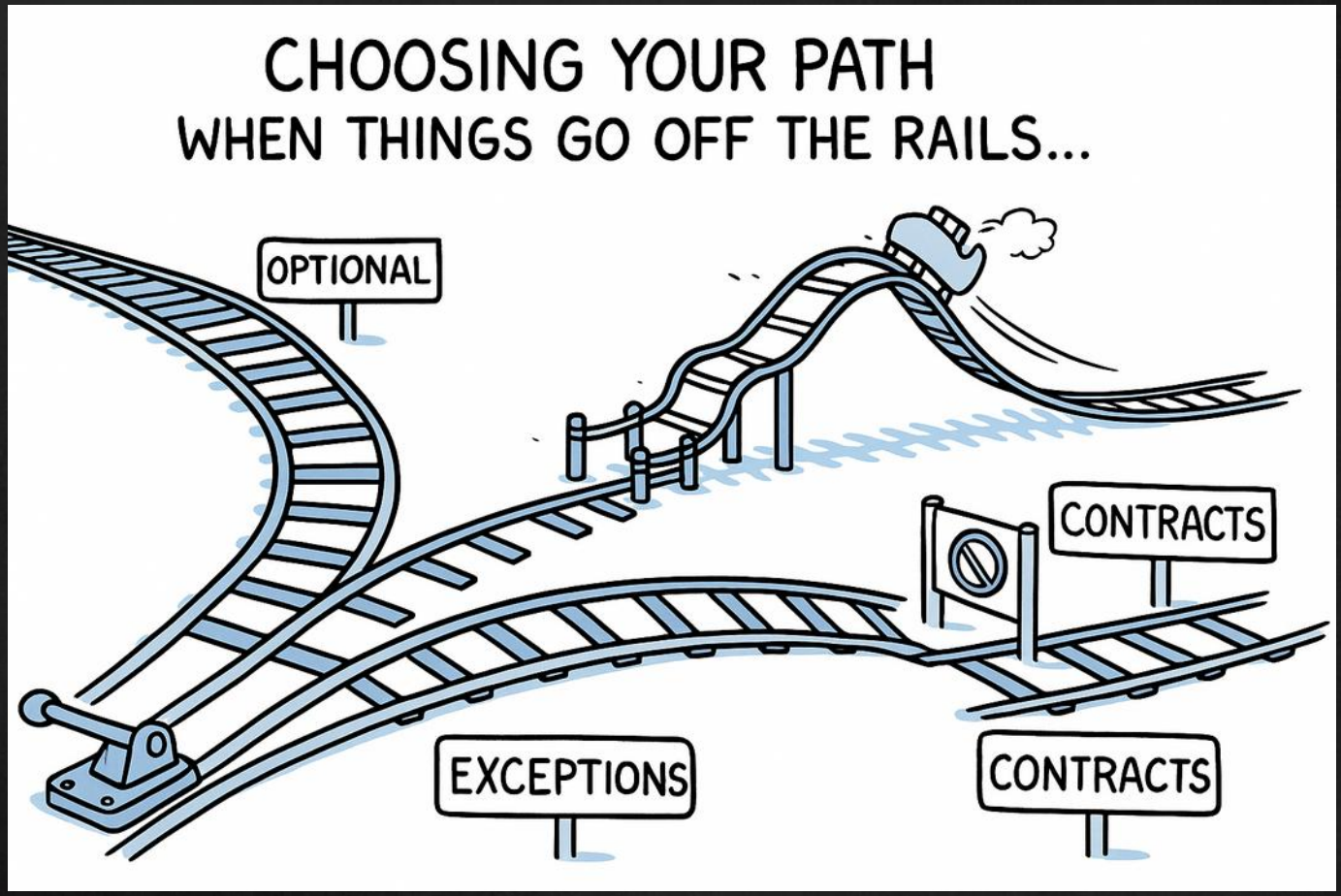
Modern C++ Error Handling

Failure Is Always an `std::option(al)` – or is it?

Ralph J. Steinhagen



Ralph J. Steinhagen, R.Steinhagen@GSI.de, 2025-06-14

Motivation

```
std::map<std::string, std::string> config = {{"port", "8080"}};
```

Goal: get int port = 8080; or report an error.

```
// C-Style
int getPort(const auto& map) {
    if (!map.contains("port")) return -1;
    int val = std::atoi(map.at("port").c_str());
    return (val > 0) ? val : -1;
}
```

```
// C++ exceptions
int getPort(const auto& map) {
    if (!map.contains("port")) throw std::runtime_error("missing port");
    return std::stoi(map.at("port"));
}
```

```
// return values, e.g. std::expected<T,E>
std::expected<int, Error> getPort(const auto& map) {
    if (!map.contains("port")) return std::unexpected(Error{"missing port"});
    return parseInt(map.at("port"));
}
```

Outline

- Historical Roots (C-style)
  - special return values
  - POSIX: `errno`, `strerror(int errnum)`
- C++ Exceptions
  - standard exceptions (try / catch)
- Value-Based Error Handling
  - since C++17: `std::optional<T>`
  - since C++23: `std::expected<T, E>`
  - since C++23: Monadic Extensions and Composition
- C++ Exceptions – Revisited
- Outlook Contracts TS (C++26 → backport → C++20)

Historical Roots, C-Style

pattern: `int func() { return -1; }`

```cpp
int parseInt(std::string_view s) {
    int val = 0;
    for (char c : s) {
        if (c < '0' || c > '9') { return -1; }
        val = val * 10 + (c - '0');
    }
    return val;
}

int main() {
    int val = parseInt("42");
    if (val == -1) { std::println("error message"); }
    else { std::println(val); }
}
```

Returns

- on success: `42`
- on failure: `error message`

Success implied by magic number (`'0'`, `<true|false>`, `NULL`, ...) -- failure by `-1`

