Monadic Operations in Modern C++: A Practical Approach

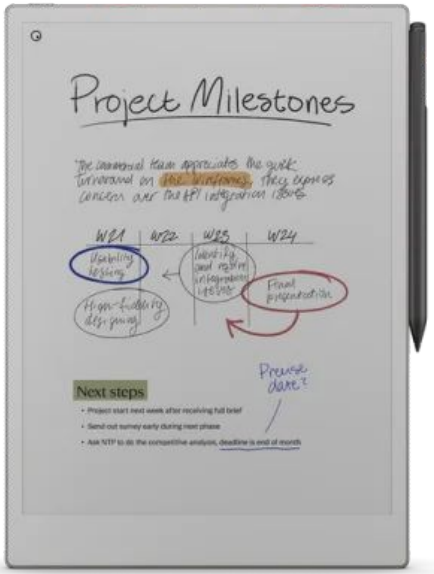
Monadic Operations in Modern C++:

A Practical Approach

VITALY FANASKOV

About me

- Vitaly Fanaskov
- Senior software engineer at reMarkable
- 10+ years of C++ experience
- GIS, VFX, frameworks, and libraries
- Ph.D (CS)



Agenda

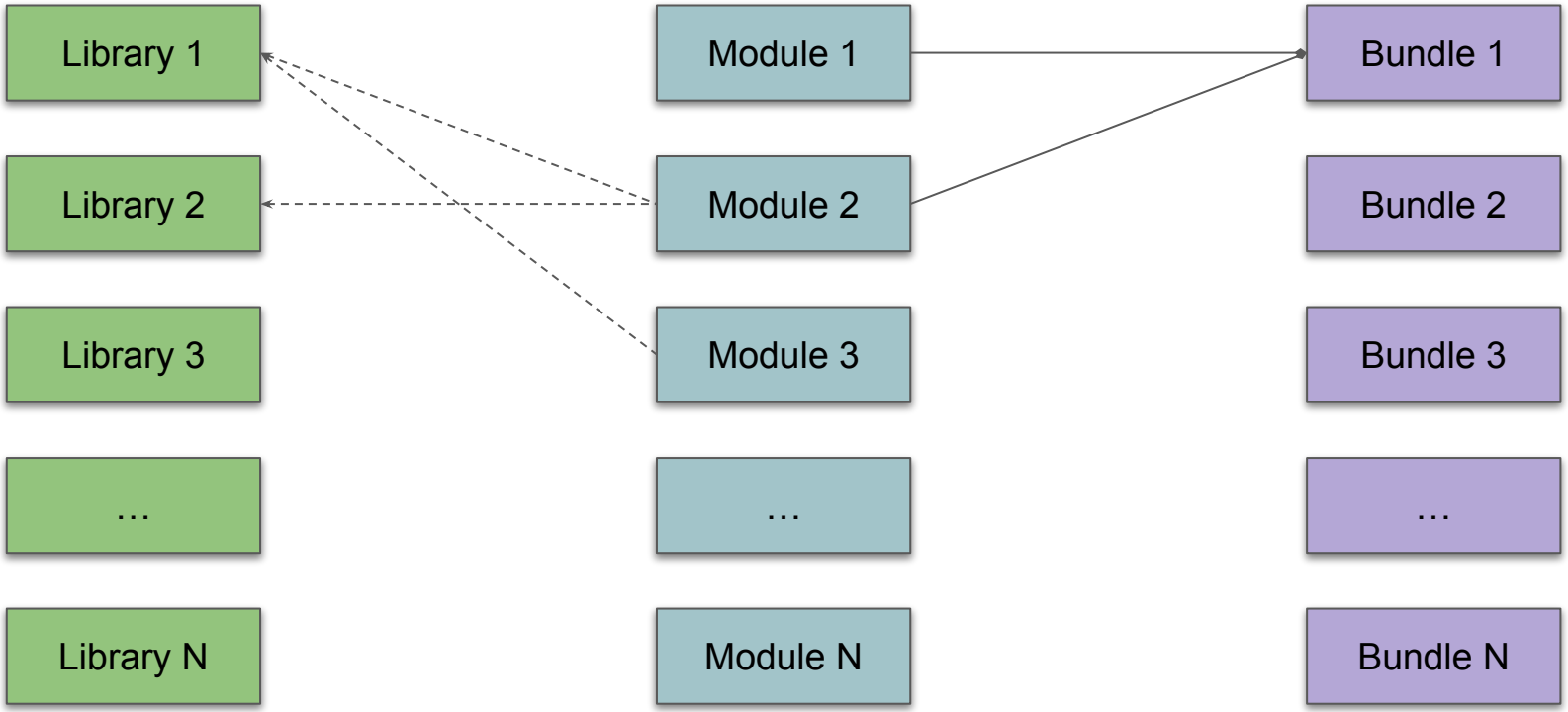
- Briefly about expected and optional
- Common use cases of expected
- Monadic operations in software development
- Tips and tricks

