Modern C++ Error Handling

PHIL NASH

Cppcon
The C++ Conference

2024

September 15 - 20

Modern C++ Error Handling

Phil Nash

SHAVEDYAKS

CppCast


Option(al)
Is Not a
Failure
@phil_nash

2018

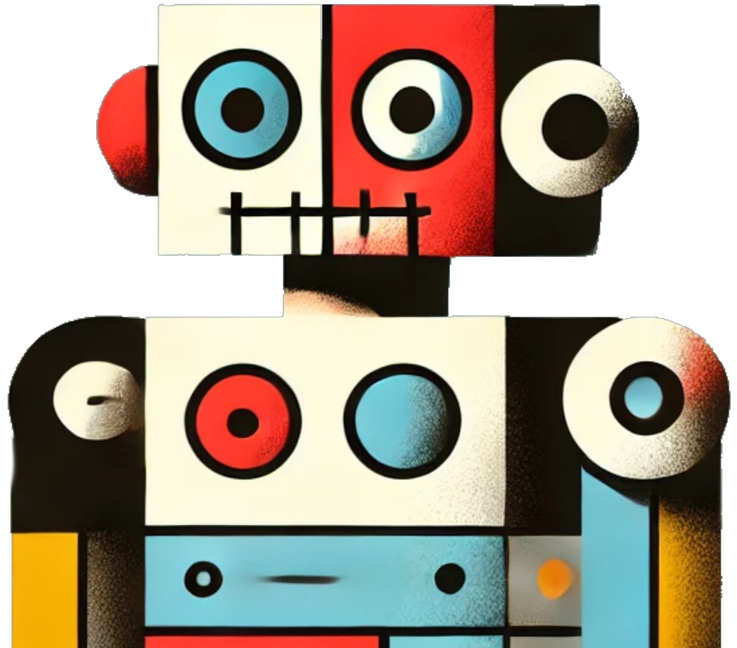

the
Dawn of a New Error
@phil_nash

2019

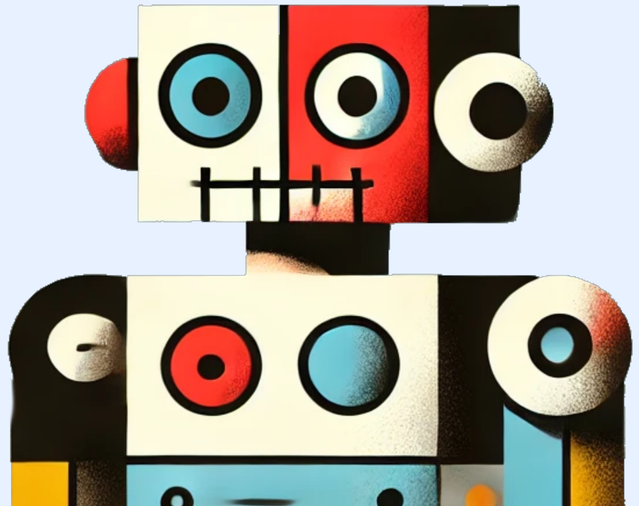
Part 1

a Series of (total) disappointments

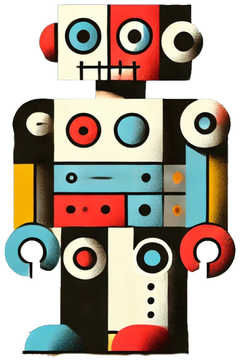
```
int parse_int(std::string_view number)
```



```cpp
int parse_int(std::string_view number) {
    int acc = 0;
    for(char c : number) {
        if(c < '0' || c > '9')   // TODO: +, -, digit separators?
            return acc;
        acc *= 10;
        acc += c-'0';
    }
    return acc;
}
```

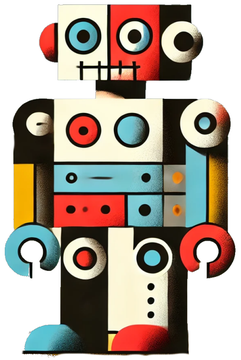


```
int parse_int(std::string_view number) {
    int acc = 0;
    for(char c : number) {
        if(c < '0' || c > '9')
            return acc;
        acc *= 10;
        acc += c-'0';
    }
    return acc;
}
```



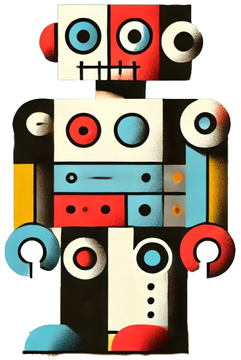
```
std::println("{}", parse_int("42") );
std::println("{}", parse_int("42x") );
std::println("{}", parse_int("x42") );
```

```
42
42
0
```



```
bool is_int(std::string_view number) {
   for(char c : number) {
       if(c < '0' || c > '9')
           return false;
   }
    return true;
}
```

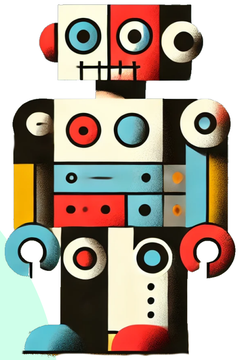
```
[[nodiscard]]
bool is_int(std::string_view number) {
   for(char c : number) {
       if(c < '0' || c > '9')
           return false;
   }
    return true;
}
```



```
void test(std::string_view str) {
    if(is_int(str))
        std::println("{} is an int", str);
    else
        std::println("{} is not an int", str);
}
```

```
[[nodiscard]]
bool is_int(std::string_view number) {
   for(char c : number) {
       if(c < '0' || c > '9')
           return false;
   }
    return true;
}
```

```
void test(std::string_view str) {
    if(is_int(str))
        std::println("{} is an int", str);
    else
        std::println("{} is not an int", str);
}
```



```
test("42");
test("42x");
test("x42");
```

→ 42 is an int

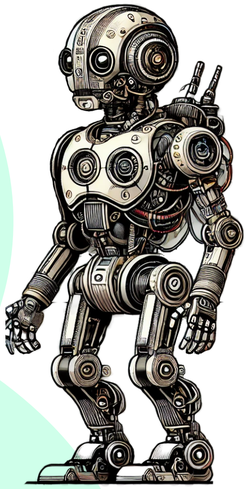
→ 42x is not an int

→ x42 is not an int

```cpp
enum class ParseStatus {
    Numeric,
    StartsNumerically,
    NonNumeric
};
```

```cpp
[[nodiscard]]
ParseStatus is_int(std::string_view number) {
    bool first_digit = true;
    for(char c : number) {
        if(c < '0' || c > '9') {
            return first_digit
                ? ParseStatus::NonNumeric
                : ParseStatus::StartsNumerically;
        }
        first_digit = false;
    }
    return ParseStatus::Numeric;
}
```

```cpp
void test(std::string_view str) {
    switch(is_int(str)) {
        case ParseStatus::Numeric:
            std::println("{} is an int", str);
            break;
        case ParseStatus::StartsNumerically:
            std::println("{} starts with an int", str);
            break;
        case ParseStatus::NonNumeric:
            std::println("{} is not an int", str);
            break;
    }
}
```



```cpp
test("42");
test("42x");
test("x42");
```

→

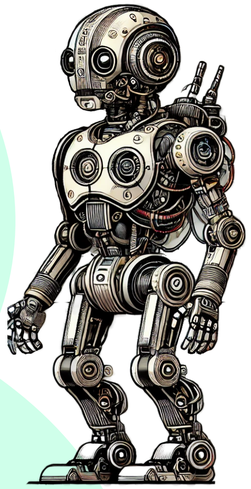
→

→

```cpp
enum class ParseStatus {
    Numeric,
    StartsNumerically,
    NonNumeric
};
```

```cpp
[[nodiscard]]
ParseStatus is_int(std::string_view number) {
    bool first_digit = true;
    for(char c : number) {
        if(c < '0' || c > '9') {
            return first_digit
                ? ParseStatus::NonNumeric
                : ParseStatus::StartsNumerically;
        }
        first_digit = false;
    }
    return ParseStatus::Numeric;
}
```

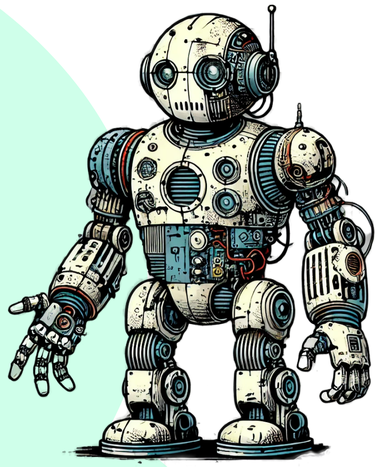
```cpp
void test(std::string_view str) {
    switch(is_int(str)) {
        case ParseStatus::Numeric:
            std::println("{} is an int", str);
            break;
        case ParseStatus::StartsNumerically:
            std::println("{} starts with an int", str);
            break;
        case ParseStatus::NonNumeric:
            std::println("{} is not an int", str);
            break;
    }
}
```



```cpp
test("42");
test("42x");
test("x42");
```

```
42 is an int
42x starts with an int
x42 is not an int
```

```
[[nodiscard]]
ParseStatus parse_int(std::string_view number, int& out) {
  out = 0;
  bool first_digit = true;
  for(char c : number) {
    if(c < '0' || c > '9') {
      return first_digit
        ? ParseStatus::NonNumeric
        : ParseStatus::StartsNumerically;
    }
    out *= 10;
    out += c-'0';
    first_digit = false;
  }
  return ParseStatus::Numeric;
}
```

