C++ now 2025

Five Issues with std::expected and How to Fix Them

Vitaly Fanaskov

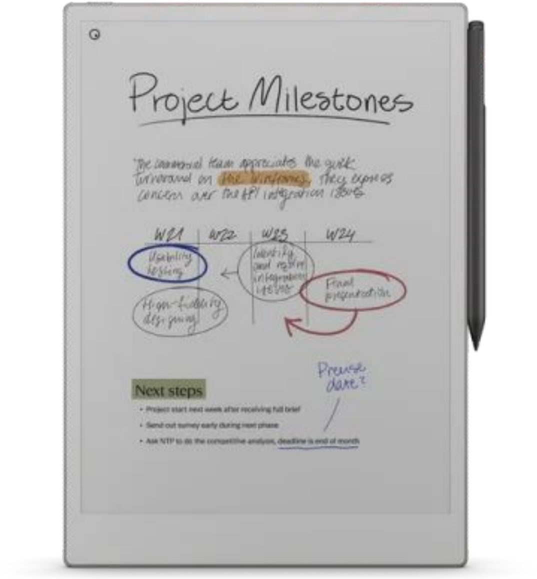
Five Issues with std::expected and How to Fix Them

About me

Vitaly Fanaskov

Principal software engineer at reMarkable

Work on C++ frameworks and libraries



Agenda

- Briefly about std::expected
- Some implementation details
- Monadic and regular interfaces
- Customization points
- bind_back and bind_front
- More monadic operations
- Conclusions

In this talk

- Almost no theory
- Simplified C++ code snippets
- Many change/improve suggestions

Examples and experimental implementation

https://github.com/vt4a2h/fl

expected.hpp

Briefly about std::expected

Definition

- Represents either of two values: *expected (T)* or *unexpected (E)*
- Union under the hood
- Has monadic operations

Basic example of using std::expected

```cpp
#include <expected>



std::expected<int, Error> expectedBox;
expectedBox = 42;

fmt::println("The value is: {}", expectedBox.value());
```

std::expected as a return value

When an operation can fail and we need to know why:

```
std::expected<Widget, WidgetError> loadWidget()
{
    // If error
    return std::unexpected(WidgetError{ /* ... */ });

    // Actual result
    return Widget{ /* ... */ };
}
```

Process std::expected

```
void loadWidget()
{
    if (const auto widgetBox = getNewWidget(); widgetBox.has_value()) {
        const auto widget = widgetBox.value();
        // Do something with the widget ...
    } else {
        const auto error = widgetBox.error();
        // Handle the error ...
    }
}
```

That was not an entirely bad example…

```
void loadWidget()
{
    const auto widgetBox = getNewWidget();

    if (widgetBox.has_value()) {
        // Do something with the widget ...
    } else {
        const auto error = widgetBox.error();
        log("Cannot get a new widget {}: {}.", widgetBox.value(), error);
    }
}
```

Monadic operations to the rescue!

```
getWidget()
    .and_then([](const auto &widget) -> std::expected<Widget, WidgetError> {
        // Do something with the widget ...
        return widget;
    })
    .transform([](const auto &widget) -> ID { return widget.id(); })
    .or_else([](const auto& error) -> std::expected<ID, WidgetError> {
        log(error);
        // Possibly recover and/or cleanup ...
        // Return value or error ...
    })
```

Available monadic operations

- `std::expected<V, E>::transform(F &&f)`
  - *`f: V -> T`*
- `std::expected<V, E>::transform_error(F &&f)`
  - *`f: E -> T`*
- `std::expected<V, E>::and_then(F &&f)`
  - *`f: V -> std::expected<T, E>`*
- `std::expected<V, E>::or_else(F &&f)`
  - *`f: E -> std::expected<V, T>`*

For more information

[Monadic Operations in Modern C++: A Practical Approach - Vitaly Fanaskov - CppCon 2024]

You will not be doing purely functional programming

- There will be side effects
- There will be object-oriented-style parts of the existing code base
- There will be integration with 3rd-party libraries
- …

A quick look under the hood

Types

- `template <class E> class unexpected`
- `template <class E> class bad_expected_access`
- `template <class T, class E> class expected`

Data structure

```
union {
    T val;
    E unex; // NOTE: not unexpected<E>!
};
bool has_val;
```

Issue #1:
Value type and error type can be the same

This is perfectly fine code

```cpp
using Expected = std::expected<std::string, std::string>;

Expected doSomething()
{
    return "Correct result"s;

    // Or

    return std::unexpected{"Something goes wrong"s};
};
```

How often do you need this?

