CS106L Lecture 15:

`std::optional` & type safety!

Rachel Fernandez, Thomas Poimenidis

Attendance

https://tinyurl.com/lecture15cs106l

Plan

1. Recap
2. Type safety
3. `std::optional`

Recapping some shuff

Move semantics

- We have move semantics because sometimes the resource we're going to take is no longer needed by the original owner

```cpp
#include <iostream>
#include <vector>
#include <utility> // for std::move

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

    // We no longer need 'a', so let's move it into 'b'
    std::vector<int> b = std::move(a);

    std::cout << "a.size() = " << a.size() << "\n"; // 0
    std::cout << "b.size() = " << b.size() << "\n"; // 3

    return 0;
}
```

Recapping some stuff

Move semantics

- We have move semantics because sometimes the resource we're going to take is no longer needed by the original owner

```cpp
#include <iostream>
#include <vector>
#include <utility> // for std::move

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

    // We no longer need 'a', so let's move it into 'b'
    std::vector<int> b = std::move(a);

    std::cout << "a.size() = " << a.size() << "\n"; // 0
    std::cout << "b.size() = " << b.size() << "\n"; // 3

    return 0;
}
```

- Use **`std::move(a)`** to turn **`a`**, an l-value, to an r-value so that you can immediately take its resources

Recapping some shtuff

Move semantics

- We have move semantics because sometimes the resource we’re going to take is no longer needed by the original owner
- Use `std::move(a)` to turn `a`, an l-value, to an r-value so that you can immediately take its resources
- **Rule of zero:** if you have self-managing member variables, and don’t need to define custom constructors, and operators, then don’t!

Rule of ZERO

Rule of zero: if you have self-managing member variables, and don't need to define custom constructors, and operators, then don't!

```cpp
#include <string>
#include <vector>

class Student {
public:
    // We don't write:
    // - destructor
    // - copy constructor
    // - copy assignment operator
    // - move constructor
    // - move assignment operator

    // Why? Because std::string and std::vector manage themselves!
    Student(std::string name, std::vector<int> scores)
        : name_(std::move(name)), scores_(std::move(scores)) {}

private:
    std::string name_;        // self-managing
    std::vector<int> scores_; // self-managing
};
```

C++ automatically gives us the following (if we don't define our own)

1. Destructor
2. Copy constructor
3. Copy assignment operator
4. Move constructor
5. Move assignment operator

These work great here!

Rule of ZERO

Rule of zero: if you have self-managing member variables, and don't need to define custom constructors, and operators, then don't!

```
#include <string>
#include <vector>

class Student {
public:
    // We don't write:
    // - destructor
    // - copy constructor
    // - copy assignment operator
    // - move constructor
    // - move assignment operator

    // Why? Because std::string and std::vector manage themselves!
    Student(std::string name, std::vector<int> scores)
        : name_(std::move(name)), scores_(std::move(scores)) {}

private:
    std::string name_;       // self-managing
    std::vector<int> scores_; // self-managing
};
```

C++ automatically gives us the following (if we don't define our own)

1. Destructor:
   `~Student();`
2. Copy constructor
   `Student(const Student& other);`
3. Copy assignment operator
   `Student& operator=(const Student& other);`
4. Move constructor
   `Student(Student&& other);`
5. Move assignment operator
   `Student& operator=(Student&& other);`

Recapping some shtuff

Move semantics

- We have move semantics because sometimes the resource we're going to take is no longer needed by the original owner
- Use **`std::move(x)`** to turn **`x`**, an l-value, to an r-value so that you can immediately take its resources
- **Rule of zero:** if you have self-managing member variables, and don't need to define custom constructors, and operators, then don't!
- **Rule of three:** if you define a custom destructor then you need to also define a custom copy constructor and copy assignment operator.