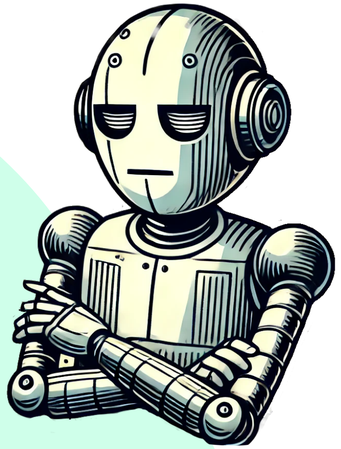


```
void test(std::string_view str) {
  switch(int i; parse_int(str)) {
    case ParseStatus::Numeric:
      std::println("{}", i);
      break;
    case ParseStatus::StartsNumerically:
      std::println("{} (starts with an int)”, i);
      break;
    case ParseStatus::NonNumeric:
      std::println(“Error: {} is not an int", str);
      break;
  }
}
```

```
test("42");   ⟶  42
test("42x");  ⟶  42 (starts with an int)
test("x42");  ⟶  Error: x42 is not an int
```

```cpp
int parse_int(std::string_view number) {
    int out = 0;
    bool first_digit = true;
    for(char c : number) {
        if(c < '0' || c > '9') {
            errno = first_digit
                ? EINVAL
                : EDOM;
            return out;
        }
        out *= 10;
        out += c-'0';
        first_digit = false;
    }
    return out;
}
```

```cpp
void test(std::string_view str) {
    int i = parse_int(str);
    if(errno == 0)
        std::println("{}", i);
    else
        std::println("Error ({})", strerror(errno));
}
```

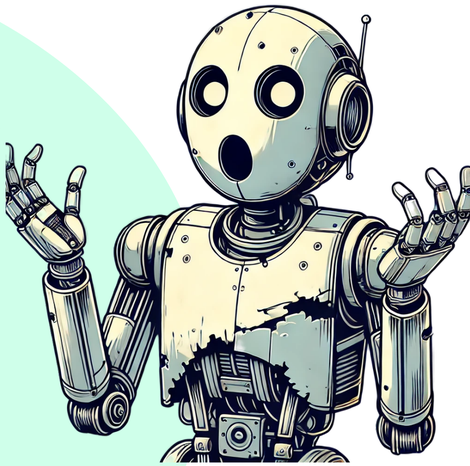


```cpp
test("42");
test("42x");
test("x42");
```

```
42
Error (Numerical argument out of domain)
Error (Invalid argument)
```

```cpp
int parse_int(std::string_view number) {
    int out = 0;
    bool first_digit = true;
    for(char c : number) {
        if(c < '0' || c > '9') {
            if(first_digit)
                throw std::runtime_error(
                    std::format("'{}' is not a number", number));
            else
                throw std::runtime_error(
                    std::format("'{}' has non-numeric digits", number));
        }
        out *= 10;
        out += c-'0';
        first_digit = false;
    }
    return out;
}
```

```cpp
void test(std::string_view str) {
    try {
        int i = parse_int(str);
        std::println("{}", i);
    }
    catch(std::exception& ex) {
        std::println("Error {}", ex.what());
    }
}
```

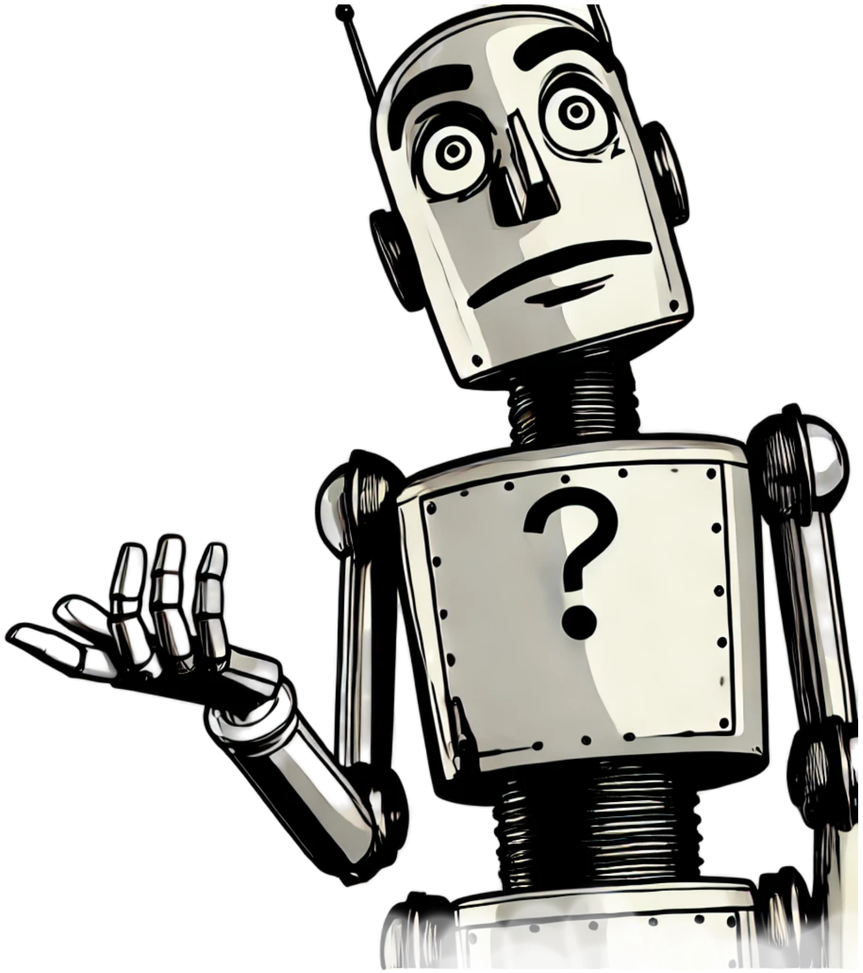


```cpp
test("42");
test("42x");
test("x42");
```

```
42
Error '42x' has non-numeric digits
Error 'x42' is not a number
```

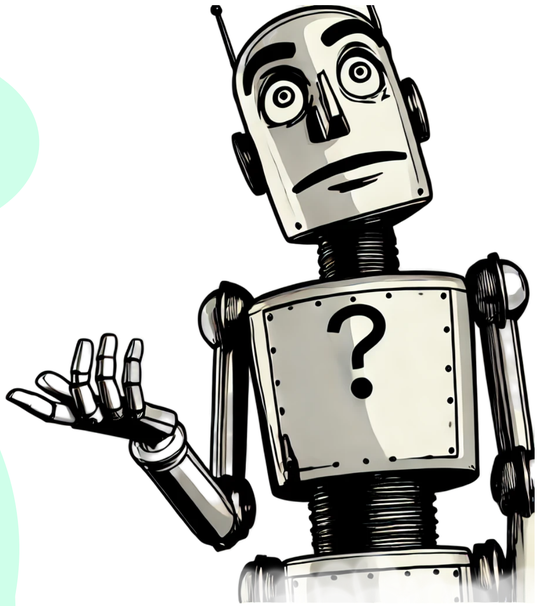
C++17

std::optional



```cpp
std::optional<int> parse_int(std::string_view number) {
    int out = 0;
    for(char c : number) {
        if(c < '0' || c > '9')
            return {};
        out *= 10;
        out += c-'0';
    }
    return out;
}
```

```cpp
void test(std::string_view str) {
    if(auto oi = parse_int(str))
        std::println("{}", *oi);
    else
        std::println("Error");
}
```

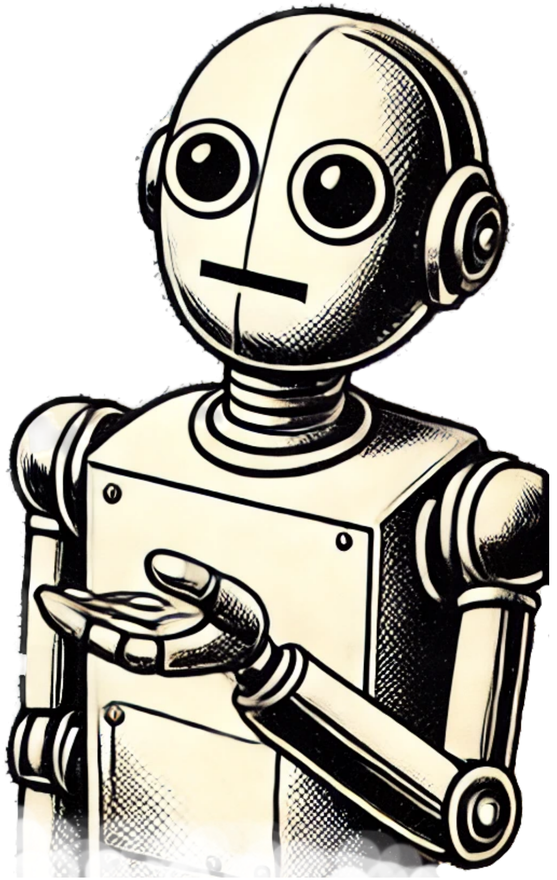



```
test("42");
test("42x");
test("x42");
```

```
42
Error
Error
```

