std::cout << a.value_or(10) << " "; // 10
std::cout << b.value_or(10) << " "; // 5
std::cout << b.has_value(); // 1
```

Let’s Practice w/ Slide 2!



What questions do you have?



bjarne_about_to_raise_hand

A safer alternative to vector::back()

```
// Hypothetical helper, NOT an STL method.
std::optional<int> safeBack(const std::vector<int>& v) {
    if (v.empty()) return std::nullopt;
    return v.back();
}
```

std::vector::back()

```
std::vector<int> v;

v.back();
// Undefined behavior if v is
   empty.
```

Our safeBack()

```
std::vector<int> v;

auto last = safeBack(v);
// Empty optional when v is
   empty.
```

Let’s Practice w/ Slide 3!



Returning an optional from a function

```
std::optional<size_t> findIndex(const std::vector<int>& v, int target) {
    for (size_t i = 0; i < v.size(); ++i) {
        if (v[i] == target) return i;
    }
    return std::nullopt;
}
```

Using the result safely

- Now the function signature communicates that searching might fail.

```
std::optional<size_t> index = findIndex({2, 4, 6}, 4);

if (index) {
    std::cout << "Found at " << *index;
} else {
    std::cout << "Not found!";
}
```

When is std::optional useful?

- Searching
  - A result may be missing.
- Parsing
  - An input might be invalid.
- Configuration
  - A setting might not be provided.

Trade-offs of optionals

- Advantages
  - No need for arbitrary sentinel values like −1.
  - The return type tells us a value might not exist.
- Trade-offs
  - Callers must handle the possibility of an empty optional.
  - .value() can throw an exception, while dereferencing an empty optional causes undefined behavior.

A C++23 aside: chaining optionals

and_then, transform, or_else

- These helpers let us compose operations while preserving missing values

```
// C++23
std::optional<int> x = 8;

// and_then: function returning optional
// transform: function returning a value
// or_else: fallback returning optional

auto squared = x.transform([](int n) {
    return n * n;
});
```

Let’s Practice w/ Slide 4!



Enums

Problem: magic numbers

```
int trafficLight = 2;

if (trafficLight == 2) {
    std::cout << "Go!";
}

// What does 2 mean?
```

What if each choice had a meaningful name?

Introducing enum class

- An enum class defines a new type with a fixed set of named values

```
enum class TrafficLight {
    Red,
    Yellow,
    Green
};

TrafficLight light = TrafficLight::Green;
```

Enums improve type safety

- Scoped enum values cannot be implicitly assigned from integers

```
TrafficLight light = TrafficLight::Red;

// This does not compile:
// light = 2;

// Write a meaningful named value:
light = TrafficLight::Green;
```

enum

```
enum Color { Red, Blue };
Color c = Red;
int n = c; // Allowed!

// Names leak into the surrounding
   scope.
```

enum class

```
enum class Color { Red, Blue };
Color c = Color::Red;
// int n = c; // Error!

// Scoped names and stronger
   checking.
```

Enums can have explicit values

- Under the hood, enum class does not implicitly become an int

```
enum class ErrorCode {
    None = 0,
    NotFound = 404,
    ServerError = 500
};

auto code = ErrorCode::NotFound;
int n = static_cast<int>(code); // 404
```



Let’s Practice w/ Slide 5!

Recap

What We Covered

- Type Safety
  - Use types to document valid operations and reduce mistakes.
- std::optional
  - Represent a value that may be absent; check before accessing.
- Enums
  - Use named, scoped choices instead of magic numbers.

Up Next:

RAII, Smart Pointers, & Building C++ Projects