Historical Roots, C-Style

alternate: `errno` & `strerror(int)` POSIX definition

```cpp
int parseInt(std::string_view s) {
    int val = 0;
    for (char c : s) {
        if (c < '0' || c > '9') {
            errno = EINVAL; // POSIX convention: only set on error
            return -1;
        }
        val = val * 10 + (c - '0');
    }
    return val;
}

int main() {
    int val = parseInt("42x");
    if (val == -1 || errno == EINVAL) { // POSIX first check magic number and only then errno
        std::println("MyError: {}", std::strerror(errno));
    } else {
        std::println("{}", val);
    }
}
```

Return on success: 42 || on failure: MyError: Invalid argument

Additional benefit: set of 78+ standardised error conditions -> reference

Historical Roots, C-Style

Why we need to care: C is part of C++

```cpp
int parseInt(std::string_view s) {
    int val = 0;
    for (char c : s) {
        if (c < '0' || c > '9') {
                return -1;
            }
        val = val * 10 + (c - '0');
    }
    return val;
}
```

```cpp
int parseInt(std::string_view s) {
    int val = 0;
    for (char c : s) {
        if (c < '0' || c > '9') {
            errno = EINVAL; // POSIX
            return -1;
        }
        val = val * 10 + (c - '0');
    }
    return val;
}
```

- Pros:
  - ✅ established fall-back (libraries, drivers, extern "C" -linkage, …),
  - ✅ cross-language & domain compatibility
- Cons:
  - ❌ error context is lost or implicit
  - ❌ easy to misuse (forget to check)
  - ❌ no type safety & limited ret-values → not composable nor expressive

Historical Roots, C-Style

composability issues

```
1  std::int32_t parseColour(std::string_view s) {
2      if (s.starts_with('{')) s.remove_prefix(1);
3      if (s.ends_with('}')) s.remove_suffix(1);
4
5      std::map<std::string_view, int> ch;
6      for (auto part : std::views::split(s, ',')) {
7          auto [key, val] = splitKeyValue(std::string_view{part}, ":");
8
9          auto parsed = parseInt(val);
10         if (parsed == -1) return -1;
11         ch[key] = parsed;
12     }
13
14     return (ch["red"] << 16) | (ch["green"] << 8) | ch["blue"];
15 }
16
17 int main() {
18     if (auto rgb = parseColour("{ red: 128, green: 2, blue: XX }"); rgb != -1)
19         std::println("RGB: 0x{:08X}", rgb);
20     else
21         std::println(stderr, "Error parsing colour — but no idea where or why");
22 }
```

https://compiler-explorer.com/z/r5xao96o7

Historical Roots, C++

exceptions

```
1  int parseInt(std::string_view s) {
2      int val = 0;
3      for (char c : s) {
4          if (c < '0' || c > '9') throw "invalid digit";
5          else val = val * 10 + (c - '0');
6      }
7      return val;
8  }
9
10 int getValue(const map_t& map, std::string key) {
11     if (!map.contains(key)) throw "key not found";
12     return parseInt(map.at(key));
13 }
14
15 int main() {
16     try {
18         std::cout << getValue(data, "answer") << '\n';
19     } catch (const char* err) {
20         std::cerr << "Exception: " << err << '\n';
21     }
22 }
```

on failure: `Exception: invalid digit`, or
`Exception: key not found`

Historical Roots, C++

exceptions contd.

```
1  int parseInt(std::string_view s) {
2      int val = 0;
3      for (char c : s) {
4          if (c < '0' || c > '9')
5              throw std::runtime_error(std::format("invalid digit '{}'", s));
6          val = val * 10 + (c - '0');
7      }
8      return val;
9  }
10
11 int getValue(const std::map<std::string, std::string>& map, std::string key) {
12     if (!map.contains(key))
13         throw std::runtime_error(std::format("key '{}' not found", key));
14     return parseInt(map.at(key));
15 }
16
17 int main() {
18     try {
19         std::println("{}", getValue({{"answer", "42"}}, "answer"));
20     } catch (const std::exception& err) { // catch from most specfic ...
21         std::println(stderr, "Exception: {}", err.what());
22     } catch (...) { // ... to catch-all
21         std::println(stderr, "Unknown Exception thrown");
22     }
23 }
```

on failure: Exception: invalid digit '42x', or
Exception: key 'AnotherAnswer' not found

Historical Roots

C++ exceptions

```
1  int parseInt(std::string_view s) {
2      int val = 0;
3      for (char c : s) {
4          if (c < '0' || c > '9') throw "invalid digit";
5          else val = val * 10 + (c - '0');
6      }
7      return val;
8  }
9
10 int getValue(const map_t& map, std::string key) {
11     if (!map.contains(key)) throw "key not found";
12     return parseInt(map.at(key));
13 }
14
15 int main() {
16     try {
18         std::cout << getValue(data, "answer") << '\n';
19     } catch (const char* err) {
20         std::cerr << "Exception: " << err << '\n';
21     }
22 }
```

on failure: `Exception: invalid digit`, or
`Exception: key not found`

Gretchen Frage: Which is the better API choice?