Consequences of the existing implementation

- Extra type std::unexpected<E>
- API users must explicitly specify this type
- Complex implementations with many different restrictions
- Harder to use std::variant<T, E> under the hood

What is wrong with std::unexpected<E>?

- Additional abstraction
- Not quite symmetrical

Examples from other languages:

`enum Result<T, E> {`
`   Ok(T),`
`   Err(E),`
`}`
 	
`...`
 	
`Result<i32, i32>`
`Ok(42)`
`Err(42)`
	`sealed abstract class Either[+A, +B]`
	`extends Product with Serializable`
	
	`...`
	
	`Either[Int, Int]`
	`Left(42)`
	`Right(42)`

Let’s add some simple restrictions

```
!std::is_same_v<Value, Error>

!std::is_convertible_v<Value, Error>

!std::is_convertible_v<Error, Value>
```

Declare “new” expected

```
template <... Value, ... Error>
   requires (...)
struct expected : public std::variant<Value, Error>
{
    using base_t = std::variant<...>;

    using base_t::base_t;
};
```

- Use `monostate` when `Value` is void
- `Error` cannot be void

What do we get from this?

- Reuse the existing abstraction
- Simplify implementation (approx. 350 lines of code with some extra functionality, approx. 1k lines of tests)
- Use std::get and all variant-related utils
- Use std::visit (!)

A short story about std::visit

std::visit can be very convenient (regular use case)

```
using Shape = std::variant<Square, Rectangle, Circle>;

template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };


double calculateArea(const Shape &shape)
{
   return std::visit(overloaded{
       [](const Square& square) { return std::pow(square.side, 2); },
       [](const Rectangle& rect) { return rect.width * rect.height; },
       [](const Circle& circle) { return std::numbers::pi * std::pow(circle.radius, 2); },
   }, shape);
}

std::println("The shape area is: {}", calculateArea(shape));
```

…and can be even better

```
template <IsVariant Variant, class... Matchers>
   requires(sizeof...(Matchers) ≥ 1)
decltype(auto) match(Variant&& variant, Matchers&&... matchers)
{
   return std::visit(
      overloaded{std::forward<Matchers>(matchers)...},
         std::forward<Variant>(variant));
}
```

match can be more convenient

```
double calculateArea(const Shape &shape)
{
   return match(shape,
      [](const Square& square) { return std::pow(square.side, 2); },
      [](const Rectangle& rect) { return rect.width * rect.height; },
      [](const Circle& circle) { return std::numbers::pi * std::pow(circle.radius, 2); },
   );
}
```

Try it with expected

```
expected<Widget, WidgetError> widgetBox = getNewWidget();


match(widgetBox,
      [](const Widget& widget) { /* Handle value ... */ },
      [](const WidgetError& error) { /* Handle error ... */ }
);
```

We almost have pattern matching!

Section conclusions

Pros

- Simpler implementation
- Drop redundant abstractions
- Use standard utilities for std::variant
- Almost pattern matching now

Cons

- Type restrictions
- Potentially negative impact on performance

Issue #2:
Monadic and regular interfaces

API duplication

```
widgetBox.and_then(
   [](const Widget& widget) -> Expected {
      // Do something with a widget;
      return widget;
}).or_else(
   [](const WidgetError &error) -> Expected {
      // Handle error
      return std::unexpected{error};
});
```

```
if (widgetBox) {
   const auto &widget = widgetBox.value();
   // Do something with a widget;
} else {
   const auto &error = widgetBox.error();
   // Handle error
}
```

It seems that non-monadic interface is simpler and less verbose, right?

