Type Safety and `std::optional`

...

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

😷 masks required

Attendance

bit.ly/3EFILhS

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
- 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?

Definition

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

Recall: Python vs C++

Python

```python
def div_3(x):
    return x / 3
div_3("hello")
```

//CRASH during runtime, can't divide a string

C++

```cpp
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`)

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

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!

`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

Checking if an optional has value...

```
std::optional<Student> lookupStudent(string name){//something}

std::optional<Student> output = lookupStudent("Keith");

if(output.has_value()){
    cout << output.value().name << " is from " <<
                                    output.value().state << endl;
} else {
    cout << "No student found" << endl;
}
```

Evaluate optionals for a value like bools!

```
std::optional<Student> lookupStudent(string name){//something}

std::optional<Student> output = lookupStudent("Keith");

if(output){
    cout << output.value().name << " is from " <<
                                   output.value().state << endl;
} else {
    cout << "No student found" << endl;
}
```

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

Code might look like this...

```
std::optional<Student> lookupStudent(string name){//something}

std::optional<Student> output = lookupStudent("Keith");

auto func = (std::optional<Student> stu)[] {

    return stu ? stu.value().name + "is from " +

                                    to_string(stu.value().state) : {};

}

cout << output.and_then(func).value_or("No student found");
```

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 :)

“Well typed programs cannot go wrong.”

- Robert Milner (very important and good CS dude)