- Pros:
  - ✅ C++ core feature
  - ✅ separates error-handling from core logic
  - ✅ stack unwinding & RAII
  - ✅ clean call sites
  - ✅ automatic propagation
  - ✅ works well for unrecoverable or logic errors
- Cons:
  - ❌ performance overhead *(real or perceived)*
  - ❌ disabled by some (e.g. embedded, HPC)
  - implicit control flow: *harder to reason about for some developers?!?*

C++17: std::optional<T>

value-based logic for T

```cpp
#include <optional>

std::optional<int> parseInt(std::string_view s) {
    int val = 0;
    for (char c : s)
        if (c < '0' || c > '9') return std::nullopt;
        else val = val * 10 + (c - '0');
    return val;
}

std::optional<int> getValue(const std::map<std::string, std::string>& map, std::string key) {
    if (!map.contains(key)) return std::nullopt;
    return parseInt(map.at(key));
}

int main() {
    if (auto val = getValue({{"answer", "42"}}, "answer"); val) {
        std::println("{}", *val);
    } else {
        std::println(stderr, "no value");
    }
}
```

✅ no magic value any more

✅ value semantic

https://compiler-explorer.com/z/Eo9cWGch5

C++23: std::expected<T,E>

value-based logic for T and also for error E

```cpp
#include <expected>

std::expected<int, std::string> parseInt(std::string_view s) {
    int val = 0;
    for (char c : s)
        if (c < '0' || c > '9') return std::unexpected("invalid digit");
        else val = val * 10 + (c - '0');
    return val;
}

std::expected<int, std::string> getValue(const map_t& map, std::string key) {
    if (!map.contains(key)) return std::unexpected("key not found");
    return parseInt(map.at(key));
}

int main() {
    if (auto val = getValue({{"answer", "42"}}, "answer"); val) {
        std::println("{}", *val);
    } else {
        std::println(stderr, "{}", val.error());
    }
}
```

- ✅ no magic value any more
- ✅ value semantic
- ✅ expressive error message

https://compiler-explorer.com/z/jveT5TM7f

C++23: std::expected<T,E>

more advanced example I/II

```cpp
#include <expected>

std::expected<int, Error> parseInt(std::string_view s) {
    int val = 0;
    for (char c : s) {
        if (c < '0' || c > '9') {
            return std::unexpected(Error(std::format("invalid digit '{}' in \"{}\"", c, s)));
        } else {
            val = val * 10 + (c - '0');
        }
    }
    return val;
}

std::expected<int, Error> getValue(const std::map<std::string, std::string>& map, std::string key) {
    if (!map.contains(key)) return std::unexpected(Error(std::format("key '{}' not found", key)));
    if (auto val = parseInt(map.at(key)); val) {
        return val;
    } else {
        return std::unexpected(Error(std::format("issue with key '{}':\n    {:f}", key, val.error())));
    }
}

int main() {
    if (auto val = getValue({{"answer", "42"}}, "answer"); val) {
        std::println("{}", *val);
    } else {
        std::println(stderr, "{:T}", val.error());
    }
}
```

https://compiler-explorer.com/z/39Y5jjz5v

C++23: std::expected<T,E>

more advanced example I/II

```cpp
#include <expected>

std::expected<int, Error> parseInt(std::string_view s) {
    int val = 0;
    for (char c : s) {
        if (c < '0' || c > '9') {
            return std::unexpected(Error(std::format("invalid digit '{}' in \"{}\"", c, s)));
        } else {
            val = val * 10 + (c - '0');
        }
    }
    return val;
}

std::expected<int, Error> getValue(const std::map<std::string, std::string>& map, std::string key) {
    if (!map.contains(key)) return std::unexpected(Error(std::format("key '{}' not found", key)));
    if (auto val = parseInt(map.at(key)); val) {
        return val;
    } else {
        return std::unexpected(Error(std::format("issue with key '{}':\n    {:f}", key, val.error())));
    }
}
```

possible returns

on success: 42

on failure, for example:

```
2025-05-06 13:25:10.612: /app/example.cpp:103: key 'AnotherAnswer' not found
```

```
2025-05-06 13:02:47.653: /app/example.cpp:107: issue with key 'answer':
    /app/example.cpp:94 in std::expected<int, Error> parseInt(std::string_view): invalid digit 'x' in "42x"
```

https://compiler-explorer.com/z/39Y5jjz5v

C++23: std::expected<T,E>

Error helper class type

```cpp
struct Error {
    using Clock  = std::chrono::system_clock::time_point;
    using SrcLoc = std::source_location;

    std::string       message;
    SrcLoc            sourceLocation;
    Clock::time_point errorTime = Clock::now();

    Error(std::string_view msg = "unknown error",SrcLoc location = SrcLoc::current(),
          Clock::time_point time = Clock::now()) noexcept
          : message(msg), sourceLocation(location), errorTime(time) {}

    explicit Error(const std::exception& ex, SrcLoc location = SrcLoc::current())
                   noexcept : Error(ex.what(), location) {}

    explicit Error(const gr::exception& ex)
                   noexcept : Error(ex.message, ex.sourceLocation, ex.errorTime) {}

    [[nodiscard]] std::string srcLoc() const noexcept {
        return std::format("{}", sourceLocation);
    }

    [[nodiscard]] std::string methodName() const noexcept {
        return {sourceLocation.function_name()};
    }

    [[nodiscard]] std::string isoTime() const noexcept { // ms-precision ISO time-format
        return std::format("{:%F %T}", //
                    std::chrono::floor<std::chrono::milliseconds>(errorTime));
    }
};
```

https://compiler-explorer.com/z/39Y5jjz5v

C++23: Monadic Operations for
std::optional<T> & std::expected<T,E>

Method:	Signature:	Executes when:	Short-Circuits ?	Purpose:
transform(f)	**T → U**	has_value()	✅	map value to a new value (expected<U, E>)
transform_error(f) (N.B. std::exceptional only)	**E → F**	has_error()	✅	Map error to another new error (expected<T, F>)
and_then(f)	**T → expected<U, E>**	has_value()	✅	daisy-chain computation on value
or_else(f)	**E → expected<T, F>**	has_error()	✅	daisy-chain alternative computation/recovery on error
value_or(A x)	**() → T or A**	always	❌	value or provide fallback if error