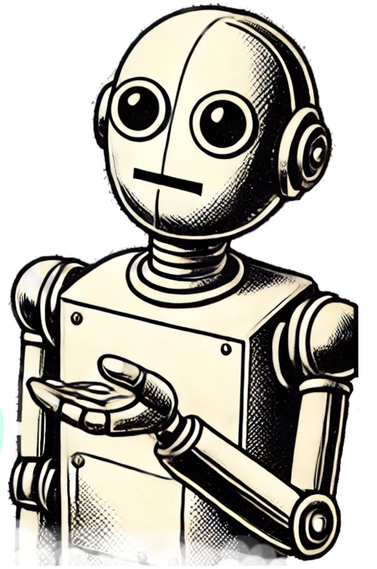
C++23

std::expected



```
std::expected<int, std::runtime_error>
parse_int(std::string_view number) {
    int out = 0;
    bool first_digit = true;
    for(char c : number) {
        if(c < '0' || c > '9') {
            return first_digit
                ? std::unexpected(
                    std::runtime_error("passed string is not a number"))
                : std::unexpected(
                    std::runtime_error("passed string has non-numeric digits"));
        }
        out *= 10;
        out += c-'0';
        first_digit = false;
    }
    return out;
}
```

```
void test(std::string_view str) {
    if(auto ei = parse_int(str))
        std::println("{}", *ei);
    else
        std::println("Error: {}", ei.error().what());
}
```

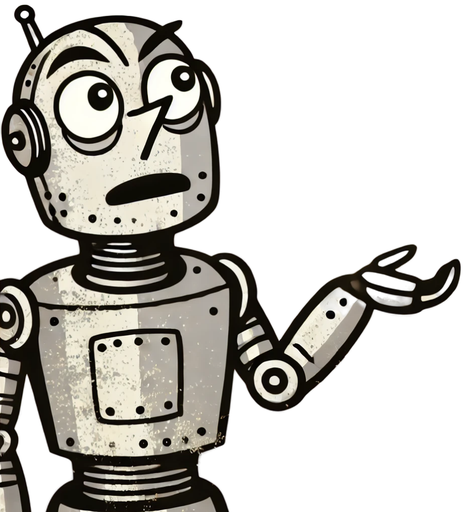


```
test("42");   ⟶
test("42x");  ⟶
test("x42");  ⟶
```

```
42
Error: passed string has non-numeric digits
Error: passed string is not a number
```

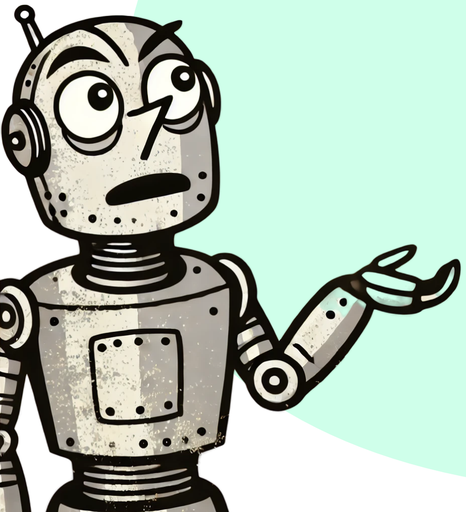
```
std::expected<int, std::runtime_error>
parse_int(std::string_view number) {
  // ...
}
```

```
void test(std::string_view str) {
  std::expected<float, std::runtime_error> ef;
  if(auto ei = parse_int(str)) {
    int i = *ei-1;
    if(i != 0)
      ef = 1.f / i; // happy path
    else
      ef = std::unexpected(std::runtime_error("Divide by zero"));
  }
  else
    ef = std::unexpected(std::move(ei.error()));

  if(ef)
    std::println("{}", *ef);
  else
    std::println("Error: {}", ef.error().what());
}
```

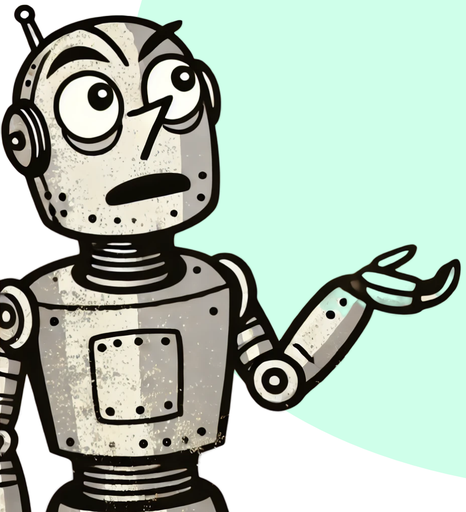


```
test("42");   ⟶  0.024390243
test("42x");  ⟶  Error: passed string is not a number
test("x42");  ⟶  Error: Divide by zero
```

```cpp
void test(std::string_view str) {
    std::expected<float, std::runtime_error> ef;
    if(auto ei = parse_int(str)) {
        int i = *ei-1;
        if(i != 0)
            ef = 1.f / i; // happy path
        else
            ef = std::unexpected(std::runtime_error("Divide by zero"));
    }
    else
        ef = std::unexpected(std::move(ei.error()));

    if(ef)
        std::println("{}", *ef);
    else
        std::println("Error: {}", ef.error().what());
}
```

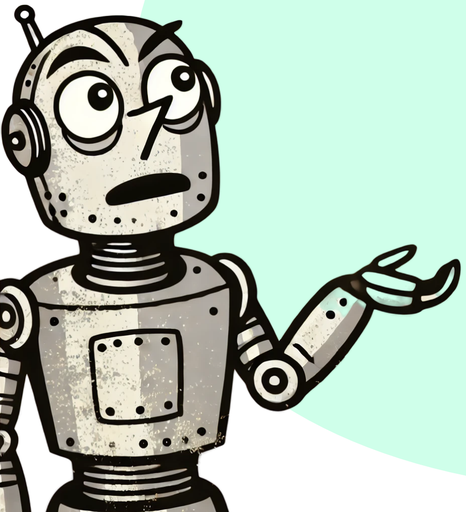


```cpp
void test(std::string_view str) {
    std::expected<float, std::runtime_error> ef;
    if(auto ei = parse_int(str)) {
        int i = *ei-1;
        if(i != 0)
            ef = 1.f / i; // happy path
        else
            ef = std::unexpected(std::runtime_error("Divide by zero"));
    }
    else
        ef = std::unexpected(std::move(ei.error()));

    if(ef)
        std::println("{}", *ef);
    else
        std::println("Error: {}", ef.error().what());
}
```



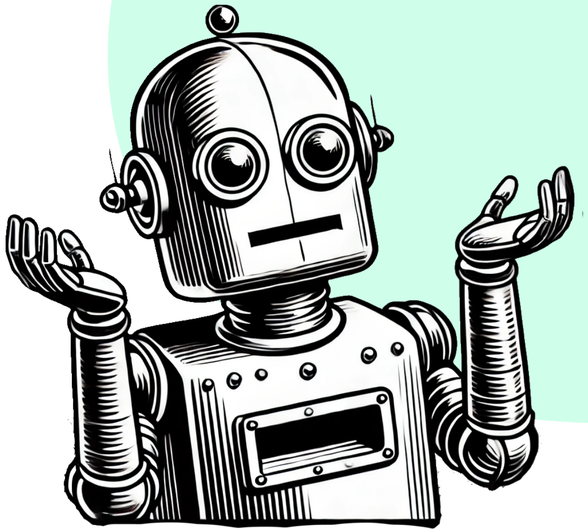
```
0.024390243
Error: passed string is not a number
Error: Divide by zero
```

```cpp
void test(std::string_view str) {
    std::expected<float, std::runtime_error> ef;
    if(auto ei = parse_int(str); !ei)
        ef = std::unexpected(std::move(ei.error()));
    else {
        int i = *ei-1;
        if(i == 0)
            ef = std::unexpected(std::runtime_error("Divide by zero"));
        else
            ef = 1.f / i; // happy path
    }

    if(ef)
        std::println("{}", *ef);
    else
        std::println("Error: {}", ef.error().what());
}
```



