

CS106L Lecture 16:

`std::optional` & type safety!

Fabio Ibanez, Jacob Roberts-Baca

Attendance



<https://tinyurl.com/optionalS25>

Plan

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

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

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

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!

Recapping some shtuff

Move semantics

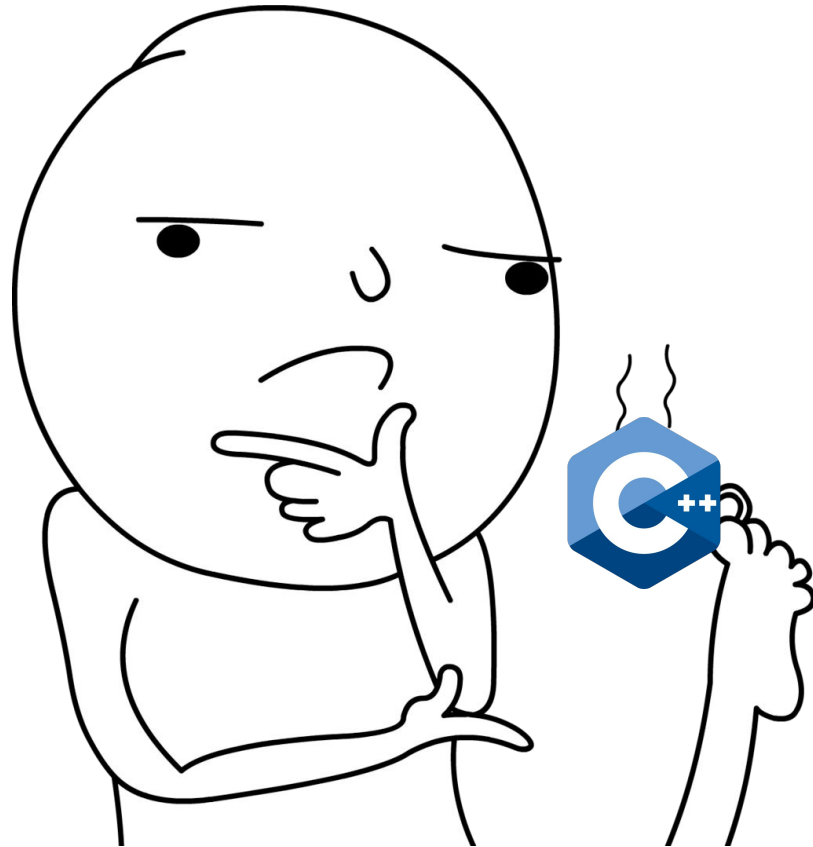
- We have move semantics because sometimes the resource we're going to take is no longer needed by the original owner
- Use **`std::move(x)`** to turn **`x`**, an l-value, to an r-value so that you can immediately take its resources
- **Rule of zero:** if you have self-managing member variables, and 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.

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!

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);  
}
```

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!

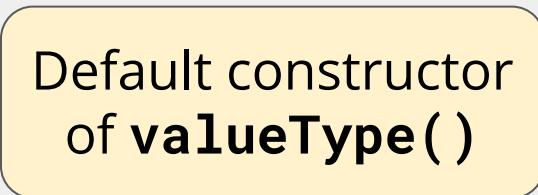
A look at a first solution

```
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`

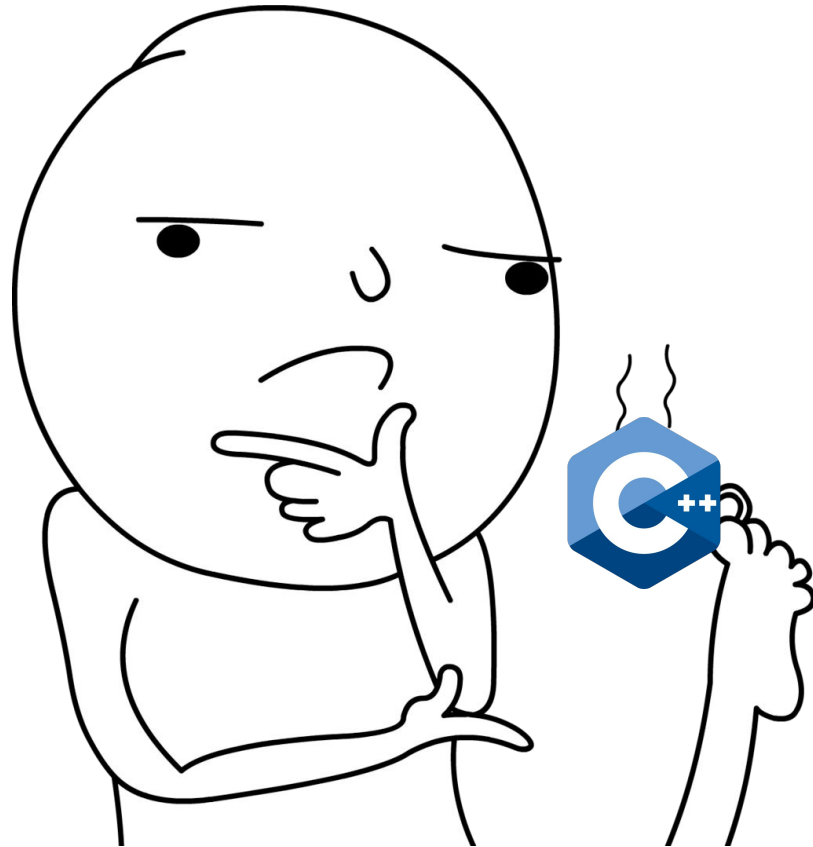
```
void removeOddsFromEnd(vector<int>& vec){  
    while(vec.back().second % 2 == 1){  
        vec.pop_back();  
    }  
}
```