Benefits of monadic operations

- Safety
- Easy to propagate value or error
- Can be used in most of cases

Example: monadic operations in unit tests

```
std::ignore = calculateArea(shape)
    .and_then([expectedArea](double actualArea) -> Expected {
        REQUIRE(actualArea == expectedArea);
        return actualArea;
    })
    .or_else([](const auto &error) -> Expected {
        FAIL(fmt::to_string(error));
        return std::unexpected{error};
    });
```

Places where you still need the regular interface

- main()
- Libraries boundary
- “Integration” boundary

Example: “integration” boundary (QML interface)

```
class IWindowLayout
{
public:
    virtual core::Expected<> add(const window::Uri& windowUri) = 0;
};
```

```
class WindowLayout : public QObject
{
    Q_OBJECT
    // ...
public:
    Q_INVOKABLE bool WindowLayout::add(const window::Uri& windowUri)
private:
    std::shared_ptr<IWindowLayout> m_windowLayout;
};
```

Example: “Integration” boundary (expose into QML)

```cpp
bool WindowLayout::add(const window::Uri& windowUri)
{
    const auto result = m_windowLayout->add(windowUri).or_else(&printError);
    return result.has_value();
    // Or return a value of another type, such as QString
}
```

Section conclusions

- Monadic API is safer than regular API
- Monadic API can be used in most common use cases
- You may still need regular API
- Implementation via std::variant can gracefully hide dangerous API

Issue #3:
No customization points

We have a problem!

```
constexpr const T& value() const &
```

Throws: bad_expected_access(as_const(error())) if has_value() is false

```
constexpr const T* operator->() const // Ditto for operator*
```

Preconditions: has_value() is true.

If has_value() is false, the behavior is undefined.

```
constexpr const E& error() const & noexcept
```

Preconditions: has_value() is false.

If has_value() is true, the behavior is undefined.

Issues with value/error APIs

- Not symmetrical
- Throws
- Undefined behavior
- Cannot customize

Different error handling techniques

[Error handling in C++ - Vitaly Fanaskov - NDC Techtown 2022]

Provide a customization point

```
template<class Self>
  requires (...)
[[nodiscard]] constexpr auto&& value(this Self&& self) noexcept (...)
{
  if (self.has_error()) {
    if constexpr (CustomValueHandlerFound<Self>) {
      handle_bad_value(std::get<error_t>(std::forward<Self>(self)));
    } else {
      default_handle_bad_value<Self>();
    }
  }
  return std::get<value_t>(std::forward<Self>(self));
}
```

Default handler example

```
template <class>
constexpr void default_handle_bad_value()
{
    if consteval {
        throw "Expected object contains error";
    } else {
        std::terminate();
    }
}
```

Custom handler example

```
namespace gui
{

    void handle_bad_value(const WidgetError&) noexcept
    {
        // Handle, log, print stack trace...
        std::terminate();
    }

} // namespace gui
```

It’s not that easy

- Can handlers throw?
- Terminate by default?
- Singleton-like get/set or template specialization or else
- Function signatures are unclear

Contracts in C++26?

- For operator*() and operator->(), not for .value()
- With customization points
- Implementation details of contracts are still questionable
- Still with potential UB

Section conclusions

- Customization points for value()/error() can be super useful
  - No exceptions
  - No undefined behavior
  - Set different behavior for libraries or test code
- Many questions to API design and use cases
- Contracts could help
- We don’t need to think of these issues if we have std::variant + monadic API

Issue #4:
No built-in bind_back/bind_front

bind_back/front

```cpp
auto add = [](int a, int b) { return a + b; };
auto addOne = std::bind_back(add, 1);

std::println("{}", addOne(2)); // prints 3
```

```cpp
auto inc = [](int &a, int v) { a += v; };

int a{};
auto incA = std::bind_front(inc, std::ref(a));

incA(2);
std::println("{}", a);
```

Use cases with monadic operations

- Lambda functions can potentially reduce readability
  - Too long
  - Too many
  - Too nested
- There are already functions to use
  - More than single parameter
  - Used in several places
  - Not easy to change
- A function is a class method

Some issues when using with monadic operations

- May not be available
  - `std::bind_front` – C++20
  - `std::bind_back` – C++23
- Preserves arguments
- Arguments must be move constructible
- Implementation can use extra intermediate structures (e.g. tuple)

Some implementation details

```cpp
template <class _Fn, class... _Args>
  requires
    is_constructible_v<decay_t<_Fn>, _Fn> &&
    is_move_constructible_v<decay_t<_Fn>> &&
    (is_constructible_v<decay_t<_Args>, _Args> && ...) &&
    (is_move_constructible_v<decay_t<_Args>> && ...)
constexpr auto bind_front(_Fn&& __f, _Args&&... __args)
{
  [...bound_args = std::forward<_Args>(__args), f = std::forward<_Fn>(__f)]
    <class Self, class... T>(this Self&&, T&&... call_args)
  noexcept(/* ... */)
  -> /* ... */
  {
    return std::invoke(f, std::forward_like<Self>(bound_args)..., std::forward<T>(call_args)...);
  }
}
```