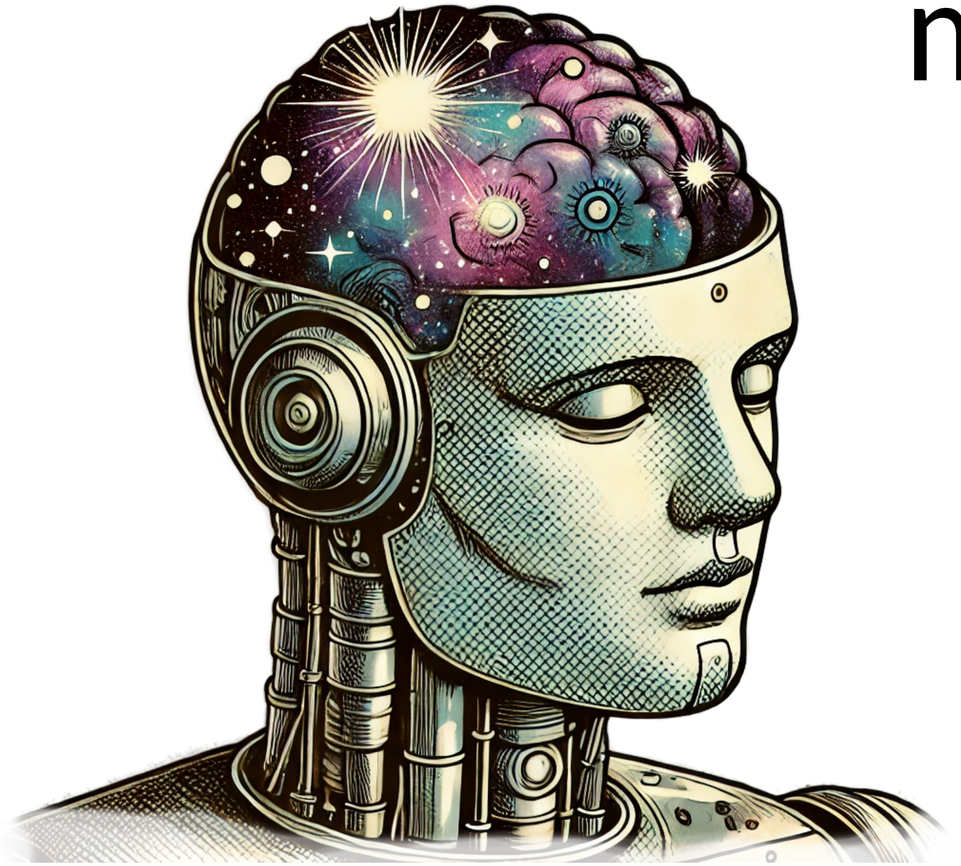
```
0.024390243
Error: passed string is not a number
Error: Divide by zero
```

```cpp
void test(std::string_view str) {
    auto ef = [str]() -> std::expected<float, std::runtime_error> {
        auto ei = parse_int(str);
        if(!ei)
            return std::unexpected(std::move(ei.error())); // propagate error
        int i = *ei-1;
        if(i == 0)
            return std::unexpected(std::runtime_error("Divide by zero"));
        return 1.f / i; // happy path
    }();

    if(ef)
        std::println("{}", *ef);
    else
        std::println("Error: {}", ef.error().what());
}
```



```
0.024390243
Error: passed string is not a number
Error: Divide by zero
```

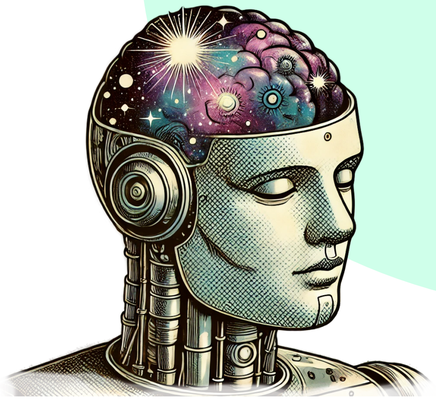
C++23



monadic operations for

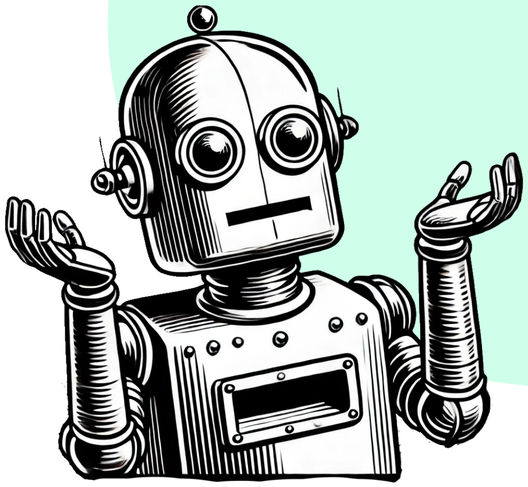
std::expected and
std::optional

```
void test(std::string_view str) {
    auto ef = parse_int(str)
        .transform([](int i)
            { return i-1; } )
        .and_then([](int i) -> std::expected<float, std::runtime_error>
            {   if(i != 0)
                    return 1.f / i;
                else
                    return std::unexpected(std::runtime_error("Divide by zero")); } );

    if(ef)
        std::println("{}", *ef);
    else
        std::println("Error: {}", ef.error().what());
}
```



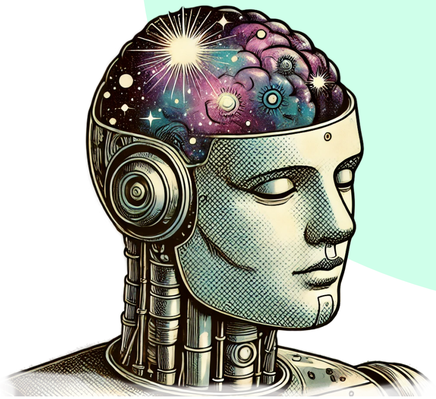
```
0.024390243
Error: passed string is not a number
Error: Divide by zero
```

```cpp
void test(std::string_view str) {
    auto ef = [str]() -> std::expected<float, std::runtime_error> {
        auto ei = parse_int(str);
        if(!ei)
            return std::unexpected(std::move(ei.error())); // propagate error
        int i = *ei-1;
        if(i == 0)
            return std::unexpected(std::runtime_error("Divide by zero"));
        return 1.f / i; // happy path
    }();

    if(ef)
        std::println("{}", *ef);
    else
        std::println("Error: {}", ef.error().what());
}
```



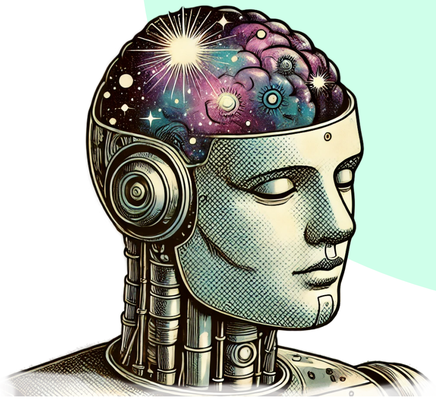
```
0.024390243
Error: passed string is not a number
Error: Divide by zero
```

```
void test(std::string_view str) {
    auto ef = parse_int(str)
        .transform([](int i)
            { return i-1; } )
        .and_then([](int i) -> std::expected<float, std::runtime_error>
            {  if(i != 0)
                    return 1.f / i;
                else
                    return std::unexpected(std::runtime_error("Divide by zero")); } );

    if(ef)
        std::println("{}", *ef);
    else
        std::println("Error: {}", ef.error().what());
}
```



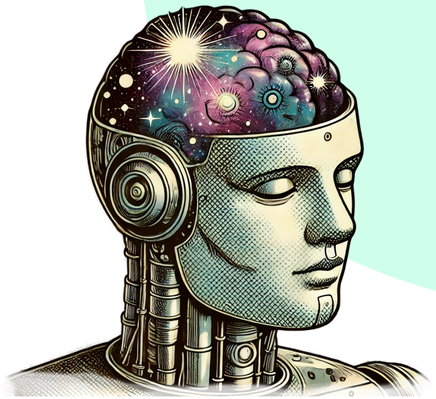
```
0.024390243
Error: passed string is not a number
Error: Divide by zero
```

```cpp
void test(std::string_view str) {
    auto ef = parse_int(str)
        .transform([](int i)
            { return i-1; } )
        .and_then([](int i) -> std::expected<float, std::runtime_error>
            {  if(i != 0)
                    return 1.f / i;
                else
                    return std::unexpected(std::runtime_error("Divide by zero")); } )
        .transform([](float f) {
            std::println("{}", f); return f; } ) // Happy path
        .transform_error([](auto const& ex) {
            std::println("Error: {}", ex.what()); return ex; } ); // Error path
}
```



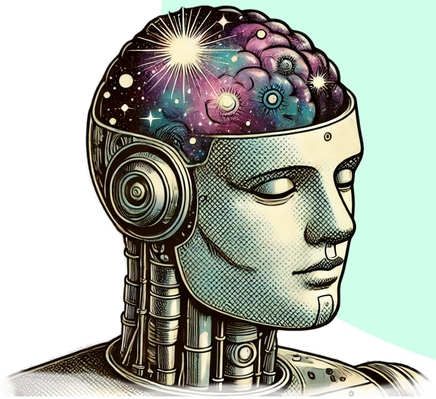
```
0.024390243
Error: passed string is not a number
Error: Divide by zero
```

```
void test(std::string_view str) {
    auto ef = parse_int(str)
        .transform([](int i)
            { return i-1; } )
        .and_then([](int i) -> std::expected<float, std::runtime_error>
            {  if(i != 0)
                    return 1.f / i;
                else
                    return std::unexpected(std::runtime_error("Divide by zero")); } )
        .or_else([](auto) -> std::expected<float, std::runtime_error> {
            return 0.f; } );

    std::println("{}", *ef);
}
```



```
0.024390243
0
0
```

```cpp
void test(std::string_view str) {
    auto ef = parse_int(str)
        .transform([](int i)
            { return i-1; } )
        .and_then([](int i) -> std::expected<float, std::runtime_error>
            {  if(i != 0)
                    return 1.f / i;
                else
                    return std::unexpected(std::runtime_error("Divide by zero")); } )
        .transform_error([](auto const& ex)
            { return std::string(ex.what()); } );
    if(ef)
        std::println("{}", *ef);
    else
        std::println("Error: {}", ef.error());
}
```



```
0.024390243
Error: passed string is not a number
Error: Divide by zero
```