Rule of THREE

Rule of three: if you define a custom destructor then you need to also define a custom copy constructor and copy assignment operator.

```cpp
class Student {
public:
    Student(const std::string& name, int numScores)
        : name_(name), numScores_(numScores), scores_(new int[numScores]) {}

    // 1. Destructor – must free the array
    ~Student() {
        delete[] scores_;
    }

    // 2. Copy constructor – deep copy the array
    Student(const Student& other)
        : name_(other.name_), numScores_(other.numScores_) {

        scores_ = new int[numScores_];          // allocate
        for (int i = 0; i < numScores_; i++)   // deep copy
            scores_[i] = other.scores_[i];
    }
```

```cpp
    // 3. Copy assignment – deep copy + avoid self-assignment
    Student& operator=(const Student& other) {
        if (this != &other) {
            // free old resource
            delete[] scores_;

            // copy non-resource fields
            name_ = other.name_;
            numScores_ = other.numScores_;

            // deep copy array
            scores_ = new int[numScores_];
            for (int i = 0; i < numScores_; i++)
                scores_[i] = other.scores_[i];
        }
        return *this;
    }

private:
    std::string name_;
    int numScores_;
    int* scores_;       // RAW pointer → not self-managing!
};
```

Recapping some shtuff

Move semantics

- We have move semantics because sometimes the resource we're going to take is no longer needed by the original owner
- Use **`std::move(x)`** to turn **`x`**, an l-value, to an r-value so that you can immediately take its resources
- **Rule of zero:** if you have self-managing member variables, and don't need to define custom constructors, and operators, then don't!
- **Rule of three:** if you define a custom destructor then you need to also define a custom copy constructor and copy assignment operator.
- **Rule of Five:** If you have a custom copy constructor, and copy assignment operator, then you should also define a move constructor and a move assignment operator!

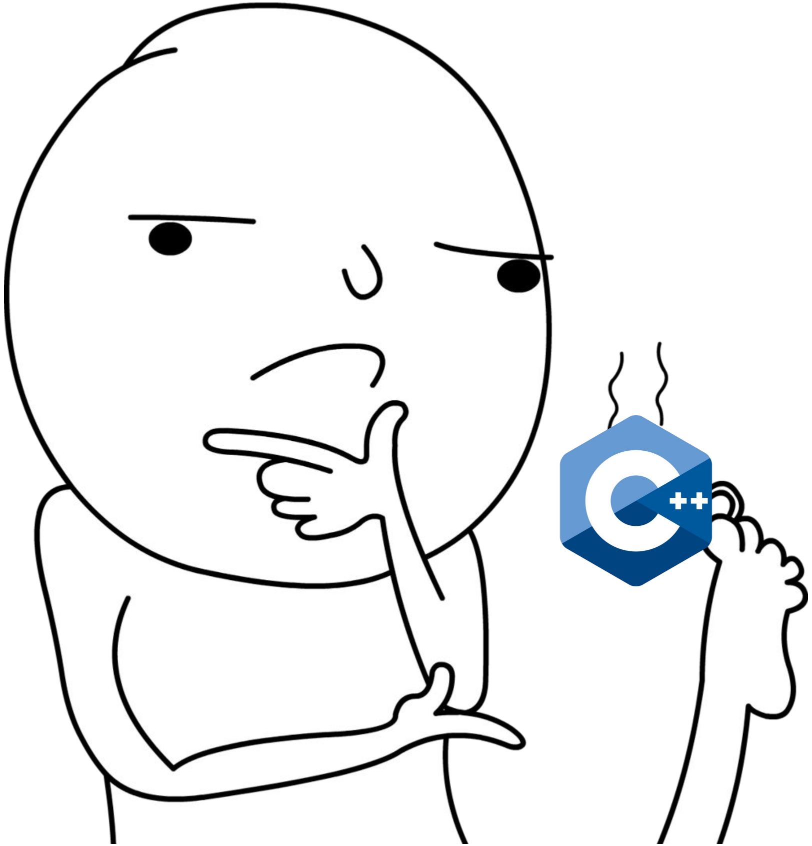
Rule of FIVE

Rule of Five: If you have a custom copy constructor, and copy assignment operator, then you should also define a move constructor and a move assignment operator!

```
    // Move constructor
    Student(Student&& other)
        : name_(std::move(other.name_)),
        numScores_(other.numScores_),
        scores_(other.scores_) {

        other.scores_ = nullptr;
        other.numScores_ = 0;
    }
```

```
    // Move assignment
    Student& operator=(Student&& other) {
        if (this != &other) {
            delete[] scores_;

            name_ = std::move(other.name_);
            numScores_ = other.numScores_;
            scores_ = other.scores_;

            other.scores_ = nullptr;
            other.numScores_ = 0;
        }
        return *this;
```

What questions do we have?



A definition!

Type Safety: The extent to which a language prevents typing errors.