Make it built-in for monadic operations

```
template<class Self, class F, class ...Args>
   requires (/* ... */)
[[nodiscard]] constexpr auto and_then(this Self&& self, F &&f, Args &&...back_args)
   noexcept (/* ... */)
   -> /* ... */
{
   if (self.has_value()) {
       return std::invoke(
          std::forward<F>(f),
          std::get<value_t>(std::forward<Self>(self)),
          std::forward<Args>(back_args)...);
   } else {
       return std::get<error_t>(std::forward<Self>(self));
   }
}
```

Almost the same for bind_front

```
/* ... */
[[nodiscard]] constexpr auto and_then(
    this Self&& self, bind_front_t, F &&f, Args &&...front_args)
/* ... */
    return std::invoke(
        std::forward<F>(f),
        std::forward<Args>(front_args)...
        std::get<value_t>(std::forward<Self>(self)));
/* ... */
```

Pros of the approach

- Required no extra functions
- Don’t need to store arguments
- Requirements to arguments are relaxed
- Don’t need to use std::ref or std::cref
- Monadic operations keep the same signature (except for bind_front)

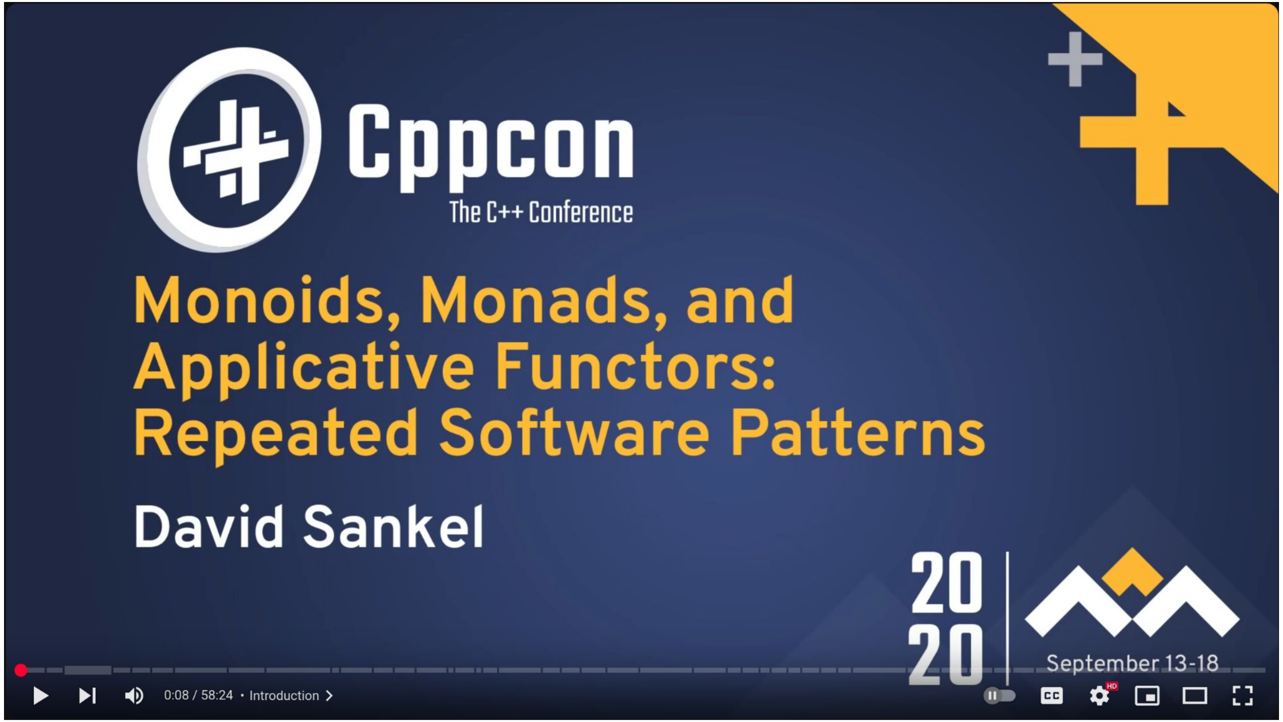
Section conclusions

- It's relatively easy to have built-in bind_back and bind_front
- Using built-in bind_back and bind_front has some practical benefits
- May not be fully idiomatically correct

Issue #5:
Not enough monadic operations!