std::optional

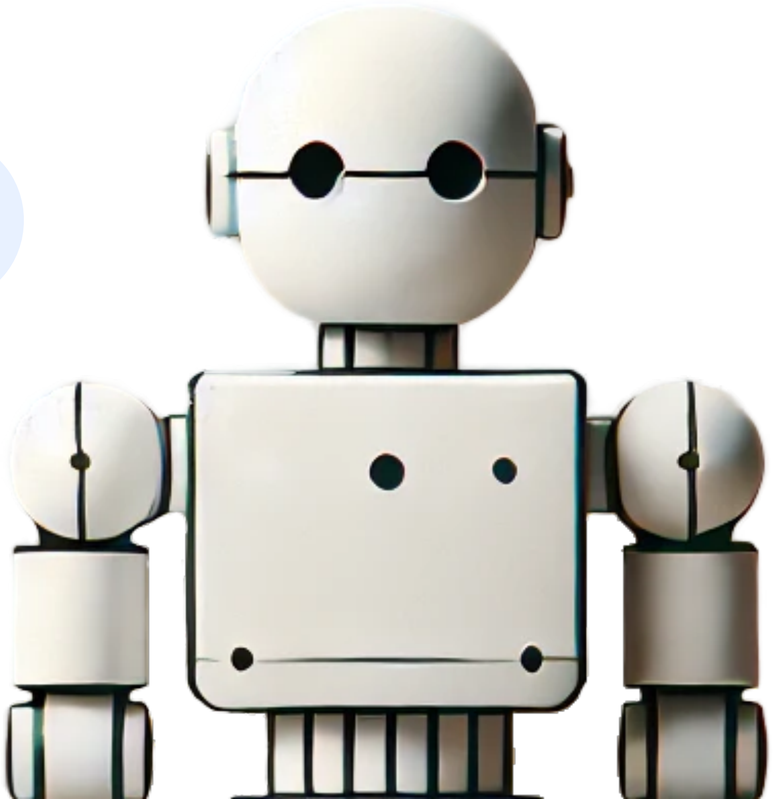
- **and_then** (f(v) -> E<v,e>)
- **transform** (f(v) -> v)
- **or_else** (f(e) -> v)

std::expected

- **transform_error** (f(e) -> e)

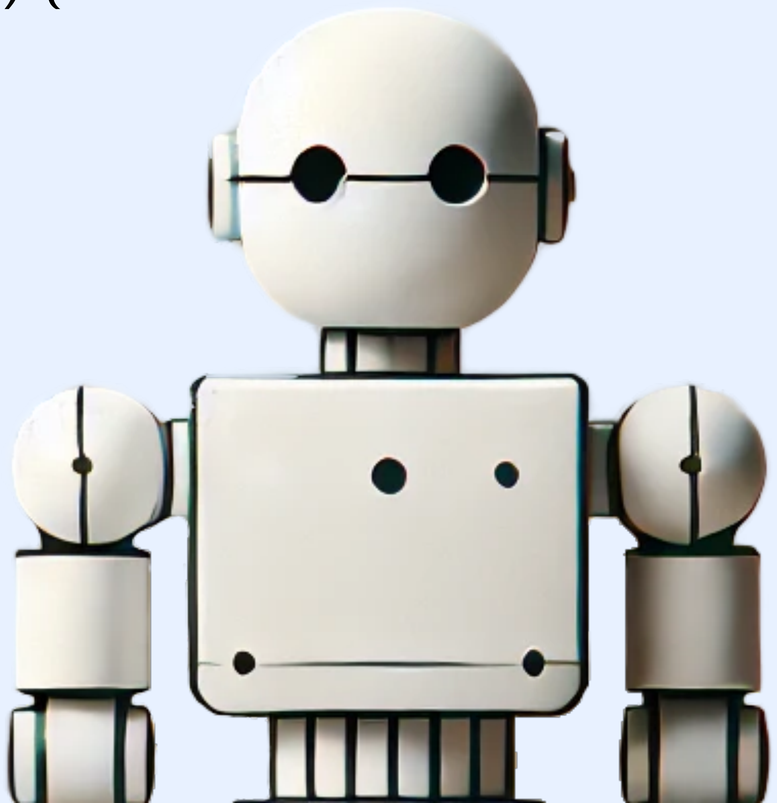
Part 2

```
std::string month_as_string(int month)
```

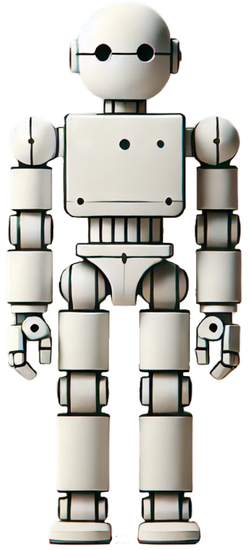


```
std::string month_names[] =
    {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
```

```
std::string month_as_string(int month) {
    return month_names[month-1];
}
```



```
std::string month_as_string(int month) {
    return month_names[month-1];
}
```



```
std::println("Month: {}", month_as_string(1));

std::println("Month: {}", month_as_string(0));
std::println("Month: {}", month_as_string(10000));
```

```
Month: Jan
Month:
Month: P7_sFILENS_19basic_for yv$oj $o
```

```
std::string month_as_string(int month) {
    if(month < 1 || month > 12)
        throw std::out_of_range("month must be between 1 and 12");
    return month_names[month-1];
}
```

```
try {
    std::println("Month: {}", month_as_string(0));
}
catch(std::exception& ex) {
    std::println("Error: {}", ex.what());
}
```

```
Error: month must be between 1 and 12
```

```
std::string month_as_string(int month) {
    assert(month >= 1 && month <= 12);
    return month_names[month-1];
}
```

```
void test_if(int month) {
    if(month >= 1 && month <=12)
        std::println("Month: {}", month_as_string(month));
    else
        std::println("Error: month ({}) is not between 1 and 12", month);
}
```

```
test_if(0);
test_if(1);
```

```
Error: month (0) is not between 1 and 12
Month: Jan
```

```cpp
std::string month_as_string(int month) {
    assert(is_valid_month(month));
    return month_names[month-1];
}
```

```cpp
std::expected<std::string, std::string> month_as_string_checked(int month) {
    if(!is_valid_month(month))
        return std::unexpected("month must be between 1 and 12");
    return month_as_string(month);
}
```

```cpp
test(0);
test(1);
```

```
Error: month (0) is not between 1 and 12
Month: Jan
```

```cpp
void test(int month) {
    if(auto result = month_as_string_checked(month))
        std::println("Month {}", *result);
    else
        std::println("Error: {}", result.error());
}
```

```
std::string month_as_string_unchecked(int month) {
    assert(is_valid_month(month));
    return month_names[month-1];
}
```

```
std::expected<std::string, std::string> month_as_string(int month) {
    if(!is_valid_month(month))
        return std::unexpected("month must be between 1 and 12");
    return month_as_string(month);
}
```

```
test(0);
test(1);
```

```
Error: month (0) is not between 1 and 12
Month: Jan
```

```
void test(int month) {
    if(auto result = month_as_string(month))
        std::println("Month {}", *result);
    else
        std::println("Error: {}", result.error());
}
```

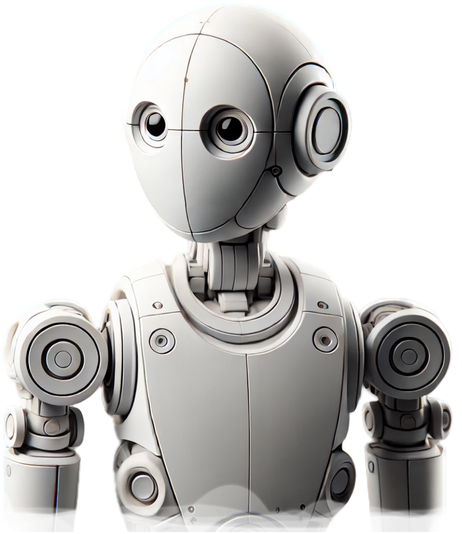
```cpp
std::string month_as_string_unchecked(int month) {
    assert(is_valid_month(month));
    return month_names[month-1];
}

std::string month_as_string_checked(int month) {
    if(!is_valid_month(month))
        throw std::out_of_range("month must be between 1 and 12");
    return month_as_string(month);
}
```

```cpp
void test(int month) {
    try {
        std::println("Month: {}", month_as_string(month));
    }
    catch(std::exception& ex) {
        std::println("Error: {}", ex.what());
    }
}
```

```cpp
test(0);
test(1);
```

```
Error: month (0) is not between 1 and 12
Month: Jan
```

```cpp
enum class CheckMode { Checked, Unchecked };

template<CheckMode check_mode = CheckMode::Checked>
std::string month_as_string(int month) {
    if constexpr(check_mode == CheckMode::Checked)
        if(!is_valid_month(month))
            throw std::out_of_range("month must be between 1 and 12");
    assert(is_valid_month(month));
    return month_names[month-1];
}
```

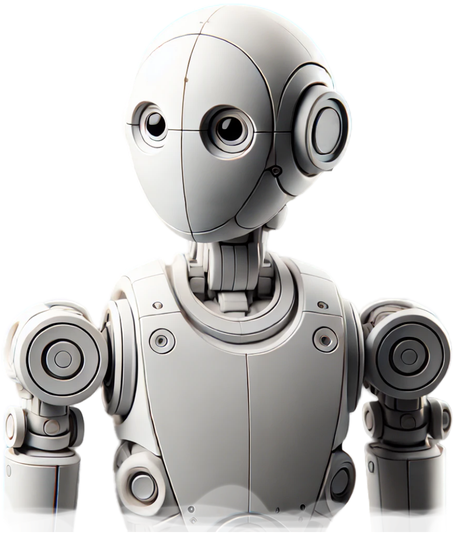


```cpp
test(0);
test(1);
```

```
Error: month (0) is not between 1 and 12
Month: Jan
```

```cpp
void test(int month) {
    try {
        std::println("Month: {}", month_as_string(month));
    }
    catch(std::exception& ex) {
        std::println("Error: {}", ex.what());
    }
}
```