C++23: Monadic Operations

short-circuiting example

```cpp
using Result = std::expected<std::string, std::string>;
Result A(bool ok) { return ok ? Result{"A"} : std::unexpected("errA"s); }
Result B(bool ok) { return ok ? Result{"B"} : std::unexpected("errB"s); }
Result C(bool ok) { return ok ? Result{"C"} : std::unexpected("errC"s); }

std::string runA(bool okA, bool okB, bool okC) {  // version short-circuits on unexpected error return
    auto combine = [](const std::string& a, const std::string& b) { return a + "|" + b; };

    return A(okA)
        .and_then([&](const std::string& a) {
            return B(okB)
                .and_then([&](const std::string& b) {
                    return C(okC)
                        .transform([&](const std::string& c) { return combine(combine(a, b), c); })
                        // short-circuit on error in C(...)
                        .or_else([&](const std::string& errC) { return Result{combine(a + "|" + b, errC)}; });
                })
                 // short-circuit on error in B(...)
                .or_else([&](const std::string& errB) { return Result{combine(a, errB)}; });
        })
        .or_else([](const std::string& errA) { return Result{errA}; }) // short-circuit on error in A(...)
        .transform([](const std::string& s) { return "[PASS] "s + s; })
        .or_else([](const std::string& e) { return Result{"[FAIL] "s + e}; })
        .value_or("[monadic-failure]");
}

int main() {
    std::println("\nVariant A - short circuit");
    std::println("{}", runA(true, true, true));     // [PASS] A|B|C
    std::println("{}", runA(false, true, true));    // [FAIL] errA
    std::println("{}", runA(true, false, true));    // [FAIL] A|errB
    std::println("{}", runA(true, true, false));    // [FAIL] A|B|errC
    std::println("{}", runA(false, false, false));  // [FAIL] errA
33  }
```

https://compiler-explorer.com/z/G5Prx7h9K

C++23: Monadic Operations

pass-through/recovery example

```cpp
using Result = std::expected<std::string, std::string>;
Result A(bool ok) { return ok ? Result{"A"} : std::unexpected("errA"s); }
Result B(bool ok) { return ok ? Result{"B"} : std::unexpected("errB"s); }
Result C(bool ok) { return ok ? Result{"C"} : std::unexpected("errC"s); }

std::string runB(bool okA, bool okB, bool okC) {
    bool success = true;

    return A(okA)
        .or_else([&](const std::string& errA) { success = false; return Result{errA}; }) // recover from error in A(...)
        .transform_error([](const std::string& errA) { return errA; })
        .and_then([&](const std::string& a) {
            return B(okB)
                .or_else([&](const std::string& errB) { success = false; return Result{errB}; }) // recover from error in B(...)
                .transform([&](const std::string& b) { return a + "|" + b; });
        }).and_then([&](const std::string& ab) {
            return C(okC)
                .or_else([&](const std::string& errC) { success = false; return Result{errC}; }) // recover from error in C(...)
                .transform([&](const std::string& c) { return ab + "|" + c; });
        })
        .transform([&](const std::string& result) {
            return (success ? "[PASS] " : "[FAIL] ") + result;
        })
        .or_else([&](const std::string& aggregatedError) {
            return Result{"[FAIL] " + aggregatedError};
        })
        .value_or("[monadic-failure]");
}

int main() {
    std::println("\nVariant B - pass through");
    std::println("{}", runB(true, true, true));     // [PASS] A|B|C
    std::println("{}", runB(false, true, true));    // [FAIL] errA|B|C
    std::println("{}", runB(true, false, true));    // [FAIL] A|errB|C
    std::println("{}", runB(true, true, false));    // [FAIL] A|B|errC
    std::println("{}", runB(false, false, false));  // [FAIL] errA|errB|errC
}
```

https://compiler-explorer.com/z/G5Prx7h9K

C++23: Monadic Operations for

composition revisited

```
147 std::expected<int, Error> parseComponent(
148     const auto& colourMap, std::string_view key,
149     std::source_location location = std::source_location::current()) {
150     return requireKey(colourMap, key, location).transform_error([&](const Error& e) {
151             return Error(std::format("missing or invalid key '{}' in {}:\n    {:f}",
152                 key, colourMap, e), location);
153         }).and_then([&](std::string_view value) {
154             return parseInt(value).transform_error([&](const Error& e) {
155                 return Error(std::format("invalid value for '{}':\n    {:f}",
156                     key, e), location);
157             });
158         });
159 }
160
161 std::expected<std::uint32_t, Error> parseColour(
162     std::string_view s,
163     std::source_location location = std::source_location::current()) {
164     if (s.starts_with('{')) s.remove_prefix(1);
165     if (s.ends_with('}')) s.remove_suffix(1);
166
167     std::map<std::string_view, std::string_view> colourMap;
168     for (auto part : std::views::split(s, ',')) {
169         auto kv = splitKeyValue(std::string_view(part),":", location);
170         if (!kv)
171             return std::unexpected(Error(std::format("split failed: {:f}",
172                                                      std::string_view(part), s,kv.error()),location));
173         auto [key, val] = *kv;
174         colourMap[key] = val;
175     }
176
177     return parseComponent(colourMap, "red", location).and_then([&](std::uint32_t r) {
178         return parseComponent(colourMap, "green", location).and_then([&](std::uint32_t g) {
179             return parseComponent(colourMap, "blue", location).transform([&](std::uint32_t b) {
180                 return (r << 16U) | (g << 8U) | b; });
181         });
182     });
183 }
```

