finally optional variant

Rethink the way we code

Bio

Kristoffel Pirard has been into C++ application development since the 20th century, including hardware integration, data processing and Qt app development.

He is a programming enthusiastic with a strong interest in the power of functional programming techniques to achieve code correctness.

He’s currently employed with Sioux Embedded Systems, where he started giving trainings in C++11/14.

Summary

Finally optional & variant!

The C++ type system has proven to be a great help in writing correct programs. C++17 adds two vocabulary types to improve code quality and performance.

This talk will show how `std::optional` brings us new idioms to treat errors, letting the compiler help us to not mess up. We can have the power of checked error propagation without the cost of exception handlers.

Also, we'll shed our light onto `std::variant`, which adds an elegant way to deal with polymorphism. We will learn a surprising way to let the compiler check correctness of our state machines.

Overview

- Variant (15’)
  - Use case
  - Api
  - Example
- Optional (30’)
  - Use case
  - Api
  - (Category theory: sum types, function composition)
  - Example (15’)
- Questions

Overview

- Variant (15’)
  - Use case
  - Api
  - Example
- Optional (30’)
  - Use case
  - Api
  - (Category theory: sum types, function composition)
  - Example (15’)
- Questions

Variant: use case

- Polymorphism, but no ‘is-a’ relationship
- Storage reuse
- Examples
  - User selects one of many commands
  - Json-like data structures (! need recursion)
  - Representing changing state (cf. later)

Variant

Remember `union`?

```
union U {
   int i;
   double d;
};
```

- Either
  - Don’t access `d` after `i` was written! (undefined behavior)
  - You’ll need a type tag…
    - And a switch
    - And lots of code reviews

```
struct DU {
   enum { Int, Double } type_tag;
   U u;
}
```

Variant

Enter C++17: `std::variant`

- type awareness
- all value-type goodies included
- Intuitive access...
  - Or is it...? Wait and see.

Variant: API

- Declaration: types known upfront
  - `std::variant<int, double, Foo> u;`
  - Hint: `using MyVariant = std::variant<int, double, Foo>;`
- Initialization
  - `u = Foo{"hi"};`
  - `u = 5L;` => error: no match for ‘operator=’
  - `MyVariant v(std::in_place_type<Foo>, "hi");`
  - Default: first declared type, if default constructor
- Getting the current type
  - `if(u.index() == 0){ stuff_with(std::get<0>(u)); }`
  - `if(auto *p = std::get_if<int>(&u)) { stuff_with(*p); }`
  - This does not make me happy.

Variant: API - Visitation

- The `get` scaffolding looks… clumsy
- You can easily forget to handle a type (if the variant is extended with another type)
- => compiler, check my code, please....

Solved by `std::visit( visitor, u )`

  - Where `visitor` implements `operator()(T t)` for each type.
  - Still verbose…
  - But look! A nice trick with variadic templates on cppreference.com!

Variant: API - Visitation

- Like `boost::variant`: `static_visitor` struct
  - One `operator(T)` per type

```
struct V {
     void operator() (int){ std::cout << "int\n"; };
     void operator() (double){ std::cout << "double\n"; };
     void operator() (Foo){ std::cout << "Foo\n"; };
 } visitor;

 std::visit(visitor, u);
```

Variant: API - visitation

```
std::visit(overloaded {
   [](auto arg) { std::cout << arg <<  ' '; },
   [](double arg) { std::cout << std::fixed << arg <<  ' '; },
   [](const std::string& arg) { std::cout <<  std::quoted(arg) << ' '; },
}, v);
```

- ! need user defined template deduction guides (C++17)
- Compiler warns you when forgetting a case!