In this talk

- Less theory
- C++ only
- Practical examples

Where do examples come from?

Our internal framework



User interface subsystem

- Collect windows and widgets from modules
- Use bindings to 3rd party libraries (e.g. Qt) to display them
- Navigation

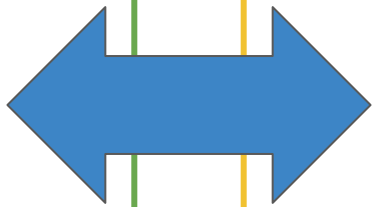
Technologies we use

- C++ 20
- vcpkg
- Many 3rd-party libraries (e.g. ranges-v3, tl-expected, catch2 etc)
- Qt for UI on devices

Briefly about Qt

C++ code:

- Business logic
- Integration
- System-level



QML code:

- UI-related things

Briefly about classes

std::optional

```
#include <optional>


std::optional<int> optionalBox;

optionalBox = 42;

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

- Contains a value
- …or doesn't contain a value
- Close to `std::pair<T, bool>`

std::optional as a return value

When an operation can fail, but it doesn't matter why:

```
template<class T>
[[nodiscard]]
std::optional<std::size_t> Vector<T>::indexOf(const T &element) const {
    // Do lookup...
    return std::nullopt; // If not found
}
```

std::optional as a parameter

When you need to pass some auxiliary arguments:

```
Url resolveUrl(
    std::string_view input,
    std::optional<Configuration> configuration = std::nullopt)
{
   // Read configuration if any
   // Resolve
}
```

It’ll be iterable

C++ 26

> The `optional` object is a `view` that contains either one element if it *contains a value*, or otherwise zero elements if it *does not contain a value*. The lifetime of the contained element is bound to the object.

std::expected

```
#include <expected>


std::expected<int, Error> expectedBox;

expectedBox = 42;

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

- Either a value
- …or an error
- Close to `std::variant<T, E>`

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{ /* ... */ };
}
```

Where can I get it?

- std::*
- tl::* (via vcpkg or Conan)

Use cases

What do we use?

- std::expected (approximately 90% of all cases)
- Error handling
- To unify interface

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 entirely bad example…

```
void loadWidgetV2()
{
   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);
   }
}
```

How do we handle this?

Monadic operations: and_then

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

```
getWidget().and_then(
    [](const auto &widget) -> std::expected<Widget, WidgetError> {
        // Do something with the widget ...
        return widget;
    });
```

Monadic operations: transform

```
if (const auto widgetBox = getWidget(); widgetBox.has_value()) {
   const auto widget = widgetBox.value();
   return widget.id();
}
```

```
getWidget().transform([](const auto &widget) → ID { return widget.id(); });
```

Monadic operations: or_else

```
if (const auto widgetBox = getWidget(); widgetBox.has_value()) {
    // ...
} else {
    log(widgetBox.error());
}
```

```
getWidget().and_then(/* ... */).or_else([](const auto& error){ log(error); });
```

Monadic operations: transfrom_error

```
if (const auto widgetBox = getWidget(); widgetBox.has_value()) {
    // ...
} else {
    return fmt::to_string(widgetBox.error());
}
```

```
getWidget().and_then(/* ... */).transform_error(&fmt::to_string<WidgetError>);
```

On the way to software development

Key parts

- People
- Design
- Start with small pieces

People

An example: widget ID – left or right?