https://compiler-explorer.com/z/jjcxdxcqK

C++23: Monadic Operations for

composition revisited

```
147 std::expected<int, Error> parseComponent(
148     const auto& colourMap, std::string_view key,
149     std::source_location location = std::source_location::current()) {
150     return requireKey(colourMap, key, location).transform_error([&](const Error& e) {
151             return Error(std::format("missing or invalid key '{}' in {}:\n    {:f}",
152                 key, colourMap, e), location);
153         }).and_then([&](std::string_view value) {
154             return parseInt(value).transform_error([&](const Error& e) {
155                 return Error(std::format("invalid value for '{}':\n    {:f}",
156                     key, e), location);
157             });
158         });
159 }
160
161 std::expected<std::uint32_t, Error> parseColour(
162     std::string_view s,
163     std::source_location location = std::source_location::current()) {
164     if (s.starts_with('{')) s.remove_prefix(1);
165     if (s.ends_with('}')) s.remove_suffix(1);
166
167     std::map<std::string_view, std::string_view> colourMap;
168     for (auto part : std::views::split(s, ',')) {
169         auto kv = splitKeyValue(std::string_view(part),":", location);
170         if (!kv)
171             return std::unexpected(Error(std::format("split failed: {:f}",
172                                                      std::string_view(part), s,kv.error()),location));
173         auto [key, val] = *kv;
174         colourMap[key] = val;
175     }
176
177     return parseComponent(colourMap, "red", location).and_then([&](std::uint32_t r) {
178         return parseComponent(colourMap, "green", location).and_then([&](std::uint32_t g) {
179             return parseComponent(colourMap, "blue", location).transform([&](std::uint32_t b) {
180                 return (r << 16U) | (g << 8U) | b; });
181         });
182     });
183 }
```

https://compiler-explorer.com/z/jjcxdxcqK

C++23: Monadic Operations for

composition revisited

```
185  int main() {
186      std::string input = "{ red: 128, green: 2, blue: 128 }";
187
188      if (auto rgb = parseColour(input); rgb) {
189          std::println("✅ RGB: 0x{:06X}", *rgb);
190      } else {
191          std::println(stderr, "❌ {}", rgb.error());
192      }
193  }
```

possible returns

on success:

✅ RGB: 0x800280

on failure, for example:

❌ /app/example.cpp:188: missing or invalid key 'blue' in {"green": "2", "orange": "xy", "red": "128"}:
/app/example.cpp:188 in int main(): missing colour component 'blue'

❌ /app/example.cpp:188: invalid value for 'blue':
/app/example.cpp:112 in std::expected<int, Error> parseInt(std::string_view): invalid digit 'x' in "xy"

❌ /app/example.cpp:188: split failed for ' blue; 128 ' in ' red: 128, green: 2, blue; 128 ':
/app/example.cpp:188 in int main(): missing value for key 'blue; 128'

https://compiler-explorer.com/z/jjcxdxcqKhttps://compiler-explorer.com/z/jjcxdxcqK

Error/Failure Handling

Which Pattern to use?

`using std::optional<T>/ std::exceptional<T,E>:`

- ✅ explicit control flow
- ✅ no runtime overhead from stack unwinding
- ✅ monadic chaining (and_then, transform, or_else...)
- ✅ clear API contracts
- ❌ more boilerplate without helper
- ❌ need for explicit eror propagation logic (may require thoughtful error structuring)
- ❌ not ideal for deeply nested call chains

`using exceptions throw, try { ... } catch (...) {}:`

- ✅ separation of logic and error handling
- ✅ clean "normal" control flow
- ✅ automatic stack unwinding (RAII)
- ✅ easy to retrofit without changing all interfaces
- ❓ potential performance hit (real or perceived)
- ❓ may be disabled in some environments (embedded/HPC)
- ❌ implicit control flow — harder to follow if overused
- ❌ difficult to know what exceptions a function might throw (no checked exceptions in C++, only `'noexcept(bool)'`)

Error/Failure Handling

Hybrid Approach

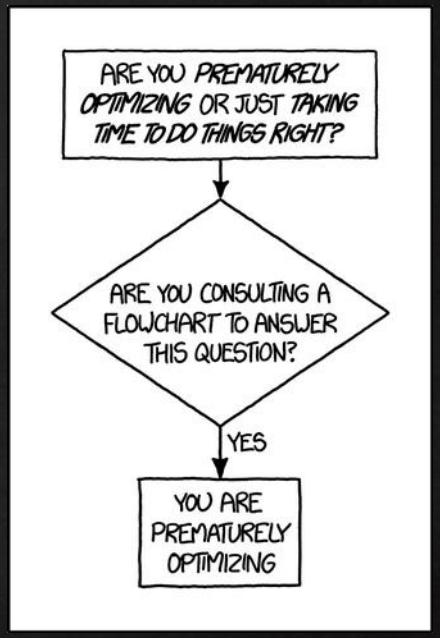
```
template<typename T, typename E>
T unwrapOrThrow(std::expected<T, E> exp) {
  if (!exp) {
     throw gr::exception(exp.error());
  }
    return *exp;
}
```

```
template<typename T, typename E, typename F>
std::expected<T, E> catchOrExpected(F&& f) {
  try {
    return std::invoke(std::forward<F>(f));
  } catch (const gr::exception& ex) {
    return std::unexpected(E{ex.error()});
  } catch (...) {
    return std::unexpected(E{"unknown e"});
  }
}
```

Rules of Thumb:

A) Exceptions:
"I don't know how to deal with this here."