General information

[Monoids, Monads, and Applicative Functors: Repeated Software Patterns - David Sankel - CppCon 2020]



Applicative-like may be the most useful

Applicative: The Forgotten Functional Pattern in C++ - Ben Deane - CppNow 2023

Let’s make it a little more practical

More canonical pattern quite far from regular use cases

```
template <class V, class E = Error>
using Expected = std::expected<V, E>;

template <class T, class U>
Expected<U> apply(Expected<std::function<U(T)>> f, Expected<T> v)
{
   if (f && v) {
       return std::invoke(*f, *v);
   } else {
       return std::unexpected{!f ? f.error() : v.error()};
   }
}

template <class T>
auto pure(T t) { return Expected<T>{t}; }
```

Let's say we use it as

```cpp
const std::function<std::string(int)> toString =
    [](int v) { return std::to_string(v); };

const Expected<int> number = getNumber();

const Expected<std::string> resultStr = apply(pure(toString), number);

// Do something with the result
```

However, most likely...

- A function is not wrapped into expected/optional/etc.
- A function parameter types are not expected/optional/etc.
- Some arguments will be wrapped into expected/optional/etc.

More real example

```
const auto toString =
    [](int v) { return std::to_string(v); };

const auto number = getNumber();

const Expected<std::string> resultStr = apply(
    toString, // ← just a function
    number    // ← can be Expected<int> or int
);
```

For expected type

- We can have an `ap`-like method (super close to `and_then`)
- Function to call is just a function
- Some arguments can be wrapped into expected with the same error type

Method interface

```
template <class Self, class F, class ...Args>
   requires (ApInvocable<F, error_t, value_t, Args...>)
[[nodiscard]] constexpr auto ap(this Self&& self, F &&f, Args &&...args)
   noexcept(
      std::is_nothrow_invocable_v<F, value_t, UnwrapOrForward<Args>...>
   )
   -> ApInvocableResult<F, error_t, value_t, Args...>
{ /* ... */ };
```

Argument restrictions

```
template <class Result, class Error>
concept NotExpectedOrSameError =
   !is_expected<std::decay_t<Result>> ||
   SameError<Error, std::decay_t<Result>>;

template <class Error, class ...Args>
concept ValidApArgs = (... && NotExpectedOrSameError<Args, Error>);
```

Function restrictions

```
template <class F, class Error, class ...Args>
concept ApInvocable = requires {
   requires ValidApArgs<Error, Args...>;
   requires std::is_invocable_v<F, UnwrapOrForward<Args>...>;
   requires NotExpectedOrSameError<
      std::invoke_result_t<F, UnwrapOrForward<Args>...>,
      Error>;
};


// UnwrapOrForward is Arg or typename std::decay_t<Arg>::value_t
```

For the result type

```
template <class F, class Error, class ...Args>
using ApInvocableResult =
    typename WrapIntoExpectedOrForward<
        JustInvocableResult<F, Args...>, Error>::type;

// WrapIntoExpectedOrForward is expected<Arg, Error>
// or Arg (if already expected)
```

Handle arguments with errors

Handle errors in arguments

```
if (auto firstError = firstError<error_t>(self, args...)) {
    return firstError.value();
} else {
    /* ... */
}
```

Get first error implementation

```
template <class Error, class ...Args>
   requires (ValidApArgs<Error, Args...>)
constexpr std::optional<Error> firstError(const Args&...args)
{
   return (std::optional<Error>{} << ... << args);
}
```

Get first error implementation (operator <<)

```cpp
template <class E, class Arg>
constexpr auto operator <<(std::optional<E> optErr, Arg &&arg) noexcept -> std::optional<E>
{
   if (optErr) {
       return optErr;
   }

   if constexpr (is_expected<std::decay_t<Arg>>) {
      if (arg.has_error()) {
         return std::make_optional<E>(
            std::get<typename std::decay_t<Arg>::error_t>(std::forward<Arg>(arg)));
      }
   }

   return std::nullopt;
}
```