Variant: API variation

```
In file included from /home/xtofl/monad_experiments/monadic-clutter/variantstuff.cpp:9:0:
/usr/include/c++/7/variant: In instantiation of 'static constexpr decltype(auto) std::__detail::__variant::__gen_vtable_impl<std::__detail::__variant::_Multi_array<_Result_type (*)(_Visitor, _Variants ...)>, std::tuple<_Rest ...>, std::integer_sequence<long
unsigned int, __indices ...> >::__visit_invoke(_Visitor&&, _Variants ...) [with _Result_type = std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>; _Visitor =
overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>, runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >&&; _Variants =
{std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&}; long unsigned int ...__indices = {2}]':
/usr/include/c++/7/variant:666:23:   required from 'static constexpr auto std::__detail::__variant::__gen_vtable_impl<std::__detail::__variant::_Multi_array<_Result_type (*)(_Visitor, _Variants ...)>, std::tuple<_Rest ...>, std::integer_sequence<long
unsigned int, __indices ...> >::_S_apply() [with _Result_type = std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>; _Visitor =
overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>, runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >&&; _Variants =
{std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&}; long unsigned int ...__indices = {2}]'
/usr/include/c++/7/variant:563:61:   required from 'static constexpr void std::__detail::__variant::__gen_vtable_impl<std::__detail::__variant::_Multi_array<_Result_type (*)(_Visitor, _Variants ...)>, __dimensions ...>, std::tuple<_Variants ...>,
std::integer_sequence<long unsigned int, __indices ...> >::_S_apply_single_alt(_Tp&) [with long unsigned int __index = 2; _Tp = std::__detail::__variant::_Multi_array<std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval,
runFSM()::FSM::Ready> (*)(overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>, runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >&&,
std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&)>; _Result_type = std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>; _Visitor =
overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>, runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >&&; long unsigned int ...__dimensions = {3}; _Variants =
{std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&}; long unsigned int ...__indices = {}]'
/usr/include/c++/7/variant:651:39:   required from 'constexpr const std::__detail::__variant::_Multi_array<std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>
(*)(overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>, runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >&&, std::variant<runFSM()::FSM::WaitForOpening,
runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&), 3> std::__detail::__variant::__gen_vtable<std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>,
overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>, runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >&&, std::variant<runFSM()::FSM::WaitForOpening,
runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&>::_S_vtable'
/usr/include/c++/7/variant:704:29:   required from 'struct std::__detail::__variant::__gen_vtable<std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>,
overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>, runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >&&, std::variant<runFSM()::FSM::WaitForOpening,
runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&>'
/usr/include/c++/7/variant:1239:23:   required from 'constexpr decltype(auto) std::visit(_Visitor&&, _Variants&& ...) [with _Visitor = overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>,
runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >; _Variants = {std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&}]'
/home/xtofl/monad_experiments/monadic-clutter/variantstuff.cpp:71:21:   required from here
/usr/include/c++/7/variant:704:49:   in constexpr expansion of 'std::__detail::__variant::__gen_vtable<std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>,
overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>, runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >&&, std::variant<runFSM()::FSM::WaitForOpening,
runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&>::_S_apply()'
/usr/include/c++/7/variant:701:38:   in constexpr expansion of 'std::__detail::__variant::__gen_vtable_impl<std::__detail::__variant::_Multi_array<std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>
(*)(overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>, runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >&&, std::variant<runFSM()::FSM::WaitForOpening,
runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&)>, std::tuple<std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&>, std::integer_sequence<long unsigned int> >::_S_apply()'
/usr/include/c++/7/variant:641:19:   in constexpr expansion of 'std::__detail::__variant::__gen_vtable_impl<std::__detail::__variant::_Multi_array<std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>
(*)(overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>, runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >&&, std::variant<runFSM()::FSM::WaitForOpening,
runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&), 3>, std::tuple<std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready>&>, std::integer_sequence<long unsigned int> >::_S_apply_all_alts<0, 1,
2>(\xe2\x80\x98result_dec\xe2\x80\x99 not supported by dump_expr#<expression error>, (std::make_index_sequence<3>(), std::make_index_sequence<3>()))'
/usr/include/c++/7/variant:679:17: error: no matching function for call to '__invoke(overloaded<runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::WaitForOpening)>,
runFSM()::FSM::process(std::__cxx11::string)::<lambda(runFSM()::FSM::OpenInterval)> >, std::variant_alternative_t<2, std::variant<runFSM()::FSM::WaitForOpening, runFSM()::FSM::OpenInterval, runFSM()::FSM::Ready> >&)'
  return __invoke(std::forward<_Visitor>(__visitor),
                 ~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                 std::get<__indices>(
                 ~~~~~~~~~~~~~~~~~~~~
                 std::forward<_Variants>(__vars))...);
                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
std::visit(overloaded{
    [](auto arg) { std::cout << arg << ' '; },
    [](double arg) { std::cout << std::fixed << arg << ' '; },
    [](const std::string& arg) { std::cout << std::quoted(arg) << ' '; },
}, v);
```

Compiler warns you when forgetting a case!