B) Expected/Optional:
"I expect this might fail, and can act accordingly."



https://xkcd.com/1691/

Error/Failure Handling Hybrid contd.

gr::Error

```cpp
struct Error {
    using Clock  = std::chrono::system_clock::time_point;
    using SrcLoc = std::source_location;

    std::string       message;
    SrcLoc            sourceLocation;
    Clock::time_point errorTime = Clock::now();

    Error(std::string_view msg = "unknown error",SrcLoc location = SrcLoc::current(),
          Clock::time_point time = Clock::now()) noexcept
          : message(msg), sourceLocation(location), errorTime(time) {}

    explicit Error(const std::exception& ex, SrcLoc location = SrcLoc::current())
                   noexcept : Error(ex.what(), location) {}

    explicit Error(const gr::exception& ex)
                   noexcept : Error(ex.message, ex.sourceLocation, ex.errorTime) {}

    [[nodiscard]] std::string srcLoc() const noexcept {
        return std::format("{}", sourceLocation);
    }

    [[nodiscard]] std::string methodName() const noexcept {
        return {sourceLocation.function_name()};
    }

    [[nodiscard]] std::string isoTime() const noexcept { // ms-precision ISO time-format
        return std::format("{:%F %T}", //
                    std::chrono::floor<std::chrono::milliseconds>(errorTime));
    }
};
```

https://compiler-explorer.com/z/39Y5jjz5v

Error/Failure Handling Hybrid contd.

gr::exception

```
1  struct exception : public std::exception {
2      using Clock  = std::chrono::system_clock;
3      using SrcLoc = std::source_location;
4      std::string message;
5      SrcLoc sourceLocation;
6      Clock::time_point errorTime = Clock::now();
7
8      exception(std::string_view msg = "unknown exception",
9                SrcLoc location = SrcLoc::current()) noexcept
10         : message(msg), sourceLocation(location) {}
11
12     // construct from Error
13     explicit exception(const Error& err) noexcept : message(err.message),
14                        sourceLocation(err.sourceLocation), errorTime(err.errorTime) {}
15
16     [[nodiscard]] const char* what() const noexcept override {
17         if (formattedMessage.empty()) {
18             formattedMessage = std::format("{} at {}:{}",
19                         message, sourceLocation.file_name(), sourceLocation.line());
20         }
21         return formattedMessage.c_str();
22     }
23
24 private:
25     mutable std::string formattedMessage;
26 };
```

C++23: Monadic Operations for

composition revisited – for comparison

```
147 std::expected<int, Error> parseComponent(
148     const auto& colourMap, std::string_view key,
149     std::source_location location = std::source_location::current()) {
150     return requireKey(colourMap, key, location).transform_error([&](const Error& e) {
151             return Error(std::format("missing or invalid key '{}' in {}:\n    {:f}",
152                 key, colourMap, e), location);
153         }).and_then([&](std::string_view value) {
154             return parseInt(value).transform_error([&](const Error& e) {
155                 return Error(std::format("invalid value for '{}':\n    {:f}",
156                     key, e), location);
157             });
158         });
159 }
160
161 std::expected<std::uint32_t, Error> parseColour(
162     std::string_view s,
163     std::source_location location = std::source_location::current()) {
164     if (s.starts_with('{')) s.remove_prefix(1);
165     if (s.ends_with('}')) s.remove_suffix(1);
166
167     std::map<std::string_view, std::string_view> colourMap;
168     for (auto part : std::views::split(s, ',')) {
169         auto kv = splitKeyValue(std::string_view(part),":", location);
170         if (!kv)
171             return std::unexpected(Error(std::format("split failed: {:f}",
172                                                      std::string_view(part), s,kv.error()),location));
173         auto [key, val] = *kv;
174         colourMap[key] = val;
175     }
176
177     return parseComponent(colourMap, "red", location).and_then([&](std::uint32_t r) {
178         return parseComponent(colourMap, "green", location).and_then([&](std::uint32_t g) {
179             return parseComponent(colourMap, "blue", location).transform([&](std::uint32_t b) {
180                 return (r << 16U) | (g << 8U) | b; });
181         });
182     });
183 }
```

https://compiler-explorer.com/z/jjcxdxcqK

C++ Exceptions

std::source_location & composition revisited

```cpp
int parseComponent(
    const std::map<std::string_view, std::string_view>& colourMap, std::string_view key,
    std::source_location loc = std::source_location::current()) {
    try {
        std::string_view value = requireKey(colourMap, key, loc);
        return parseInt(value, loc);
    } catch (const gr::exception& e) {
        throw gr::exception(std::format("invalid value for '{}': {}", key, e.what()), loc);
    }
}

std::uint32_t parseColour(
    std::string_view s,
    std::source_location location = std::source_location::current()) {
    if (s.starts_with('{')) s.remove_prefix(1);
    if (s.ends_with('}')) s.remove_suffix(1);

    std::map<std::string_view, std::string_view> colourMap;
    for (auto part : std::views::split(s, ',')) {
        auto [key, val] = splitKeyValue(std::string_view(part), ":", location);
        colourMap[key] = val;
    }

    std::uint32_t r = parseComponent(colourMap, "red", location);
    std::uint32_t g = parseComponent(colourMap, "green", location);
    std::uint32_t b = parseComponent(colourMap, "blue", location);

    return (r << 16U) | (g << 8U) | b;
}
```

https://compiler-explorer.com/z/6s49jxofx

C++ Exceptions

`std::source_location` & composition revisited

```
106  int main() {
107      try {
108          std::uint32_t rgb = parseColour("{ red: 128, green: 2, blue; 128 }");
109          std::println("✅ RGB: 0x{:06X}", rgb);
110      } catch (const gr::exception& e) {
111          std::println(stderr, "❌ {}", e.what());
112      }
113  }
```

possible returns

on success:

✅ RGB: 0x800280

on failure, for example:

❌ invalid value for 'blue': missing colour component 'blue' at /app/example.cpp:108
in colourMap {"green": "2", "orange": "128", "red": "128"} at /app/example.cpp:108

❌ invalid value for 'blue': invalid digit 'x' in "xx" at /app/example.cpp:108
in colourMap {"blue": "xx", "green": "2", "red": "128"} at /app/example.cpp:108

