More “monadic” operations for modern C++

Oslo C++ Users Group

About me

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

- Zivid – 3D cameras for industrial robots
- https://www.zivid.com/career-join-zivid

Agenda

- “Monadic” operations
- Motivation
- std::optional
- std::expected
- Logging with monads

“Monadic” operations

“Monadic” operations

- Wrap a value in the “monadic” type
- Compose together

“Monadic” wrapper

```
template<class T>
struct AndThen {
   template<class F>
   auto andThen(F &&f) const & { return std::invoke(f, _value); }

   template<class F>
   auto andThen(F &&f) && { return std::invoke(f, std::move(_value)); }

   T _value;
};

template<class T>
AndThen(T &&v) -> AndThen<std::remove_cvref_t<T>>;


template<class V>
auto makeComposable(V &&v) { return AndThen{std::forward<V>(v)}; }
```

The task

```
{1, 2, 3, ..., n} ->
    {“1”, “2”, “3”, ..., “n”} ->
        “1 - 2 - 3 - ... - n”
```

Display result

Example: “monadic” operations

```
const auto toStr = [](const std::vector<int> &v) {
   std::vector<std::string> r;
   r.reserve(v.size());

   std::transform(std::begin(v), std::end(v), std::back_inserter(r),
                  [](auto v) { return std::to_string(v); });

   return makeComposable(std::move(r));
};

const auto join = [] (const std::vector<std::string> &v)  {
   assert(!v.empty());
   return makeComposable(std::accumulate(std::next(std::begin(v)), std::end(v), *std::begin(v),
                         [](auto acc, auto v) { return std::move(acc) + " - " + v; }));
};


const auto print = [](const std::string &s) { std::cout << s << std::endl; };
```

Example: “monadic” operations

```
makeComposable(std::vector{1, 2, 3})
    .andThen(toStr)
    .andThen(join)
    .andThen(print);


const auto result =
    makeComposable(std::vector{1, 2, 3})
        .andThen(toStr)
        .andThen(join);

print(*result);
// prints: 1 - 2 - 3
```

Example: “monadic” operations – looks familiar?

```
#include <ranges> // or ranges-v3

namespace rv = ranges::view;

const auto r = rv::iota(1, 4)
    | rv::transform([](int v) { return std::to_string(v); })
    | rv::join(std::string(" - "))
    | ranges::to<std::string>();

print(r);

// operator | is used to wrap values
```

“Monadic” operations in C++ – definition

```
f: V -> Composable<T>
```

```
f: Composable<V> -> Composable<T>
```

Shall I give it a try?

Pros of using “monadic” operations

- Fine-grained functions
- Re-usability
- Reducing boilerplate code
- Clean control-flow
- No side-effects (hopefully)

Cons of using “monadic” operations

- Performance penalty (not always)
- Extra memory allocations (not always)

Optional value

std::optional

- Value / no value
- No dynamic allocations
- Convertible to bool
- Can be constexpr
- Supports some “monadic” operations

std::optional – possible implementation

```
template <class T>
struct Optional {
   // ...

   template<class F>
   auto and_then(F &&f) const &
   { return _has_value ? std::invoke(f, _value) : nullopt; }

   // ...

   T _value;
   bool _has_value;
};
```

std::optional – example

```
std::optional<double> divide(double numerator, double denominator) {
    return denominator != 0 ?
        std::make_optional(numerator / denominator) : std::nullopt;
}

// ...

if (auto v = divide(v1, v2); v) {
    fmt::print("{}\n", v);
} else {
    fmt::print("Cannot complete operation.");
}
```

std::optional – more “monadic” example

```
auto sqrt = [] (double v) {
   return v > 0 ? std::make_optional(std::pow(v, 0.5)) : std::nullopt;
};

auto pow3 = [](double v) { return std::make_optional(std::pow(v, 3)); };

auto to_value_str = [](double v) {
   return fmt::format("The value is: {}", v);
};

auto to_error_str = [] {
   return std::make_optional(std::string("Invalid operation"));
};
```

std::optional – more “monadic” example

```
auto result = divide(v1, v2)
    .and_then(sqrt)
    .and_then(pow3)
    .transform(to_value_str)
    .or_else(to_error_str);
fmt::print("{}\n", result);

// 4, 2 -- 2.8 ...
// -4, 2 -- Invalid operation
// 4, 0 -- Invalid operation
```

std::optional – “monadic” API review

- `std::optional<T>::and_then(F &&f)`
  - *`f: T -> std::optional<V>`*
- `std::optional<T>::transform(F &&f)`
  - *`f: T -> V`*
- `std::optional<T>::or_else(F &&f)`
  - *`f: T -> std::optional<V>`*

When should I use std::optional?