Variant: surprising use case

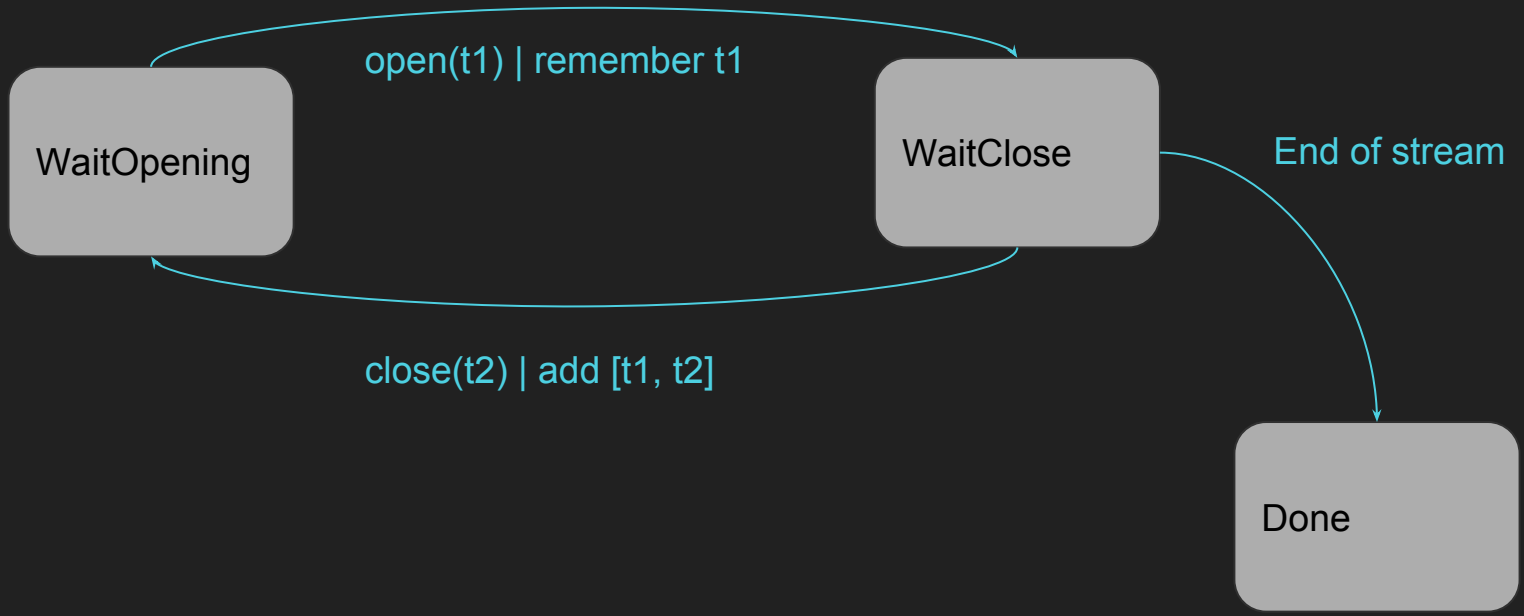
State machines…

E.g. listening socket: create, bind, listen, accept…

- In each state, different values are relevant
- Common solution: keep superset
  - Discipline needed to not use irrelevant values
- => variant<State1, State2, State3>
- http://khuttun.github.io/2017/02/04/implementing-state-machines-with-std-variant.html

Variant: FSM example

Process open/close stream of events:



Variant: FSM example

Process open/close stream of events:

```
struct FSM {
    // states
    struct WaitForOpening{ std::vector<Interval> intervals; };
    struct OpenInterval{ std::vector<Interval> intervals; TimeStamp started; };
    struct Ready{ std::vector<Interval> intervals; };
    State state = WaitForOpening{{}};
```

Variant: FSM example

```
state = std::visit(overloaded{
    [=](WaitForOpening w) -> State {
       if (token == "stop") { return Ready{w.intervals}; }
       if (token == "open") { return OpenInterval{w.intervals, now()}; }
          throw bad_event("unknown event");
    },...
```

Variant: FSM example

```
state = std::visit(overloaded{
    ...
    [=](OpenInterval o) -> State {
        if (token == "close") {
            o.intervals.push_back({o.started, now()});
            return WaitForOpening{o.intervals};
        }
        throw bad_event("unknown event");
    },
```

Variant: Conclusions

- When?
  - Types known upfront
  - possible applications unlimited
- Visitation…
  - Use struct
  - (C++17) overloaded + deduction guides
- Promising for FSMs

Where to go from here?