Python (english) vs. C++

Python

```
def div_3(x):
    return x / 3
div_3(“hello”)

//CRASH during runtime,
can’t divide a string
```

C++

```
int div_3(int x){
    return x / 3;
}
div_3(“hello”)
//Compile error: this code
will never run
```

Python (english) vs. C++

Type Safety: The extent to which a language **guarantees the behavior of programs.**

What does this code do?

```
void removeOddsFromEnd(vector<int>& vec){
    while(vec.back() % 2 == 1){
        vec.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.

Anyone see a problem?

```
void removeOddsFromEnd(vector<int>& vec){
    while(vec.back() % 2 == 1){
        vec.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.

Anyone see a problem?

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

Hint!

`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.

Anyone see a problem?

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

What if **vec** is { } / an empty vector!?

`std::vector` documentation

std::vector<T,Allocator>::back

`reference back();`	(until C++20)
`constexpr reference back();`	(since C++20)
`const_reference back() const;`	(until C++20)
`constexpr const_reference back() const;`	(since C++20)

Returns a reference to the last element in the container.

Calling back on an empty container causes undefined behavior.

Undefined behavior: Function could crash, could give us garbage, could accidentally give us some actual value

Taking another look at our code

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

We can make no guarantees about what this function does!

Credit to Jonathan Müller of foonathan.net for the example!

One solution

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

One solution

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

Key idea: it is the **programmers job** to enforce the **precondition** that **vec** be non-empty, otherwise we get undefined behavior!

There may or may not be a “last element” in vec

How can `vec.back()` have deterministic behavior in either case?

The problem

```
valueType& vector<valueType>::back(){
    return *(begin() + size() - 1);
}
```

What happens if size() = 0?

Dereferencing a pointer without
verifying it points to real memory is undefined behavior!

The problem

```
valueType& vector<valueType>::back(){
   if(empty()) throw std::out_of_range;
   return *(begin() + size() - 1);
}
```

Now, we will at least reliably error and stop the program **or** return the last element whenever back( ) is called

The problem

Deterministic behavior is great, but can we do better?

There may or may not be a “last element” in vec
How can `vec.back()` warn us of that when we call it?

Revisiting our definition

Type Safety: The extent to which a **function signature** guarantees the behavior of a **function**.

Back to the problem

```
valueType& vector<valueType>::back(){
    return *(begin() + size() - 1);
}
```

`back()` is promising to return something of type **`valueType`** when its possible no such value exists!

A look at a first solution

```
std::pair<bool, valueType&> vector<valueType>::back(){
    if(empty()){
        return {false, valueType()};
    }
    return {true, *(begin() + size() - 1)};
}
```

back() now advertises that there may or may not be a last element

A look at a first solution

```
std::pair<bool, valueType&> vector<valueType>::back(){
    if(empty()){
        return {false, valueType()};
    }
    return {true, *(begin() + size() - 1)};
}
```

Default constructor of **`valueType()`**

`back()` now advertises that there may or may not be a last element

Problems with std::pair

```
std::pair<bool, valueType&> vector<valueType>::back(){
    if(empty()){
        return {false, valueType()};
    }
    return {true, *(begin() + size() - 1)};
}
```

- **valueType** may not have a default constructor :(((

Problems with std::pair

```
std::pair<bool, valueType&> vector<valueType>::back(){
    if(empty()){
        return {false, valueType()};
    }
    return {true, *(begin() + size() - 1)};
}
```

- **valueType** may not have a default constructor
- Even if it does, calling constructors is **expensive**