- Chain computation (without exceptions)
- Clearly identify the absence of a value
  - `std::optional<std::string> value; // Questionable`
  - `std::optional<std::unique_ptr<SomeStruct>> value; // Probably not a good idea`
  - `return expr ? std::make_optional(value) : std::nullopt; // Looks good`
- The result is either presented or not

Result/Expected/Either

try-catch example

```
const auto divide = [](double n, double d) {
    if (d == 0)
        throw std::invalid_argument(
            fmt::format("[divide] Expected non-zero denominator. Got {}.", d));

    return n / d;
};

const auto sqrt = [] (double v) {
    if (v < 0)
        throw std::invalid_argument(
            fmt::format("[sqrt] Expected positive value. Got {}.", v));

    return std::pow(v, 0.5);
};

const auto pow3 = [](double v) { return std::pow(v, 3); };
```

try-catch example

```
try {
    auto result = divide(4, -2);
    result = sqrt(result);
    result = pow3(result);

    fmt::print("{}\n", result);
} catch (const std::invalid_argument &e) {
    fmt::print("Invalid operation: {}\n", e.what());
}

// prints "Invalid operation: [sqrt] Expected positive value. Got -2."
```

std::expected

- Contains either an expected value T or an error E
- No dynamic allocations
- Convertible to bool
- Supports some “monadic” operations (not in STL 😭)
- https://wg21.link/P0323R12 (accepted for C++ 23)
- Available implementations:
  - https://github.com/TartanLlama/expected
  - Latest gcc and MSVC

std::expected – naive implementation

```
template <class V, class E>
struct Result : public std::variant<V, E> {
   using std::variant<V, E>::variant;

   const V &operator *() const { return std::get<V>(*this); }
   const E &error() const { return std::get<E>(*this); }

   explicit operator bool () const { return std::holds_alternative<V>(*this);
}
};
```

tl::expected example

```cpp
#include <tl/expected.hpp>

using OpResult = tl::expected<double, std::string>;

const auto divide = [](double n, double d) -> OpResult {
   if (d == 0)
       return tl::make_unexpected(
           fmt::format("[divide] Expected non-zero denominator. Got {}.", d));

   return n / d;
};

const auto sqrt = [] (double v) -> OpResult {
   if (v < 0)
       return tl::make_unexpected(
           fmt::format("[sqrt] Expected positive value. Got {}.", v));

   return std::pow(v, 0.5);
};

const auto pow3 = [](double v) -> OpResult { return std::pow(v, 3); };
```

tl::expected example

```
const auto result = divide(4, -2)
    .and_then(sqrt)
    .and_then(pow3);

if (result) // result.has_value()
    fmt::print("{}\n", result.value());
else
    fmt::print("Invalid operation: {}\n", result.error());
// prints "Invalid operation: [sqrt] Expected positive value. Got -2."
```

tl::expected – “monadic” API review

- `tl::expected<V, E>::map(F &&f)`
  - *`f: V -> T`*
- `tl::expected<V, E>::map_error(F &&f)`
  - *`f: E -> T`*
- `tl::expected<V, E>::and_then(F &&f)`
  - *`f: V -> tl::expected<T, R>`*
- `tl::expected<V, E>::or_else(F &&f)`
  - *`f: E -> std::expected<T, R>`*

When should I use std::expected?

- As an alternative for exceptions
- Chain computation
- Pass additional error message (possible with context)

“Monadic” logging?

Factorial example

```
std::uint64_t factorial(std::uint64_t i) {
   return (i == 0 ? 1 : factorial(i - 1) * i);
}

// Add some logging

std::uint64_t factorial(std::uint64_t i) {
   if (i == 0) {
       spdlog::debug("Factorial of 0 is 1");
       return 1;
   } else {
       const auto ans = factorial(i - 1) * i;
       spdlog::debug("Factorial of {} is {}", i, ans);
       return ans;
   }
}
```

What’s wrong?