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`**)

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!

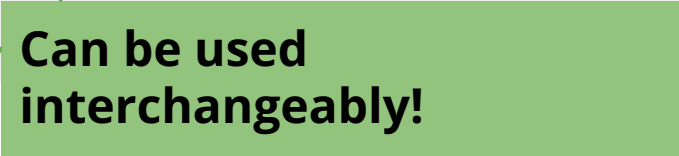
`nullptr`: an object that can be converted to a value of any **pointer** type

`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`**)

```
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

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){  
    return *(begin() + index);  
}
```

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

Is this.....good?

- `.and_then(f)` returns the result of `f` if the value is not `None`, otherwise `None`
 - `.transform(f)` returns the result of `f` if the value is not `None`, otherwise `None`
 - `.or_else(f)` returns the result of `f` if the value is `None`, otherwise the value
- 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`

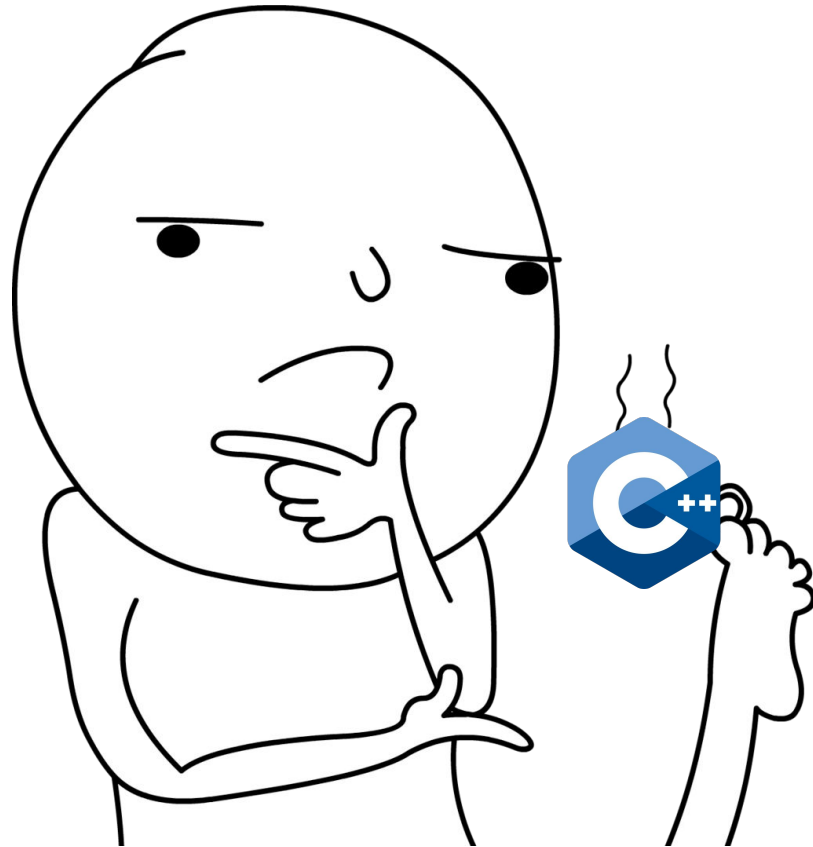
- You can guarantee the behavior of your programs by using a strict type system!
- **`std::optional`** is a tool that could make this happen: you can return either a value or nothing: **`.has_value()`**, **`.value_or()`**, **`.value()`**
- This can be unwieldy and slow, so cpp doesn't use optionals in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