

Type Safety and `std::optional` ...

How can we use c++'s type system to prevent errors at compile time?

Attendance

bit.ly/3WuEsxv



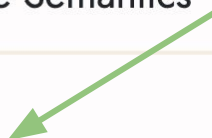
Announcements

Announcements

- Only 2 lectures left (including today)!

8	<p>MAY 23</p> <p>13. Move Semantics</p>	<p>MAY 25</p> <p>14. std::optional and Type Safety</p>
9	<p>MAY 30</p> <p>15. RAII, Smart Pointers, and Building C++ Projects</p>	<p>JUNE 1</p> <p>Optional: No Class, Extra Office Hours</p>
10	<p>JUNE 6</p> <p>Optional: No Class, Extra Office Hours</p>	<p>JUNE 8</p> <p>No class, No office hours</p>

**Can start
assignment 2
after this lecture**



Important Announcements

- **Wednesday, June 7th at 11:59pm PT is the last day we can accept any assignments (1 or 2)**
- **We want everyone to pass!** Please turn in your assignments! Come to office hours, post on Ed, email us to get help!
- **Reminder you need to complete assignment 1 and 2 without build errors to pass the class**

Check Ed after reading Assn 2 Handout for Extra Tips

Assignment 2 Tips and Tricks (Firm Deadline: Wed, June 7th @ 11:59pm) #23



Sarah McCarthy **STAFF**

2 days ago in **General**



UNPIN



STAR



WATCHING

9

VIEWS



(Normally I would send this out next Tuesday but seems like some people may be starting a bit early so sending out now)

Hi guys!

Just wanted to share some tips/help/guidance for assignment 2. **This is not a substitute for reading the handout and code but something to help nudge you in the right direction.** Giving the handout a close read is worth your time. We discuss many important notes to help you get through the assignment faster.

Today



- Recap: Const-correctness
- Type Safety
- The need for “sometimes-a-thing”
 - `std::optional`

Today



**But first! A quick story about
move semantics**

- type safety
- The need for
“sometimes-a-thing”
- `std::optional`



Haven's birthday is coming up! I know! I'll make a copy of my favorite truck to give to her!





To: Haven



Better get started on
making the copy!





Oh wow! Making a truck from scratch actually takes a lot of time and resources



To: Haven



Wait! I don't need to build a copy. I can just give Haven the original



To: Haven



But what if I want to play
with my truck again later?





You can't! There's no take backs with move semantics! Once you've said you don't need it anymore, you can't get it back.





To: Haven



That's why we need both
move and copy
constructors/assignment
operators.





To: Haven



Sometimes we want to play with our truck after gifting it, so we need to make a copy



To: Haven



Other times, we're done playing with our truck, so we can gift the original! (yay for saving time and resources!)



TLDR: Move Semantics

- Move semantics is a way to make copying things faster and more efficient
- Using move semantics tells the program “you can use this now, I don’t need it anymore”

TLDR: Move Semantics

- Move semantics is a way to make copying things faster and more efficient
- Using move semantics tells the program “you can use this now, I don’t need it anymore”
- If your class has **copy constructor** and **copy assignment** defined, you should also define a **move constructor** and **move assignment**
- Define these by overloading your copy constructor and assignment to be defined for `Type&& other` as well as `Type& other`

TLDR: Move Semantics

- Move semantics is a way to make copying things faster and more efficient
- Using move semantics tells the program “you can use this now, I don’t need it anymore”
- If your class has **copy constructor** and **copy assignment** defined, you should also define a **move constructor** and **move assignment**
- Define these by overloading your copy constructor and assignment to be defined for `Type&& other` as well as `Type& other`
- Use `std::move` to force the use of other types’ move assignments and constructors
- All `std::move(x)` does is cast `x` as an rvalue
- Be wary of `std::move(x)` in main function code!

Today



- Recap: Const-correctness
- Type Safety
- The need for “sometimes-a-thing”
 - `std::optional`

Recap: Const-Correctness

- We pass big pieces of data **by reference** into helper functions by to avoid making copies of that data

Recap: Const-Correctness

- We pass big pieces of data **by reference** into helper functions by to avoid making copies of that data
- If this function accidentally or sneakily changes that piece of data, it can lead to hard to find bugs!