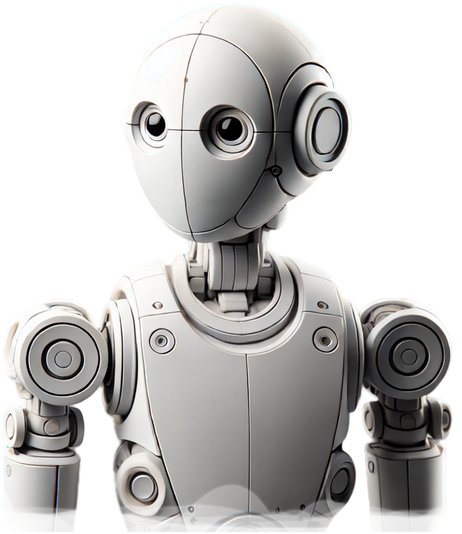
```cpp
enum class CheckMode { Checked, Unchecked };

template<CheckMode check_mode = CheckMode::Checked>
std::string month_as_string(int month) {
    if constexpr(check_mode == CheckMode::Checked)
        if(!is_valid_month(month))
            throw std::out_of_range("month must be between 1 and 12");
    assert(is_valid_month(month));
    return month_names[month-1];
}
```



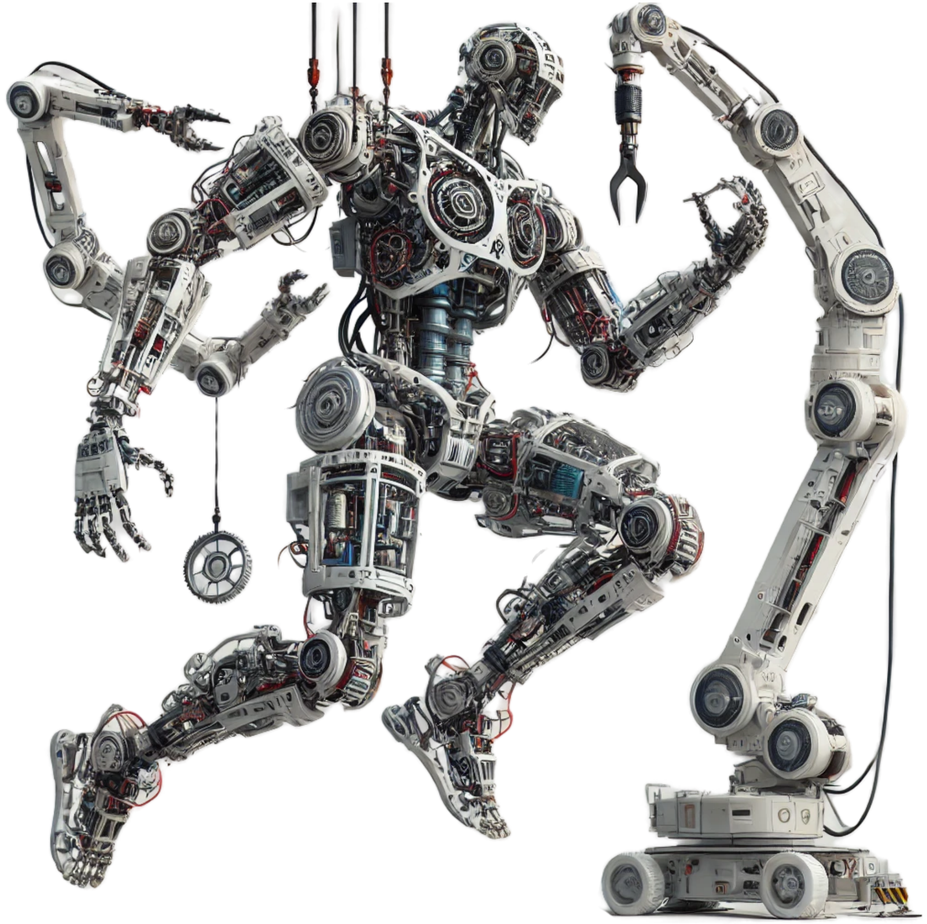
```cpp
month_as_string<CheckMode::Unchecked>(0);
```

```
Assertion failed: (is_valid_month(month)),
function month_as_string, file month_as_string.cpp, line 154.
```

```
enum class CheckMode { Checked, Unchecked };

#define ASSERT_OR_THROW(condition, exception_decl) \
   if constexpr(check_mode == CheckMode::Checked) \
   if(!(condition)) \
      throw exception_decl; \
   assert(condition)
```

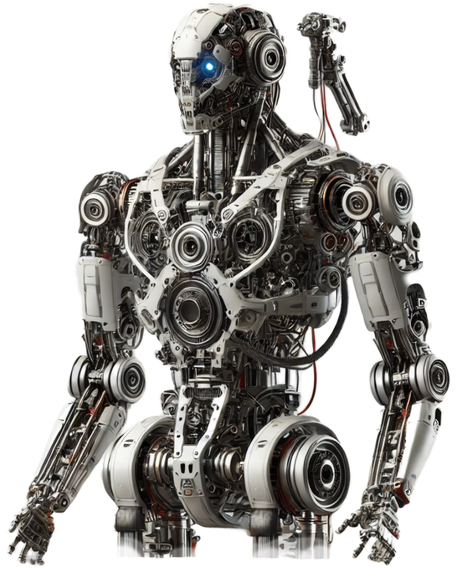


```
template<CheckMode check_mode = CheckMode::Checked>
std::string month_as_string(int month) {
   ASSERT_OR_THROW(month >= 1 && month <= 12,
      std::out_of_range("month must be between 1 and 12"));
   return month_names[month-1];
}
```

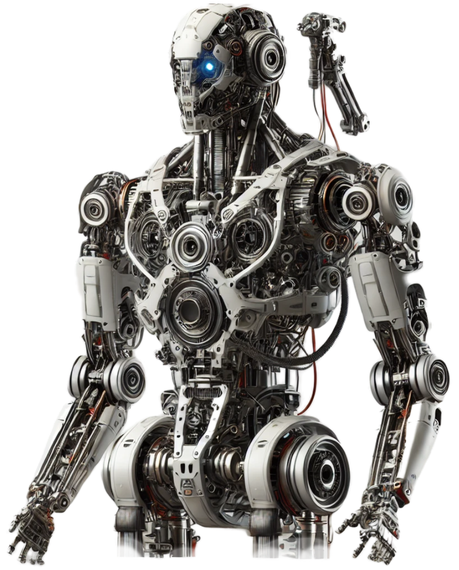


Correct by Construction

```
class Month {
    int month;
public:
    static bool is_valid(int month) {
        return month >= 1 && month <= 12;
    }
    Month(int month) : month(month) {
        if( !is_valid(month) )
            throw std::out_of_range("month must be between 1 and 12");
    }
    int value() const { return month; }
};

std::string month_as_string(Month month) {
    return month_names[month.value()-1];
}
```



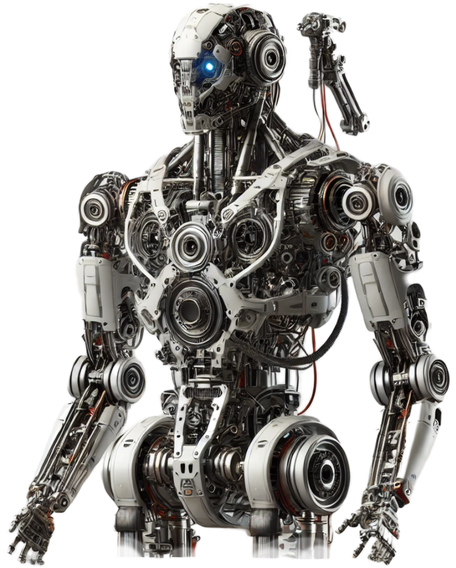
```
class Month {
    int month;
public:
    static bool is_valid(int month) {
        return month >= 1 && month <= 12;
    }
    Month(int month) : month(month) {
        if( !is_valid(month) )
            throw std::out_of_range("month must be between 1 and 12");
    }
    int value() const { return month; }
};

std::string month_as_string(Month month) {
    return month_names[month.value()-1];
}
```



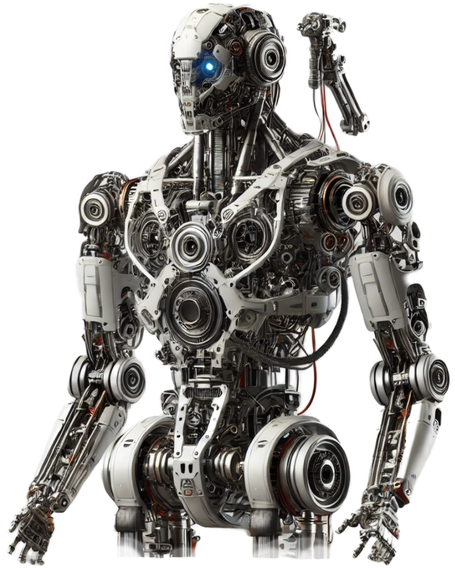
```
void test(int month) {
    try {
        std::println("Month: {}", month_as_string(Month(month)));
    }
    catch(std::exception& ex) {
        std::println("Error: {}", ex.what());
    }
}
```

```
struct unchecked{};
```