- Side effects
  - Hard to compose
  - Hard to test
- No referential transparency
  - Less predictable

Abstraction Writer – naive implementation

```
template<class LogType, class ValueType>
struct Writer {
   auto map(F &&f) const; // f: ValueType -> V
   auto flatMap(F &&f) const; // f: ValueType -> Writer<LogType, V>

   auto tell(const LogType &l) const;
   auto swap() const;
   auto reset() const

   LogType _log;
   ValueType _value;
};
```

Writer – types for examples

```
using Log = std::vector<std::string>;

using Value = int;

using Logger = fl::Writer<Log, Value>;
```

Writer – examples

```
auto l = Logger{{}, 1}.map([](int v) { return ++v; });
// l.value() == 2

auto l = Logger{{}, 1}.map([](int v) { return std::to_string(v); });
// l.value() == std::string("1")

auto l = Logger{{"foo"}, 1}.flatMap([](int v) { return Logger{{"bar"}, v + 1}; });
// l.log() == Log{"foo", "bar"}), l.value() == 2

auto l = Logger{}.tell({"foo"}).tell({"bar"}).tell({"baz", "baz"});
// l.log() == Log{"foo", "bar", "baz", "baz"}
```

How Writer combines logs? (simplified implementation)

```
template<class T>
struct Semigroup {
   T combine(const T& v1, const T& v2) const;
};

template<>
struct Semigroup<std::string> {
   std::string combine(const std::string& s1, const std::string& s2) const {
       return s1 + s2;
   }
};

auto tell(const LogType &l) const {
   return _details::make_writer(Semigroup<LogType>{}.combine(_log, l), _value);
}
```

Factorial with Writer – first iteration

```
[[nodiscard]]
Logger factorial(int i) {
    if (i == 0) {
        return Logger{{"Factorial of 0 is 1"}, 1};
    } else {
        auto [l, v] = factorial(i - 1);

        auto ans = v * i;
        l.emplace_back(fmt::format("Factorial of {} is {}", i, ans));

        return Logger{std::move(l), ans};
    }
}
```

Factorial with Writer – how to use

```
const auto [log, result] = factorial(5);

// result == 120
// log:
//    Factorial of 0 is 1
//    Factorial of 1 is 1
//    Factorial of 2 is 2
//    Factorial of 3 is 6
//    Factorial of 4 is 24
//    Factorial of 5 is 120
```

Factorial with Writer – better?

```
Logger factorial(Value i) {
    if (i == 0) {
        return Logger{{"Factorial of 0 is 1"}, 1};
    } else {
        auto r = factorial(i - 1).map([&](Value v) { return v * i; });
        return r.tell({fmt::format("Factorial of {} is {}", i, r.value())});
    }
}
```

Factorial with Writer – functional

```
auto logEntry(Value i, Value r) {
   return Log{fmt::format("Factorial of {} is {}", i, r), };
}

[[nodiscard]]
Logger factorial(Value i) {
   const auto mult = [&](auto v) { return v * i; };
   const auto addLogEntry = [&](auto a) { return Logger{logerEntry(i, a), a}; };

   return (i == 0 ? Logger{{}, 1} : factorial(i - 1).map(mult)).flatMap(addLogEntry);
}
```

Writer – “monadic” API review

- `fl::Writer<L, V>::map(F &&f)`
  - *`f: V -> T`*
- `fl::Writer<L, V>::flat_map(F &&f)`
  - *`f: V -> fl::Writer<L, T>`*
- `fl::Writer<L, V>::tell(const L &l)`
  - *`L -> fl::Writer<L, V>`*

When should I use Writer?

- Logging without side effects
- Non performance-critical systems

Should I use Writer?

```
using Writer = fl::Writer<Log, fl::Either<SomeResult, Error>>;
// some_writer.map(
//    either.map(
//        result.flatMap( ... )
//    )
// ).flatMap( ... )
```

- Can bloat the code base
- Many nested expressions
- Not for free

Thank you!