Problems with std::pair

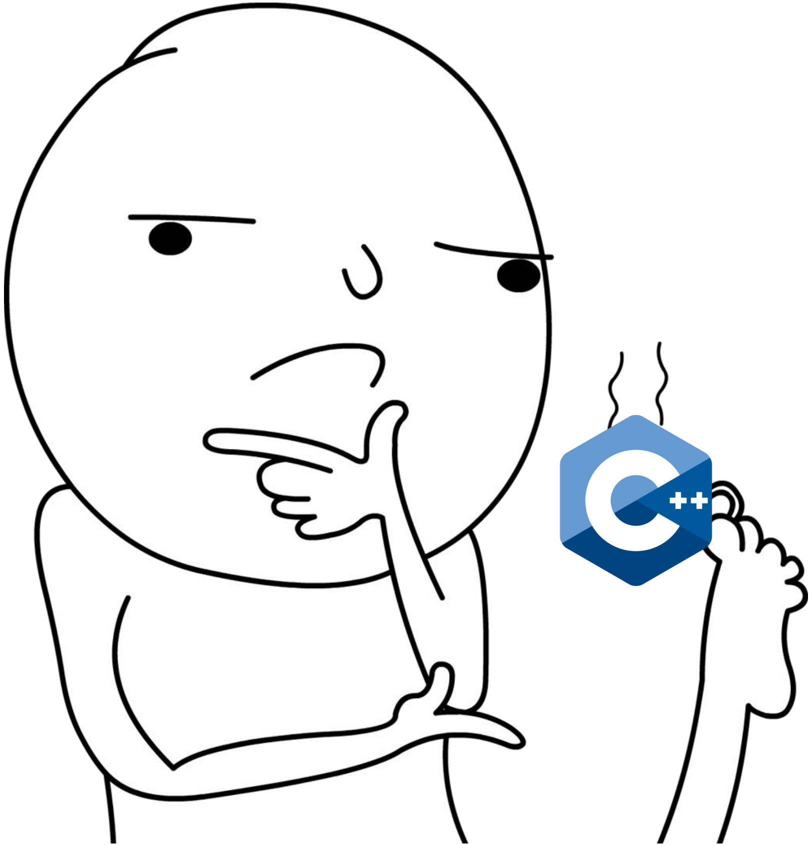
```
void removeOddsFromEnd(vector<int>& vec){
    while(vec.back().second % 2 == 1){
        vec.pop_back();
    }
}
```

This is still pretty unpredictable behavior! What if the default constructor for an int produced an **odd number?**

What should back return in this case?

```
??? vector<valueType>::back(){
    if(empty()){
        return ??;
    }
    return *(begin() + size() - 1);
}
```

What questions do we have?



Introducing `std::optional`

What is **std::optional<T>**

- **std::optional** is a template class which will either contain a value of type **T** or contain nothing (expressed as **nullopt**)

std::optional

Defined in header <optional>

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

(since C++17)

The class template std::optional manages an optional contained value, i.e. a value that may or may not be present.

A common use case for optional is the return value of a function that may fail. As opposed to other approaches, such as std::pair<T, bool>, optional handles expensive-to-construct objects well and is more readable, as the intent is expressed explicitly.

What is `std::optional<T>`

- **`std::optional`** is a template class which will either contain a value of type **`T`** or contain nothing (expressed as **`nullopt`**)

Note: that's `nullopt` NOT `nullptr`. It's a new thing!

`nullptr`: an object that can be converted to a value of any **pointer** type

`nullopt`: an object that can be converted to a value of any **optional** type

What is `std::optional<T>`

- **`std::optional`** is a template class which will either contain a value of type **T** or contain nothing (expressed as **`nullopt`**)

```cpp
int* p = nullptr;    // p points to nothing
if (p == nullptr) {
    std::cout << "p is a null POINTER\n";
}

std::optional<int> x = nullptr;    // ERROR — nullptr is NOT for optionals
```

```cpp
std::optional<int> x = std::nullopt;    // x contains nothing
if (!x) {
    std::cout << "x is an EMPTY OPTIONAL\n";
}

int* p = std::nullopt;    // ERROR — nullopt is NOT a pointer
```

Note: that's `nullopt` NOT `nullptr`. It's a new thing!

`nullptr`: an object that can be converted to a value of any **pointer** type

`nullopt`: an object that can be converted to a value of any **optional** type

What is `std::optional<T>`

- **`std::optional`** is a template class which will either contain a value of type **T** or contain nothing (expressed as **`nullopt`**)

```
void main(){
    std::optional<int> num1 = {}; //num1 does not have a value
    num1 = 1; //now it does!
    num1 = std::nullopt; //now it doesn't anymore
}
```

Can be used interchangeably!

What is std::optional<T>

```
std::optional<valueType> vector<valueType>::back(){
    if(empty()){
        return {};
    }
    return *(begin() + size() - 1);
}
```

What using back( ) look like:

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

We can’t do arithmetic with an optional, we have to get the value inside the optional (if it exists) first!

What’s the interface of `std::optional`?

`std::optional` types have a:

- `.value()` method:
  returns the contained value or throws `bad_optional_access` error

What's the interface of `std::optional`?