Merge errors

- Semigroup-like combine (associative binary operation)
- combine(x, combine(y, z)) = combine(combine(x, y), z)

Combine two errors example

```cpp
struct Error
{
   std::string data;
};

constexpr auto combine(Error e1, Error e2)
{
   return Error{
      .data = fmt::format("{};{}", std::move(e1.data), std::move(e2.data))
   };
}
```

Combine optional example

```
template <class T>
   requires (is_optional<std::decay_t<T>>)
constexpr std::decay_t<T> combine(T &&a1, T &&a2)
{
   if (a1 && a2) { return combine(*std::forward<T>(a1), *std::forward<T>(a2)); }

   if (a1) { return std::forward<T>(a1); }

   if (a2) { return std::forward<T>(a2); }

   return std::nullopt;
}
```

Combine errors example with operator <<

```cpp
template <class E, class Arg>
   requires (CanCombine<std::optional<std::decay_t<E>>> &&
             CanCombine<std::decay_t<E>>)
constexpr auto operator <<(std::optional<E> optErr, Arg &&arg) noexcept -> std::optional<E>
{
   if constexpr (is_expected<std::decay_t<Arg>>) {
      if (arg.has_error()) {
         return combine(std::move(optErr),
                        std::make_optional<E>(
                           std::get<typename std::decay_t<Arg>::error_t>(std::forward<Arg>(arg))));
      }
   }

   return optErr;
}
```

Continue with ap implementation

Invoke a function

```
if (auto firstError =  /* ... */) {
    /* ... */
} else {
    return std::invoke(
        std::forward<F>(f),
        std::get<value_t>(std::forward<Self>(self)),
        unwrapOrForward(std::forward<Args>(args))...
    );
}
```

Use case

```
Expected openWindow(
   const WindowNavigator& windowNavigator, const Uri& uri, const State& initialState);

// ...

const fl::expected<gui::Uri, gui::Error> expectedUri = gui::fromString(/* input */);
const gui::State initialState = gui::getInitialState();

const auto result = gui::getWindowNavigator()
   .ap(&gui::openWindow, expectedUri, initialState);

// Or

/* ... */ = ap(gui::getWindowNavigator(), &gui::openWindow, expectedUri, initialState);
```

Future improvements

- Can be a part of and_then (with a certain tag)
- Can combine errors if supported or return the first on if not

Section conclusions

- Built-in “applicative” is valuable when
    - Some arguments are wrapped into expected
    - There are relatively many arguments
- Not general functionality
- May not belong to the expected interface

Can we add useful functionality without changing expected class?

Many changes are possible

- Freestanding functions
- Operator |
- Don’t use standard interface

Widget example with |

```
getWidget()
    | and_then([](const auto &widget) -> std::expected<Widget, WidgetError> {
        // Do something with the widget ...
        return widget;
    })
    | transform([](const auto &widget) -> ID { return widget.id(); })
    | or_else([](const auto& error) -> std::expected<ID, WidgetError> {
        log(error);
        // Possibly recover and/or cleanup ...
        // Return value or error ...
    })
```

Create unwrap-like functionality for tests

- Used with expected
- Fails is an expected object contains an error
- Returns a value if expected object contains a value

Simplified implementation

```
namespace detail {

struct unwrap_t {
   template <IsExpected Expected>
   [[nodiscard]] friend constexpr decltype(auto) operator |(Expected &&e, detail::unwrap_t)
   {
       if (e.has_error()) {
           FAIL(fmt::format("Expected object contains an error {:?}.", e.error()));
       }

       return std::forward<Expected>(e).value();
   }
};

}

inline constexpr detail::unwrap_t unwrap;
```

Usage example of unwrap

```
TEST_CASE("...")
{
    const Shape shape = /* ... */;

    /* ... */

    const auto area = calculateArea(shape) | unwrap;

    REQUIRE(area == /* ... */);
}
```

Section conclusions

- All existing monadic operations can be implemented via operator |
- Many extra monadic operations can be implemented via operator |
- You don’t need to re-create std::expected from scratch in this case

Final thoughts

Conclusions

- Current interface of std::expected has some issues
- Some of the issues result in error-prone code
- The interface lacks some practically useful functionality
- Most likely you may need to create your own "expected"
- Creating a pipe-like interface for std::expected can be a good compromise

Thank you!