❌ missing value for key 'blue; 128' at /app/example.cpp:108

https://compiler-explorer.com/z/6s49jxofx

C++ Exceptions

Performance & Binary Size

known:

- ✅ happy-path: no costs,
- ❌ exception: stack unwinding (but w/ RAII)

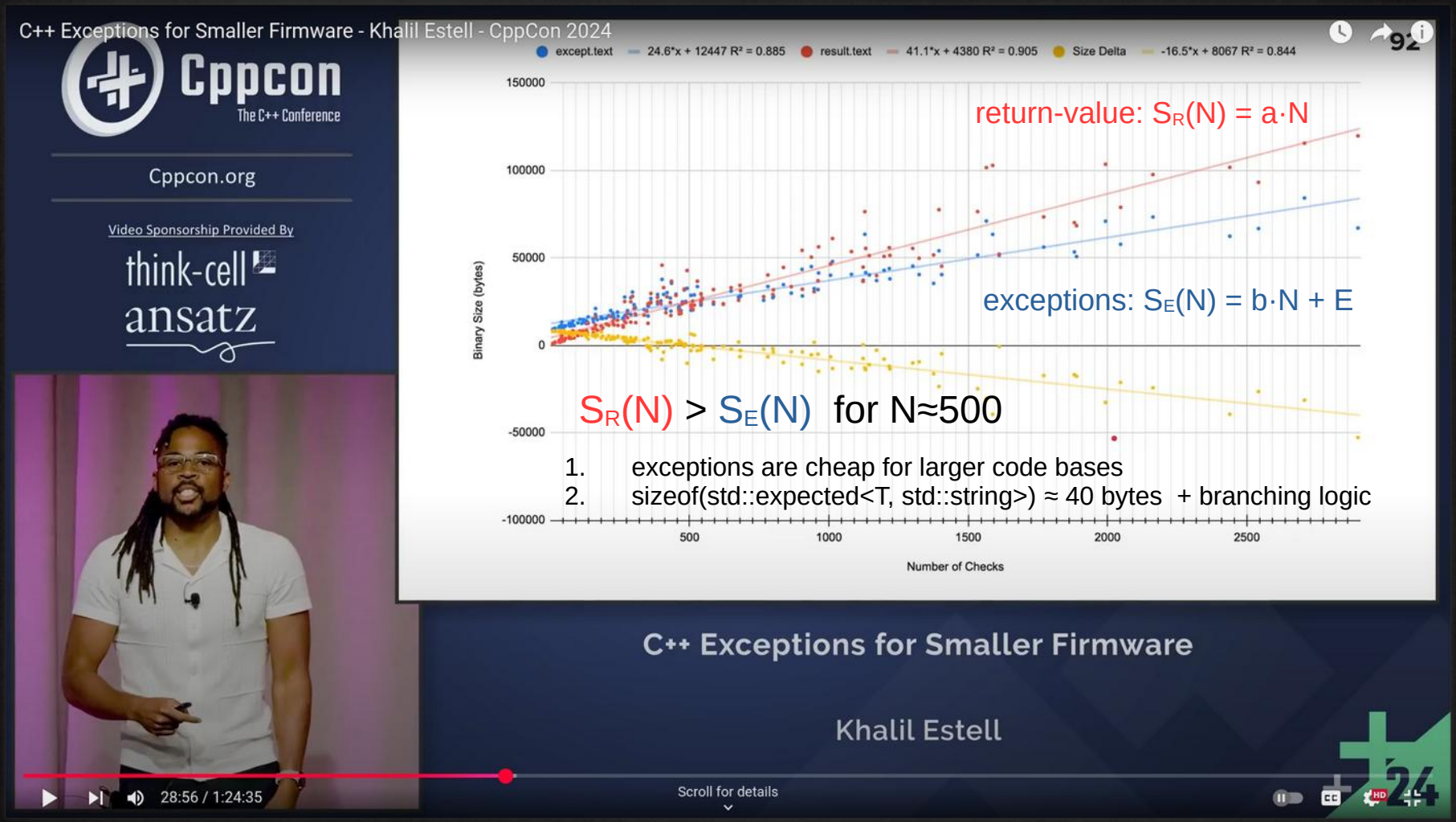
```cpp
[[gnu::noinline]] int start() {
  throw 42;
}

int main() {
  volatile int return_code = 0;
  try {
    return_code = start();
  } catch (...) {
    return_code = -1;
  }
  return return_code;
}
```

```asm
start():
  push    rax
  mov     edi, 4
  call    __cxa_allocate_exception
  xor     edx, edx
  mov     esi, OFFSET FLAT:typeinfo for int
  mov     DWORD PTR [rax], 42
  mov     rdi, rax
  call    __cxa_throw
main:
  sub     rsp, 24
  xor     eax, eax
  mov     DWORD PTR [rsp+12], eax
  call    start()
  mov     rdi, rax
  call    __cxa_begin_catch
  mov     DWORD PTR [rsp+12], -1
  call    __cxa_end_catch
  mov     eax, DWORD PTR [rsp+12]
  add     rsp, 24
  ret
```

https://compiler-explorer.com/z/GxM4zhxPh

C++ Exceptions

PMO? → Always Benchmark First!



"C++ Exceptions for Smaller Firmware", Khalil Estell, CppCon'24 (video, slides)

https://github.com/kammce/exception_research

C++26: Contracts

Remember the Mars Climate Orbiter incident from 1999?

C++26: Contracts

additional constraints

```
1 int clamp(int x, int lo, int hi)
2   pre (lo <= hi)
3   post (r: r >= lo && r <= hi) {
5  return std::max(lo, std::min(x, hi));
6 }
```

- part of C++26 (P2961)
- replace ad-hoc asserts with declarative syntax
- contracts ≠ error handling — defines API correctness
- "Use contracts to catch bugs early — not to recover from them."