```
std::expected<ID, WidgetError> loadWidget()
{
   if (const auto widgetBox = getWidget()) {
       const auto widget = widgetBox.value();

       // Do something with the widget ...

       return widget.id();
   } else {
       return {};
   }
}
```

```
std::expected<ID, WidgetError> loadWidget()
{
   return getWidget()
       /* .and_then ... */
       .transform(&Widget::id);
}
```

Observations

- Functional programming is difficult for many
- Combining functions using “monadic operations” is not straightforward
- std::optional and std::expected is terra incognita

What should I do about this?

- Show practical benefits (e.g. less boilerplate code)
- Do more workshops
- Do design review

Design

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
- …

Try splitting functional-style code from the rest

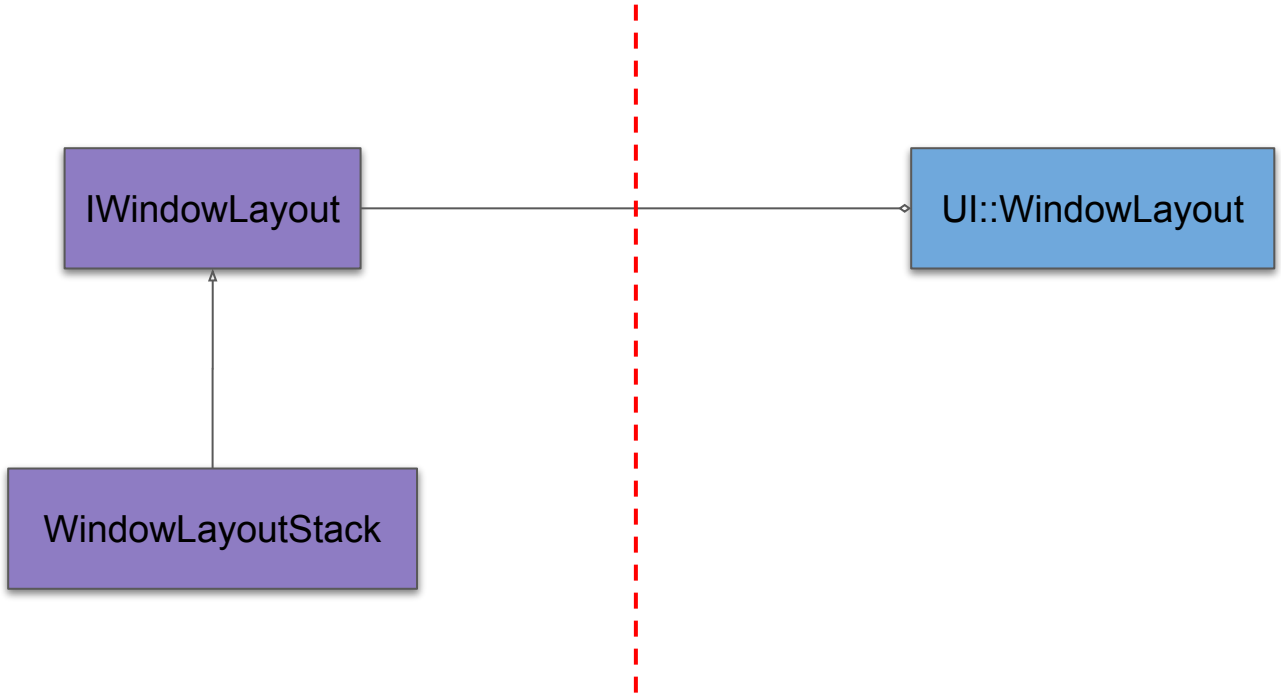
- A part where we use “monadics” and apply the best practices of functional programming

- A part where it can be difficult to do (or it doesn't make much sense)
  - Final logging output
  - Integration with UI libraries
  - Glue-code
  - ...

Where is the boundary?

- Class interface
- Library interface

Example: window layout class diagram



Example: window layout

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

// ...

core::Expected<> WindowLayoutStack::add(const window::Uri& windowUri)
{
    return loadWindow(windowUri, m_windowLoader)
        .and_then(&addToLayout)
        .and_then(&updateActiveWindow)
        .or_else(&addWindowLayoutPrefix);
}
```

Example: window layout wrapper

```
// Inside another namespace
class WindowLayout : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString activeWindow READ activeWindow NOTIFY activeWindowChanged)
    // ...
private:
    std::shared_ptr<IWindowLayout> m_windowLayout;
};
```

Example: window layout wrapper

```
void WindowLayout::add(const window::Uri& windowUri)
{
     m_windowLayout->add(windowUri).or_else(&printError);
}
```

What did we achieve?