- `boost::variant` (adds recursion!)
- Ben Dean’s talk (https://youtu.be/ojZbFIQSdI8?t=18m49s)
- Sum types, Algebraic data types

Overview

- Variant (15’)
  - Use case
  - Api
  - Example
- Optional (30’)
  - Use case
  - Api
  - (Category theory: sum types, function composition)
  - Example (15’)
- Questions

Theoretical Approach

Algebra:

- “Sum” a + b + c
  - x + 0 = x
  - x + y = y + x
  - x + y + z = (x+y) + z = x + (y+z)
- “Product” a * b * c
  - x * 0 = 0
  - x * 1 = 1 * x = x
  - xy = yx
  - xyz = x(yz) = (xy)z

Type:

- “Sum type” union {a, b, c}
  - union{int x} + {} ~= union{int x}
  - union{int x; bool y} ~= {bool y, int x}
  - {int x, bool y, string z} ~= {{int x, bool y}, string z}
- “Product” tuple<a, b, c>
  - tuple<a, EMPTY> ~= EMPTY
  - tuple<a,SING> ~= tuple<SING, a> ~= tuple<SING>
  - tuple<a, b> ~= tuple<b, a>
  - tuple<a, b, c>
    ~= tuple<a, tuple<b, c>>
    ~= tuple<tuple<a, b>, c>

Optional: use case

P.1 express ideas directly in code

indicate the ***possibility*** that something isn't there.

- `optional<int> lines_in_file;` // file may not be present
- `optional<double> average(range<int>);` // range may be empty

...

The alternatives

- `Special element`
  - `nullptr (cf. fopen("does_not_exist.txt");)`
  - sum of empty list = 0 OK, but average?
  - `std::find(begin(xs), end(xs), 10);` // == end(xs) convention
  - **:(check not enforced**
- ‘is_valid’ field
  - `struct Result { int value; bool is_valid; }`
- Exceptions
  - check : enforced, or propagated
  - :(sometimes not desired (embedded)

Optional: API - Basic Usage

- Create:
  - `optional<Foo> u;`
  - `optional<Foo> u{std::in_place, 1, “forwards args to Foo ctor”};`
  - `auto u = make_optional<Foo>(1, “forwards args to Foo ctor”)`
- Query:
  - `arg.has_value()`
- Get value
  - `arg.value()` // throws bad_optional_access if `` `!arg.has_value()` ``!
- Test/get value:
  - `arg.value_or(my_default)`

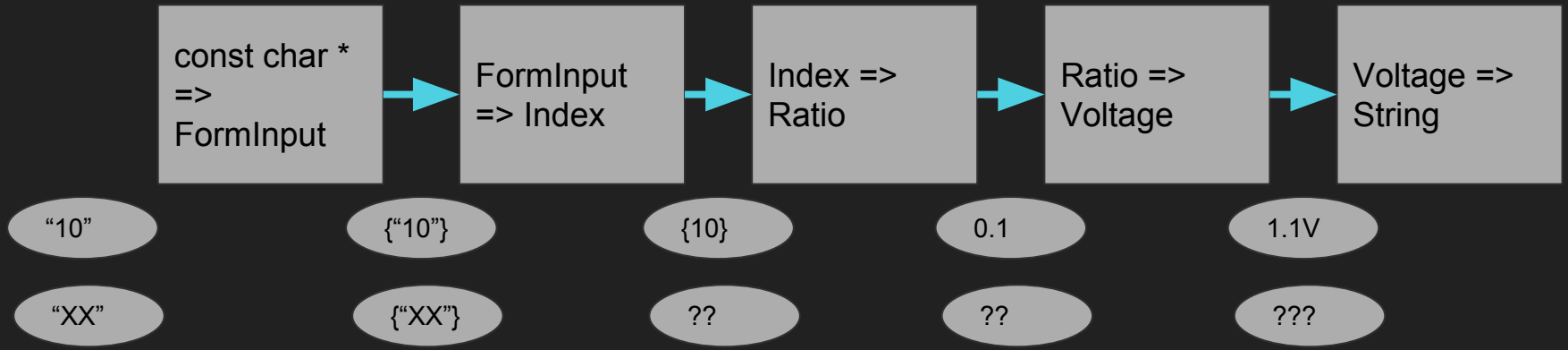
Optional: API

- Syntactic Sugar:
  - `if(arg) {...}`
  - `arg = another_foo` (note: not your regular assignment!)
- Pointer-like interface: defined if `arg.has_value()`, UB if not
  - `*arg`
  - `b = arg->bar()`