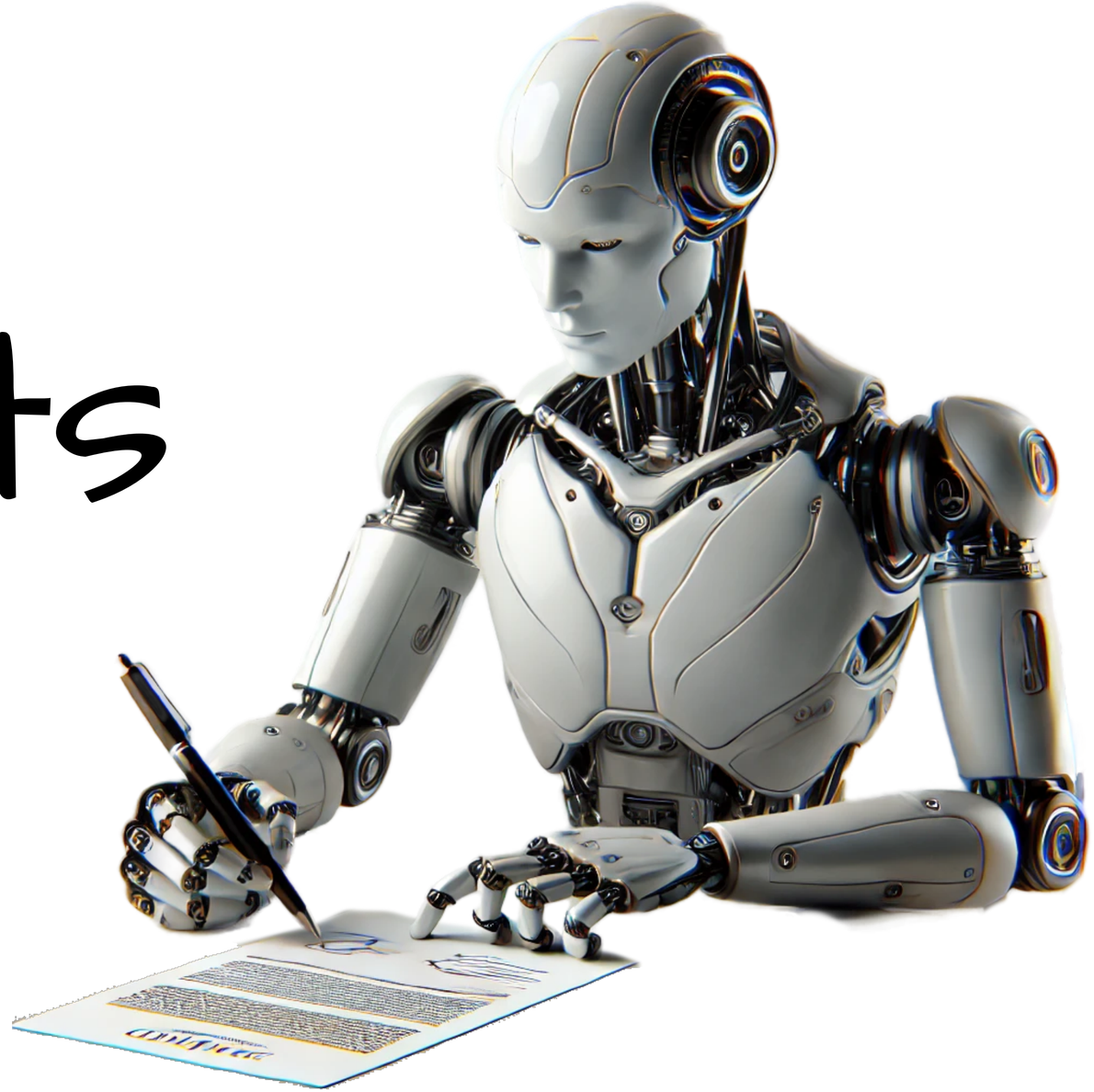
```
class Month {
    int month;
public:
    static bool is_valid(int month) {
        return month >= 1 && month <= 12;
    }
   Month(int month) : month(month) {
        if( !is_valid(month) )
            throw std::out_of_range("month must be between 1 and 12");
    }
    Month(int month, unchecked) : month(month) {
        assert(is_valid(month));
    }
    int value() const { return month; }
```



```
class Month {
    int month;
public:
    static bool is_valid(int month) {
        return month >= 1 && month <= 12;
    }
   Month(int month) : month(month) {
        if( !is_valid(month) )
            throw std::out_of_range("month must be between 1 and 12");
    }
    Month(int month, unchecked) : month(month) {
        assert(is_valid(month));
    }
    int value() const { return month; }
    std::string as_string() const {
        return month_names[month-1];
    }
```



Contracts



```cpp
bool is_valid_month(int month) {
    return month >= 1 && month <= 12;
}

std::string month_as_string(int month) {
    pre(is_valid_month(month));
    return month_names[month-1];
}
```

```cpp
void test(int month) {
    std::println("{}", month_as_string(month));
}
```

```cpp
test(0);
test(1);
```

```
Jan
month_as_string2.cpp:49:9: precondition failed: (is_valid_month(month)):
 in function: std::string month_as_string(int)
Process finished with exit code 134 (interrupted by signal 6:SIGABRT)
```

```
#ifndef NDEBUG
#define pre(expr) do{ if(!(expr)){ violation_handler({#expr}); } } while(false)
#else
#define pre(expr) do{} while(false)
#endif
```

```
std::string month_as_string(int month) {
    pre(is_valid_month(month));
    return month_names[month-1];
}
```

```
#ifndef NDEBUG
#define pre(expr) do{ if(!(expr)){ violation_handler({#expr}); } } while(false)
#else
#define pre(expr) do{} while(false)
#endif


std::string month_as_string(int month) {
    pre(is_valid_month(month));
    return month_names[month-1];
}
```

```
struct ViolationInfo{
    std::string_view expr;
    std::source_location loc = std::source_location::current();

    std::string make_message() const {
        return std::format("{}:{}:{}: precondition failed: ({}): in function: {}",
                loc.file_name(), loc.line(), loc.column(), expr, loc.function_name() );
    }
};

void default_violation_handler(ViolationInfo const& info) {
    std::print("{}", info.make_message());
    std::abort();
}

auto violation_handler = &default_violation_handler;
```

```
void throwing_violation_handler(ViolationInfo const& info) {
    throw std::logic_error(info.make_message());
}
```

```
void test(int month) {
    violation_handler = &throwing_violation_handler;
    try {
        std::println("{}", month_as_string(month));
    }
    catch(std::exception& ex) {
        std::println("Error: {}", ex.what());
    }
}
```

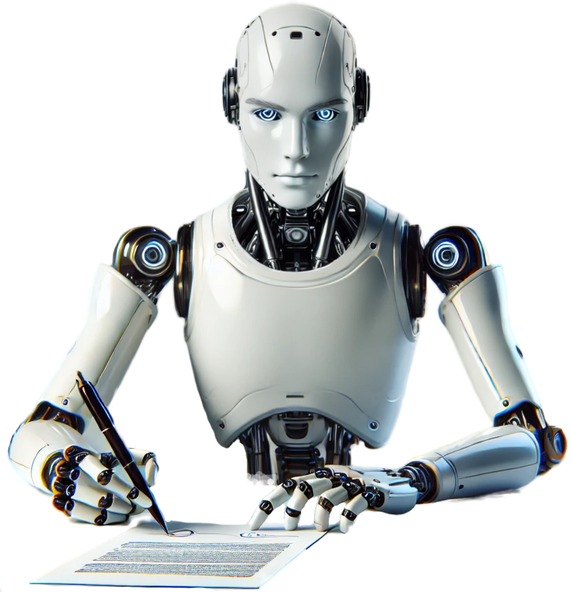
```
test(0);
test(1);
```

```
Jan
Error: month_as_string2.cpp:43:9: precondition failed: (is_valid_month(month)):
 in function: std::string month_as_string(int)
```

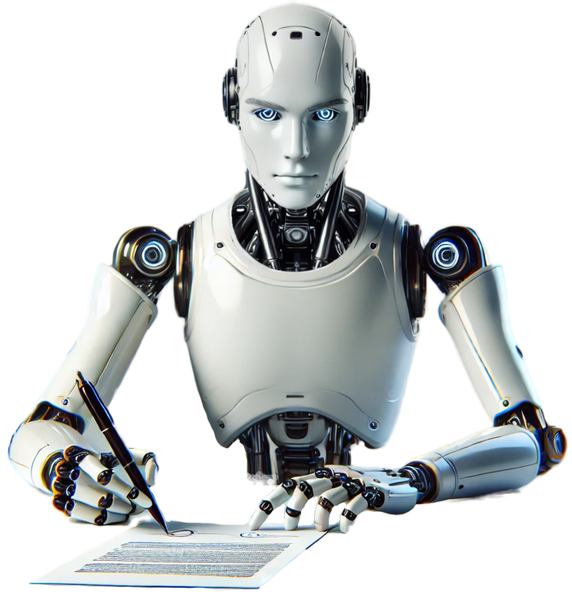
```
std::string month_as_string(int month) {
    pre(is_valid_month(month));
    return month_names[month-1];
}
```

```
std::string_view month_as_string(int month) {
    pre(is_valid_month(month));
    return month_names[month-1];
}
```

```
std::string_view month_as_string(int month) noexcept {
    pre(is_valid_month(month));
    return month_names[month-1];
}
```