- Integration with the existing codebase
- Monadic interface
- Separation of functional and non-functional code

Small steps

General approach (assume you have OO-style code-base)

- Start with the implementation of the methods
- Partially change class interface
- Fully change class interface
- (Optionally) Drop the entire class

Example: add a window – initial state

```
std::shared_ptr<Window> loadWindow(
    const window::Uri &windowUri, const window::Loader &loader);


// ...


bool WindowLayoutStack::add(const window::Uri& windowUri)
{
    if (auto window = loadWindow(windowUri, m_loader))
    {
        // ...
    }

    return false;
}
```

Example: add a window – change implementation

```
core::Expected<std::shared_ptr<Window>> loadWindow(
    const window::Uri &windowUri, const window::Loader &loader);


// ...


bool WindowLayoutStack::add(const window::Uri &windowUri)
{
    auto result = loadWindow(windowUri, m_loader)
        .and_then(&addToLayout)
        /* .and_then... */;

    // ...

    return result.has_value();
}
```

Example: add a window – change interface

```
core::Expected<> WindowLayoutStack::add(const window::Uri& windowUri)
{
    return loadWindow(windowUri, m_loader)
        .and_then(&addToLayout)
        /* .and_then... */;
}
```

Tips and Tricks

Assuming

- You’re at the very beginning of your journey
- There are not many well-defined practices for using monadic operations
- There are not too many functional programming libraries used in the project

Lambda functions

- Lambda functions are flexible and powerful tool
- With noisy syntax
- And this is yet another footgun

Use less:

- Nested lambda functions
- Long lambda functions
- Lambda functions assigned to local variables

Use less lambda functions

```
core::Expected<> WindowLayoutStack::add(
    const window::Uri& windowUri)
{
    return loadWindow(
        windowUri, m_windowLoader)
        .and_then(&addToLayout)
        .and_then(&updateActiveWindow)
        .or_else(&addWindowLayoutPrefix);
}
```

```
core::Expected<> WindowLayoutStack::add(
    const window::Uri& windowUri)
{
    const auto addToLayout =
        []( /**/ ) { /**/ };
    const auto updateActiveWindow =
        []( /**/ ) { /**/ };
    const auto addWindowLayoutPrefix =
        []( /**/ ) { /**/ };

    return loadWindow(
        windowUri, m_windowLoader)
        .and_then(addToLayout)
        .and_then(updateActiveWindow)
        .or_else(addWindowLayoutPrefix);
}
```

…and more free functions in general

- Small steps with names
- Easy to reuse
- Easy to test

Use bind_back/front

- There is already a function
- …used in many places
- …and you cannot easily change a signature
- A solution with lambda functions looks too cumbersome

- `std::bind_front` – C++20
- `std::bind_back` – C++23

bind_back/front

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

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

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

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

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

Possible implementation of bind_back

```
template<class F, class ...BoundArgs>
auto bind_back(F &&f, BoundArgs &&...boundArgs)
{
    return [...boundArgs = std::forward<BoundArgs>(boundArgs), f = f]
        <class ...RemainArgs>(RemainArgs &&...remainArgs) {
        return std::invoke(f, std::forward<RemainArgs>(remainArgs)..., boundArgs...);
   };
}
```

Example of using xostd::bind_back

```
core::Expected<> addWindowLayoutPrefix(
    const std::string &errorString, std::string_view customPrefix)
{
    return core::make_unexpected(fmt::format("{} {}", errorString, customPrefix));
};

//...

.or_else(std::bind_back(&addWindowLayoutPrefix, "[StackLayout]"));
```

Make all functions return expected/optional

- Easy to compose
- Doesn’t require additional libraries or helpers

```
core::Expected<> removeFromLayout(MapIt windowIterator, Map& windows, Stack& windowsStack)
{
    windowsStack.erase(windowIterator->second);
    windows.erase(windowIterator);

    return {};
}
```

Tuples your best friends

- No need to create extra structures
- Easy to pass several objects

```
core::Expected<std::tuple<Context, Component, QString>>
    createComponent(QQmlEngine& engine, const QString& fileName, const QString& viewName)
{
    // ...
    return std::make_tuple(std::move(context), std::move(component), viewName);
}
```

Don’t forget about transform_error

- 3rd-party libraries can have different error types
- Need to pass additional context
- Need to amend existing error (e.g. add a message prefix)

Explore 3rd-party libraries

- Get inspired
- Learn
- Use well-tested solutions

Thank you!