Optional: example

Simple conversion chain:

- Convert an ‘index’ (0-100) to a Voltage between 1.0 and 10.0 Volts



Optional: example

Design strategy

- Start with ‘happy flow’
- Add safety checks ‘exceptions’
- Rewind… add safety checks ‘optional’

Optional: example - semantic types

(aside: compiler disallows Voltage + Velocity)

```
struct FormInput { std::string_view value; };
struct Index { int value; };
struct Ratio { double value; };
struct Voltage { double value; };
struct VoltageRange { Voltage low; Voltage high; };
```

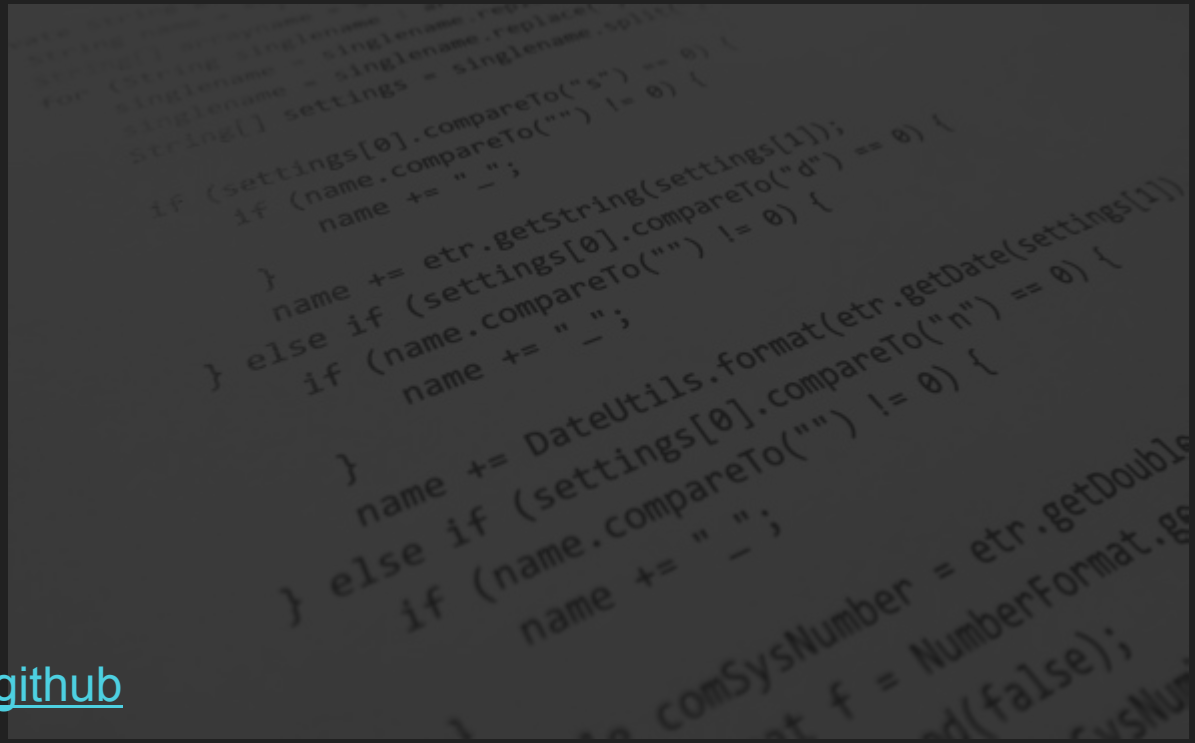
Optional: example - happy flow

1. arg[1] -> integer index (0-100)
2. Index -> fraction (0.0 -> 1.0)
3. Fraction -> voltage (1.0 -> 10.0)

Optional: example - error cases

1. arg[1] -> integer index (0-100) *(via FormInput)*
    a. There is no arg[1]
    b. arg[1] can’t be converted to int
2. Index -> fraction (0.0 -> 1.0)
    a. No Problem! Divide by 100!
3. Fraction -> voltage (1.0 -> 10.0)
    a. Voltage out of bounds (i.e. fraction out of [0, 1])

Coding Time!