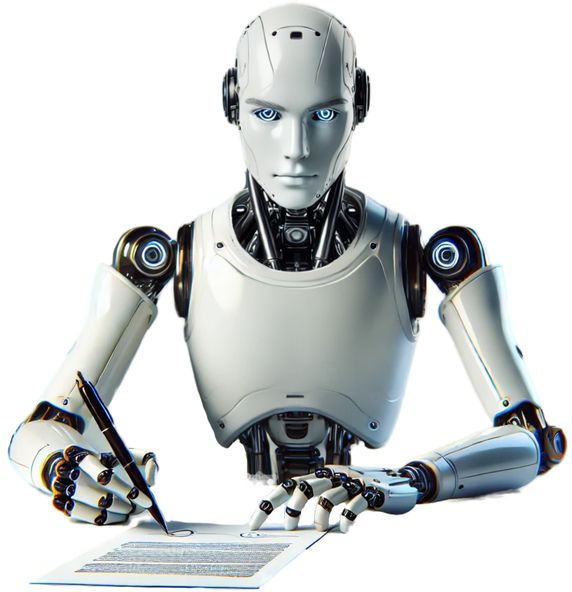


```
std::string_view month_as_string(int month) noexcept {
    pre(is_valid_month(month));
    return month_names[month-1];
}
```



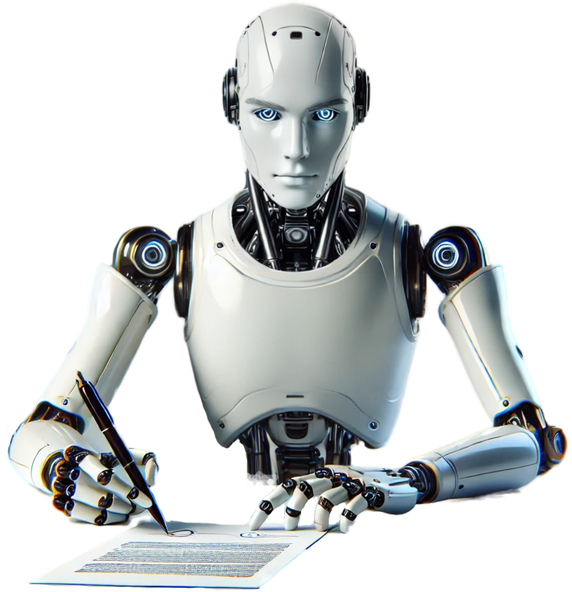
```
TEST_CASE("Out of bounds calls to month_as_string are caught by contract") {
    REQUIRE_THROWS(month_as_string(0));
    REQUIRE_THROWS(month_as_string(13));
    // ...
}
```

```
std::string_view month_as_string(int month) {
    pre(is_valid_month(month));
    return month_names[month-1];
}
```



```
TEST_CASE("Out of bounds calls to month_as_string are caught by contract") {
    REQUIRE_THROWS(month_as_string(0));
    REQUIRE_THROWS(month_as_string(13));
    // ...
}
```

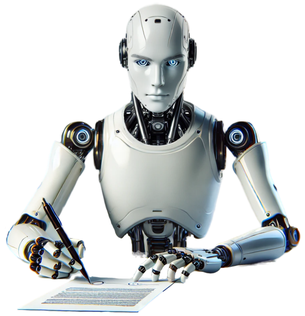
```
std::string_view month_as_string(int month) {
    pre(is_valid_month(month));
    return month_names[month-1];
}
```



the “Lakos Rule” (p2861)

```
TEST_CASE("Out of bounds calls to month_as_string are caught by contract") {
    REQUIRE_THROWS(month_as_string(0));
    REQUIRE_THROWS(month_as_string(13));
    // ...
}
```

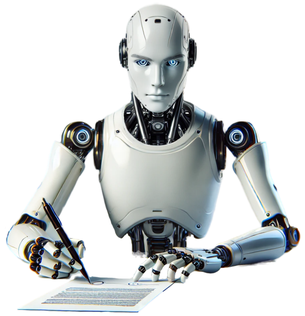
the “Lakos Rule”



“Absent a vanishingly rare and compelling engineering reason to do otherwise, deviating from the Lakos Rule is invariably and absolutely a terribly bad idea, especially within the C++ Standard Library specification.”

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/**p2861**r0.pdf

Working around the “Lakos Rule” (for testing)



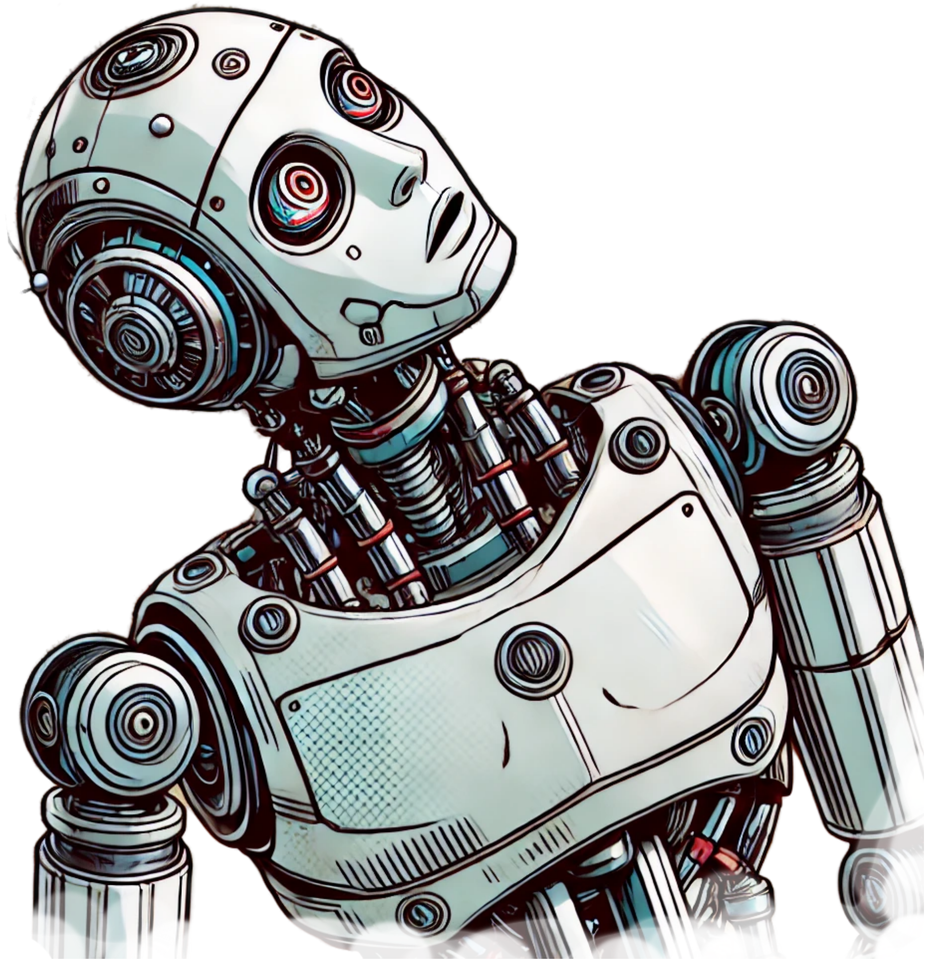
👍 1. Don’t use noexcept with preconditions you want to test

👍 2. For simple preconditions (like is_valid_month()) consider if just testing the predicate is enough

👍 3. Consider the overload/ checking policy idiom and test the checked version (conditionally noexcept)

🤔 3. Wrap noexcept in a macro that you can conditionally compile for testing purposes

🤞 4. Use "death tests" to see if the process actually terminates



Summary

Summary

Disappointments

bool/ code return

- Separate tester (e.g. is_int)

Out param

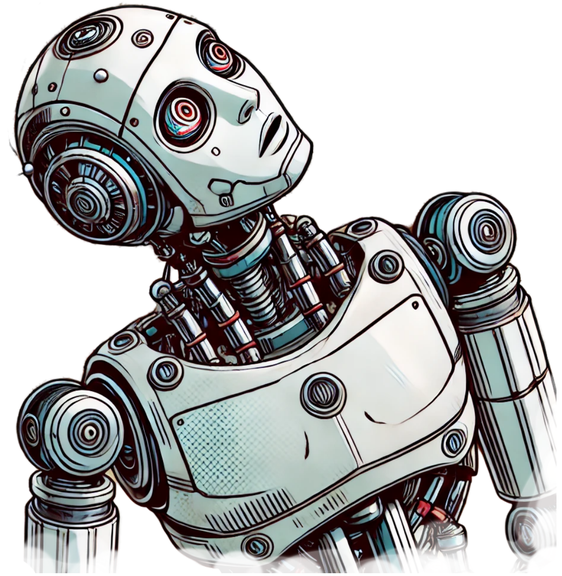
Global/ TL variable

- Boost Leaf?

Exceptions

optional/ expected

- Raw
- Raw in lambda
- Monadic operations



Narrow Contracts

assert

Checked/ unchecked versions

Policy-based Checking

Correct-by-construction

Preconditions

Throwing violation-handler

- No noexcept

Modern C++ Error Handling

Phil Nash

SHAVEDYAKS

CppCast