`std::optional` types have a:

- **`.value()`** method:
  returns the contained value or throws **`bad_optional_access`** error

- **`.value_or(valueType val)`**

  returns the contained value or default value, parameter **`val`**

What's the interface of `std::optional`?

`std::optional` types have a:

- **`.value()`** method:
  returns the contained value or throws **`bad_optional_access`** error

- **`.value_or(valueType val)`**

  returns the contained value or default value, parameter **`val`**

- **`.has_value()`**

  returns **`true`** if contained value exists, **`false`** otherwise

What’s the interface of `std::optional`?

```cpp
#include <iostream>
#include <optional>

int main() {
    std::optional<int> a = 5;
    std::optional<int> b = std::nullopt;

    // -------------------------
    // 1. has_value()
    // -------------------------
    std::cout << "a.has_value(): " << a.has_value() << "\n";  // 1 (true)
    std::cout << "b.has_value(): " << b.has_value() << "\n";  // 0 (false)
```

What's the interface of `std::optional`?

```
    // -------------------------
    // 2. value()
    // -------------------------
    if (a.has_value()) {
        std::cout << "a.value(): " << a.value() << "\n";        // 5
    }

    // Uncommenting this would throw bad_optional_access:
    // std::cout << b.value() << "\n";
```

What's the interface of `std::optional`?

```
    // Uncommenting this would throw bad_optional_access:
    // std::cout << b.value() << "\n";

    // -------------------------
    // 3. value_or(default)
    // -------------------------
    std::cout << "a.value_or(999): " << a.value_or(999) << "\n";  // 5
    std::cout << "b.value_or(999): " << b.value_or(999) << "\n";  // 999

    return 0;
```

Revisiting back()

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

Now, if we access the back of an empty vector, we will at least reliably get the **bad_optional_access** error

Revisiting back( )

```
void removeOddsFromEnd(vector<int>& vec){
    while(vec.back().has_value() && vec.back().value() % 2 == 1){
        vec.pop_back();
    }
}
```

This will no longer error, but it is pretty unwieldy :/

Revisiting back()

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

Better? You can just call `vec.back()` since `nullopt` is falsy!

Recap: The problem with **`std::vector::back()`**

- Why is it so easy to accidentally call **`back()`** on empty vectors if the outcome is so dangerous?
- The function signature gives us a false promise!

```
valueType& vector<valueType>::back()
```

- Promises to return an something of type **`valueType`**
- But in reality, there either may or may not be a “last element” in a vector

An `optional` take on `realVector`

More bad code!

```
int foo(vector<int>& vec){
    return vec[0];
}
```

What happens if `vec` is empty? More undefined behavior!

std::optional<T&> is not available!

```
std::optional<valueType&>
vector<valueType>::operator[](size_t index){
    if (index < size()) {
            return *(begin() + index);
    }
            return std::nullopt;
}
```

std::optional<T&> is not available!

```
std::optional<valueType&>
vector<valueType>::operator[](size_t index){
    if (index < size()) {
        return *(begin() + index);
    }
        return std::nullopt;
}
```

A reference must be to a valid object, and `optional` doesn't guarantee that, think about having an optional to a `nullopt`

std::optional<T&> is not available!

```
std::optional<valueType&>
vector<valueType>::operator[](size_t index){
   if (index < size()) {
         return *(begin() + index);
   }
         return std::nullopt;
}
```

```
int main() {
    vector <int> v;
    v.data = {10, 20, 30};

    auto optRef = v[5];   // returns std::nullopt
    // ERROR: int& must store a reference to a real integer, not std::nullopt
}
```

Best we can do is error..which is what .at() does

```
valueType& vector<valueType>::operator[](size_t index){
    return *(begin() + index);
}
valueType& vector<valueType>::at(size_t index){
    if(index >= size()) throw std::out_of_range;
    return *(begin() + index);
}
```

🤔 Why have both?

Is this.....good?

Pros of using **`std::optional`** returns:

- Function signatures create more informative contracts
- Class function calls have guaranteed and usable behavior

Cons:

- You will need to use **`.value()`** EVERYWHERE
- (In cpp) It’s still possible to do a **`bad_optional_access`**
- (In cpp) optionals can have undefined behavior too (**`*optional`** does same thing as **`.value()`** with no error checking)
- In a lot of cases we want **`std::optional<T&>`**...which we don’t have

Why even bother with optionals?