- Good for:
  - Precondition checking
  - Invariants, e.g. loop or object state validation
  - Documenting and enforcing function interfaces
- Not for:
  - User input validation
  - Recoverable errors
  - Replacing exceptions/expected

C++26: Contracts

When and Why?

- Good for:
  - Precondition checking
  - Invariants, e.g. loop or object state validation
  - Documenting and enforcing function interfaces
- Not for:
  - User input validation
  - Recoverable errors
  - Replacing exceptions/expected

Mimicking C++26 contracts in C++20

```cpp
constexpr std::array<std::string_view, 12> monthNames = {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "" };
constexpr bool is_valid_month(int month) { return month >= 1 && month <= 12; }

std::string_view month_as_string(int month) {
    pre(is_valid_month(month));
    auto name = monthNames[month - 1];
    post(!name.empty());
    return name;
}

int add_positive(int a, int b) {
    pre(a >= 0 && b >= 0);
    int result = a + b;
    return post_return(result, result >= a && result >= b); // alt syntax
}

void demo(int month) {
   try { std::println("Month {}: {}", month, month_as_string(month)); }
   catch (const std::exception& ex) { std::println("caught contract violation: {}", ex.what()); }
}

int main() {
    violation_handler = &throwing_violation_handler;

    demo(5);
    demo(13);  // triggers pre-condition
    demo(12);  // triggers post-condition

  // [..]

    contract_assert(2 + 2 == 5); // triggers assertion
}
```

```
Program returned: 139
terminate called after throwing an instance of 'std::logic_error'
  what():  /app/example.cpp:32:5: assertion failed: (2 + 2 == 5): in function: int main()
Program terminated with signal: SIGSEGV
Month 5: May
caught contract violation: /app/example.cpp:6:5: precondition failed: (is_valid_month(month)): in function: std::string_view
month_as_string(int)
caught contract violation: /app/example.cpp:8:5: postcondition failed: (!name.empty()): in function: std::string_view month_as_string(int)
Sum = 7
caught contract violation: /app/example.cpp:13:5: precondition failed: (a >= 0 && b >= 0): in function: int add_positive(int, int)
```

https://compiler-explorer.com/z/v4h3K5vT1

Summary

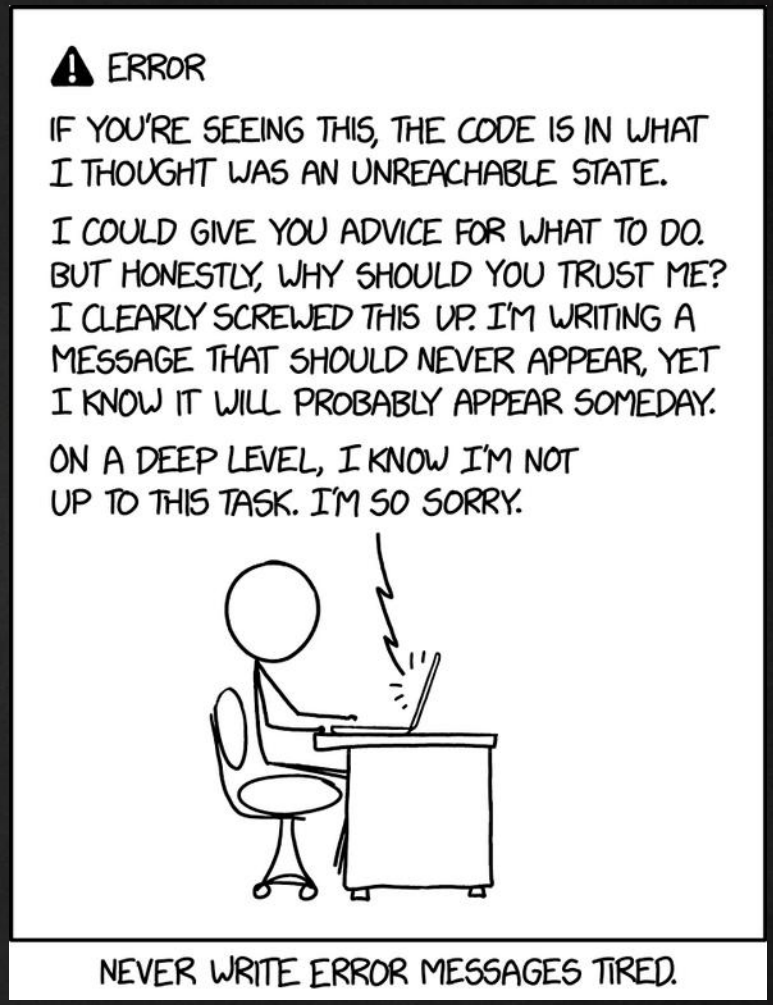
- No one-fits-all solution: kiss – measure – avoid PMO (e.g. avoiding exceptions)!
- Choosing the right error handling strategy
  - missing value but valid → `std::optional<T>`
  - expected recoverable error → `std::expected<T,E>`
  - unexpected or unrecoverable → `C++ exceptions (throw/catch)`
  - contract enforcement / invariants → `pre(), post(), ...`
- Choose lightest-weight tool that fits the semantics of the error:

  A) Exceptions:
  "I don't know how to deal with this here."

  B) Expected/Optional:
  "I expect this might fail, and can act accordingly."

Resources

- "Modern C++ Error Handling", Phil Nash, CppCon'24, (video, slides)
- "The Expected Outcome", Ivan Čukić, Meeting C++, 2022, (video, slides)
- "C++ Exceptions for Smaller Firmware", Khalil Estell, CppCon'24 (video, slides)
- "Contracts for C++", Timur Doumler, CppCon'24 (video, slides)



That's all – Questions?

Appendix