[example on github]

Optional as a Functor/Monad

Note the pattern!

```
if (!arg) return std::string{"?"};
const auto input = FormInput{ *arg };
```

```
if (!arg) return NOTHING;
const auto result = optional(FUNCTION_OF(*arg));
```

Let’s extract this into ‘`transform`’

Optional as a Functor/Monad

```
if (!arg) return std::string{"?"};
const auto input = FormInput{ *arg };
```

```
const optional<FormInput> input =
      transform(arg, toFormInput);
```

```
template<typename X, typename Fxy>
auto transform(const std::optional<X> &opt, Fxy f)
-> std::optional<decltype(f(*opt))> {
   if (opt) { return {f(*opt)}; }
   else { return {}; }
}
```

Optional: code result

Compare with exceptions vs. with optional

```
auto toVoltageString(const std::optional<std::string_view> &arg)
{
   const auto input = transform(arg, [](auto x){ return FormInput{x}; });
   const auto index = flatten(transform(input, fromForm));
   const auto ratio = transform(index, fromIndex);
   const auto voltage = flatten(transform(ratio, toVoltage));
   return transform(voltage, voltageToString);
}
```

Optional: conclusions

- Explicitize failure/absence of data
- Enforce error handling
- Consistent error handling
  - So consistent it can be factored out!
- *transform / flatten* functions dealing with optional
  - Leave original functions alone
  - With a little help from Category Theory we can even factor out this composition!

Where to go from here?

- Take a look at std::expected, Boost.Outcome
- Haskell - the Maybe monad
- Sum types, Algebraic data types

Optional Composition: it can be simpler

We like ‘straight’ code

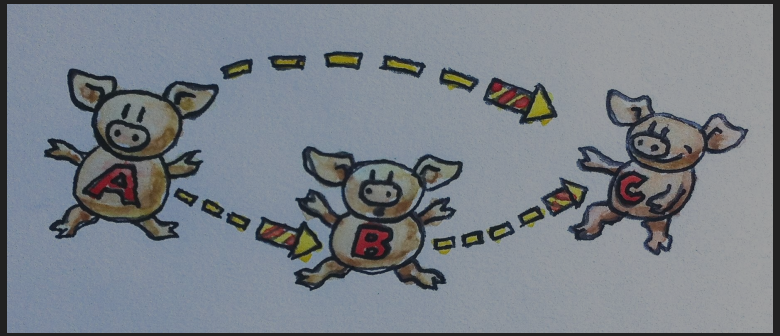
- accidental complexity should be hidden
  - So glad we didn’t use nesting!
- function composition creates the complexity!
  - `if (!x) return “?”;`


```
auto toVoltageString = compose(
    fromForm,
    fromIndex,
    toVoltage,
    voltageToString);
```

Optional Composition

Category Theory gives us some useful terminology...

- Terms:
  - Object
  - Arrow (morphism)
  - Composition
- Corresponds to
  - Type
  - Functions T1 -> T2
  - Function body: `b = f(a); c = g(b); ….`



Optional Composition

Category Theory gives us some useful terminology...

- Functor: something you can apply a function to
  - a.k.a. ‘vectorizing’, ‘mappable’, …
  - Examples:
    - `std::vector` + `boost::transform`: `transform(vec<in
    - `try{ return to_string(i) } catch (){throw;}`
- Monad: Functor… with composition
  - optional<int> + to_string => optional<string>
  - optional<string> + encode => optional<bytebuffer>
  - … optional<int> + {to_string, encode} => optional<bytebu

Questions?

```
std::variant<
   std::optional<
      std::vector<question>
   >,
   the_end
>
```

References

- Optional:
  - Fluent c++
  - https://hackernoon.com/error-handling-in-c-or-why-you-should-use-eithers-in-favor-of-exceptions-and-error-codes-f0640912eb45
  - https://blog.tartanllama.xyz/optional-expected/
  - https://youtu.be/vkcxgagQ4bM
- Variant:
  - https://akrzemi1.wordpress.com/2016/02/27/another-polymorphism/
  - http://khuttun.github.io/2017/02/04/implementing-state-machines-with-std-variant.html