Is this.....good?

- **`.and_then(function f)`**

  returns the result of calling `f(value)` if contained value exists, otherwise `nullopt` (f must return `optional`)

.and_then(function f)

```
#include <iostream>
#include <optional>

std::optional<int> half(int x) {
    if (x % 2 == 0) return x / 2;
    return std::nullopt;
}

int main() {
    std::optional<int> a = 8;

    auto result = a.and_then(half)       // 8 → 4
                    .and_then(half)     // 4 → 2
                    .and_then(half);    // 2 → 1

    if (result)
        std::cout << *result;   // prints 1

    std::optional<int> b = 7;

    auto result2 = b.and_then(half);  // 7 is odd → nullopt

    if (!result2)
        std::cout << "\nhalf(7) failed!\n";
}
```

Is this.....good?

- **`.and_then(function f)`**

  returns the result of calling `f(value)` if contained value exists, otherwise `nullopt` (f must return `optional`)

- **`.transform(function f)`**

  returns the result of calling `f(value)` if contained value exists, otherwise `nullopt` (f must return `optional<valueType>`)

.transform(function f)

```
#include <iostream>
#include <optional>

int square(int x) { return x * x; }

int main() {
    std::optional<int> x = 5;

    auto y = x.transform(square);

    if (y)
        std::cout << *y;      // prints 25

    std::optional<int> z = std::nullopt;

    auto w = z.transform(square);  // z is empty → nullopt

    if (!w)
        std::cout << "\nsquare(nullopt) = nullopt\n";
}
```

Is this.....good?

- **`.and_then(function f)`**

  returns the result of calling `f(value)` if contained value exists, otherwise `nullopt` (f must return `optional`)

- **`.transform(function f)`**

  returns the result of calling `f(value)` if contained value exists, otherwise `nullopt` (f must return `optional<valueType>`)

- **`.or_else(function f)`**

  returns value if it exists, otherwise returns result of calling f

.or_else(function f)

```cpp
#include <iostream>
#include <optional>

std::optional<int> fallback() {
    return 42;
}

int main() {
    std::optional<int> good = 10;
    std::optional<int> bad  = std::nullopt;

    auto r1 = good.or_else(fallback);  // returns optional(10)
    auto r2 = bad.or_else(fallback);   // returns optional(42)

    std::cout << "r1 = " << *r1 << "\n";  // 10
    std::cout << "r2 = " << *r2 << "\n";  // 42
}
```

Is this.....good?

- `.and_then(fu`

  returns the
  otherwise n

- `.transform(f`

  returns the
  otherwise n

- `.or_else(fun`

  returns valu

Monadic: a software design pattern with a structure that combines program fragments (functions) and wraps their return values in a type with additional computation

These all let you try a function and will either return the result of the computation or some default value.

Is this.....good?

- `.and_then(function f)`

  returns the result of calling `f(value)` if contained value exists, otherwise `null_opt` (f must return `optional`)

- `.transform(function f)`

  returns the result of calling `f(value)` if contained value exists, otherwise `null_opt` (f must return `optional<valueType>`)

- `.or_else(function f)`

  returns value if it exists, otherwise returns result of calling f

Revisiting our back( ) code…again!

```
void removeOddsFromEnd(vector<int>& vec){
    auto isOdd = [](optional<int> num){
        if(num)
            return num % 2 == 1;
        else
            return std::nullopt;
        //return num ? (num % 2 == 1) : {};
    };
    while(vec.back().and_then(isOdd)){
        vec.pop_back();
    }
}
```

Revisiting our back( ) code...again!

```
void removeOddsFromEnd(vector<int>& vec){
   auto isOdd = [](optional<int> num){
       if(num)
           return num % 2 == 1;
       else
           return std::nullopt;
       //return num ? (num % 2 == 1) : {};
   };
   while(vec.back().and_then(isOdd)){
       vec.pop_back();
   }
}
```

Recall lambda functions!