Recap: Const-Correctness

- We pass big pieces of data **by reference** into helper functions to avoid making copies of that data
- If this function accidentally or sneakily changes that piece of data, it can lead to hard to find bugs!
- **Solution:** mark those reference parameters `const` to guarantee they won't be changed in the function!

How does the compiler know when it's safe to call
member functions of `const` variables?

Definition

const-interface: All member functions marked `const` in a class definition. Objects of type `const ClassName` may only use the **const-interface**.

RealVector's const-interface

```
template<class ValueType> class RealVector {  
public:  
    using iterator = ValueType*;  
    using const_iterator = const ValueType*;  
    /*...*/  
    size_t size() const;  
    bool empty() const;  
    /*...*/  
    void push_back(const ValueType& elem);  
    iterator begin();  
    iterator end();  
    const_iterator cbegin() const;  
    const_iterator cend() const;  
    /*...*/  
};
```

Key Idea: Sometimes **less** functionality is **better** functionality

- Technically, adding a const-interface only **limits** what `RealVector` objects marked `const` can do
- Using types to enforce assumptions we make about function calls help us prevent programmer errors!

Questions?

Today



- ~~Recap: Const correctness~~
- Type Safety
- The need for “sometimes-a-thing”
 - `std::optional`

Definition

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

Recall: Python 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
```


Definition

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.

What does this code do?

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

What happens when input is {} ?

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

What does this code do?

```
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 (also the status quo)

```
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

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?

Definition

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

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 first solution?

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

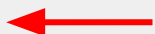
`back()` now advertises that there may or may not be a last element

Problems with using `std::pair<bool, valueType>`

```
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 using `std::pair<bool, valueType>`

```
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 using `std::pair<bool, valueType>`

```
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?

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


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  
}
```

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 if `back()` returned an optional?

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


How would it look to use `back()` ?

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

This would not compile!

How would it look to use `back()` ?

```
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!

(some of) `std::optional` **interface**

- `.value()`

returns the contained value or throws `bad_optional_access`
error

`std::optional` **interface**

- `.value()`

returns the contained value or throws `bad_optional_access` error

- `.value_or(valueType val)`

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

`std::optional` **interface**

- `.value()`

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

How would it look to use `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

How would it look to use `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 :/

How would it look to use `back()` ?

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

Better?

How would it look to use `back()` ?

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

Totally hacky, but totally works ;)

How would it look to use `back()` ?

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

Totally hacky, but totally works ;) don't do this ;)

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

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

Implementation of vector [] operator

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

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);  
}
```

The underlying memory implications actually get very complicated...

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?

`std::optional` “monadic” interface (C++23 sneak peek!)

- `.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`

`std::optional` “monadic” interface (C++23 sneak peek!)

- `.and_then(f)`
returns the result of `f` if the optional has a value, otherwise `optional()`
- `.transform(f)`
returns the result of `f` applied to the value of the optional, otherwise `optional()`
- `.or_else(f)`
returns the value of the optional if it has a value, otherwise the result of `f`

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.

`std::optional` “monadic” interface (C++23 sneak peek!)

- `.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`

How would it look to use `back()` ?

```
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();  
    }  
}
```

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

Recall: Design Philosophy of C++

- Only add features if they solve an actual problem
- Programmers should be free to choose their own 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 ~~optionals~~ monads

- Rust 🥰🥰

Systems language that guarantees memory and thread safety (take 110L!)

- Swift

Apple's language, made especially for app development

- JavaScript

Everyone's favorite

Type safety still matters in C++!

A sneaky example of type safety...

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

More bad code

```
void removeFirstA(string& str) {  
    int index = str.find('a');  
    //do something with index  
}
```

- What if there is no 'a' in str?
- No reason str.find shouldn't return an optional (IMO)

Classes with an emphasis on safety

- CS110L - Safety in Systems Programming
 - Companion course to ~~110~~ 111, whenever you take it!
 - Systems...but in Rust
- CS242 - Programming Languages
 - Take at least 107 first!
 - Learn a lot of languages
 - Emphasis on Rust

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!
- The ball is in your court!
- Besides using them in classes, you can use them in application code where it makes sense! This is highly encouraged :)

Example: Use `std::optional` in application code!

CODE DEMO!

“Well typed programs cannot go wrong.”

- Robert Milner (very important and good CS dude)

Thanks for coming!

...

Tuesday is the last lecture: Haven will talk about a grab bag of topics! Enjoy your long weekend!