Disclaimer: std::vector::back() doesn’t actually return an optional
(and probably never will)

Recall: Design philosophies of C++

- Only add features if they solve an actual problem
- Programmers should be free to choose their style
- Compartmentalization is key
- Allow the programmer full control if they want it
- Don’t sacrifice performance except as a last resort
- Enforce safety at compile time whenever possible

Recall: Design philosophies of C++

- **Only add features if they solve an actual problem**
- **Programmers should be free to choose their style**
- Compartmentalization is key
- **Allow the programmer full control if they want it**
- Don’t sacrifice performance except as a last resort
- **Enforce safety at compile time whenever possible**

Languages that *really* use ~~optional~~ monads

- Rust 🥰😍

    Systems language that guarantees memory and thread safety

- Swift

    Apple’s language, made especially for app development

- JavaScript

    Everyone’s favorite

Recap: Type safety and `std::optional`

- You can guarantee the behavior of your programs by using a strict type system!

Recap: Type safety and `std::optional`

- You can guarantee the behavior of your programs by using a strict type system!
- **`std::optional`** is a tool that could make this happen: you can return either a value or nothing: **`.has_value()`**, **`.value_or()`**, **`.value()`**

Recap: Type safety and `std::optional`

- You can guarantee the behavior of your programs by using a strict type system!
- **`std::optional`** is a tool that could make this happen: you can return either a value or nothing: **`.has_value()`**, **`.value_or()`**, **`.value()`**
- This can be unwieldy and slow, so cpp doesn't use optionals in most stl data structures

Recap: Type safety and `std::optional`

- You can guarantee the behavior of your programs by using a strict type system!
- **`std::optional`** is a tool that could make this happen: you can return either a value or nothing: **`.has_value()`**, **`.value_or()`**, **`.value()`**
- This can be unwieldy and slow, so cpp doesn't use optionals in most stl data structures
- Many languages, however, do!

Recap: Type safety and `std::optional`

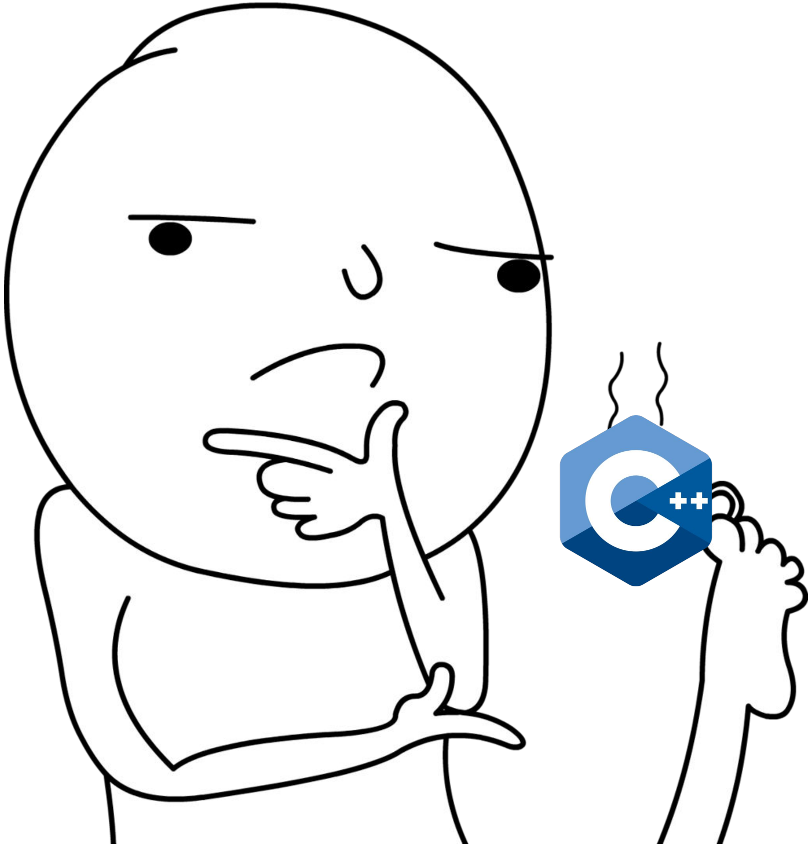
- You can guarantee the behavior of your programs by using a strict type system!
- **`std::optional`** is a tool that could make this happen: you can return either a value or nothing: **`.has_value()`**, **`.value_or()`**, **`.value()`**
- This can be unwieldy and slow, so cpp doesn't use optionals in most stl data structures
- Many languages, however, do!
- Besides using them in classes, you can use them in application code where it makes sense! This is highly encouraged :)

All in all

“Well typed programs cannot go wrong.”

- Robert Milner (very important and good CS dude)

What questions do we have?



Let’s look at some code

https://tinyurl.com/lecture15practice