# Monadic Operations in C++23


ROBERT SCHIMKOWITSCH

https://mastodon.social/@asperamanca
https://github.com/Asperamanca/
https://cppusergroupvienna.org/

# Calling successive functions that might fail

```
bool getNumericTableValueNegative
    (const CDb& db, const Key& key,
     const CLocation& location, bool& out)
{
    CElement element;
    if ( ! getElement(db, key, element))          { return false; }

    CTableData table;
    if ( ! getTable(element, table))              { return false; }

    CTableCell cell;
    if ( ! getCell(table, location, cell))        { return false; }

    int value;
    if ( ! getNumericCellValue(cell, value))      { return false; }

    result = (value < 0);
    return true;
}
```

# Calling successive functions that might fail

```
bool getNumericTableValueNegative
     (const CDb& db, const Key& key,
      const CLocation& location, bool& out)
{
     CElement element;
     if ( ! getElement(db, key, element))          { return false; }

     CTableData table;
     if ( ! getTable(element, table))              { return false; }

     CTableCell cell;
     if ( ! getCell(table, location, cell))        { return false; }

     int value;
     if ( ! getNumericCellValue(cell, value))      { return false; }

     result = (value < 0);
     return true;
}
```

# ...with exceptions

```
throw std::out_of_range("row out of bounds");
```

```
bool getNumericTableValueNegative(const CDb& db, const Key& key,
                                  const CLocation& location)
{
    auto table = getTable(getElement(db,key));
    auto cell = getCell(table, cellLocation);
    return (getNumericCellValue(cell) < 0);
}
```

```
catch (const std::invalid_argument& e)
{
    //...
}
catch (const std::out_of_range& e)
{
    //...
}
catch (...) //...
```

# But I can't or won't use Exceptions!

```
bool getNumericTableValueNegative
    (const CDb& db, const Key& key,
     const CLocation& location, bool& out)
{
    CElement element;
    if ( ! getElement(db, key, element))          { return false; }

    CTableData table;
    if ( ! getTable(element, table))              { return false; }

    CTableCell cell;
    if ( ! getCell(table, location, cell))        { return false; }

    int value;
    if ( ! getNumericCellValue(cell, value))      { return false; }

    result = (value < 0);
    return true;
}
```

# Does an error flag help?

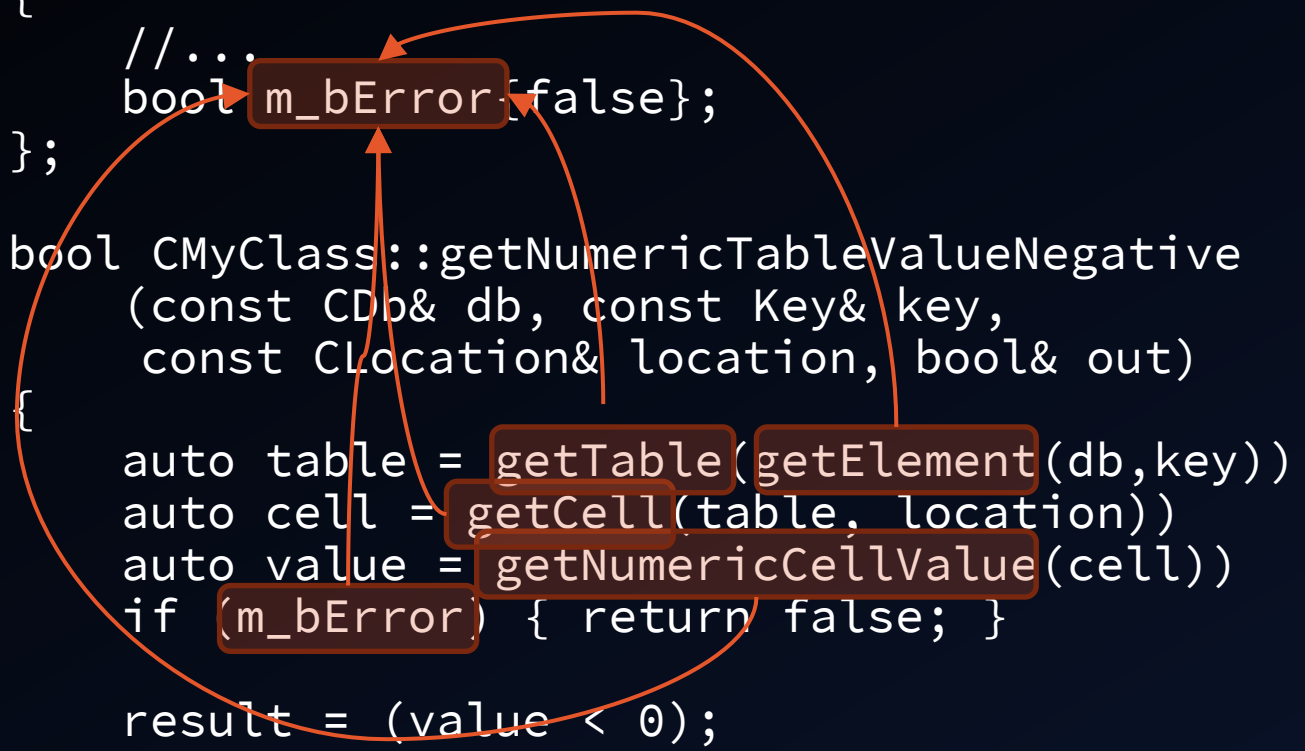
```
class CMyClass
{
    //...
    bool m_bError{false};
};

bool CMyClass::getNumericTableValueNegative
    (const CDb& db, const Key& key,
     const CLocation& location, bool& out)
{
    auto table = getTable(getElement(db,key))
    auto cell = getCell(table, location))
    auto value = getNumericCellValue(cell))
    if (m_bError) { return false; }

    result = (value < 0);
    return true;
}
```

# Does an error flag help?

```
class CMyClass
{
    //...
    bool m_bError{false};
};

bool CMyClass::getNumericTableValueNegative
    (const CDb& db, const Key& key;
     const CLocation& location, bool& out)
{
    auto table = getTable(getElement(db,key))
    auto cell = getCell(table, location))
    auto value = getNumericCellValue(cell))
    if (m_bError) { return false; }

    result = (value < 0);
    return true;
}
```

```
CTableData CMyClass::getTable
{
    if (m_bError)
    {
      return{};
    }
    //...
```

# Does an error flag help?



```
class CMyClass
{
    //...
    bool m_bError{false};
};

bool CMyClass::getNumericTableValueNegative
    (const CDb& db, const Key& key,
     const CLocation& location, bool& out)
{
    auto table = getTable(getElement(db,key))
    auto cell = getCell(table, location))
    auto value = getNumericCellValue(cell))
    if (m_bError) { return false; }

    result = (value < 0);
    return true;
}
```

# Fixing the return type with std::optional

```
std::optional<bool> isNumericTableCellValueNegative
      (const CDb& db, const Key& key, const CLocation& location)
{
      auto oElement = getElement(db,key);
      if ( ! oElement.has_value()) { return {}; }

      auto oTable = getTable(oElement.value());
      if ( ! oTable.has_value())   { return {}; }

      auto oCell = getCell(oTable.value(),location);
      if ( ! oCell.has_value())    { return {}; }

      auto oValue = getNumericCellValue(oCell.value());
      if ( ! oValue.has_value())   { return {}; }

      return (oValue.value() < 0);
}
```

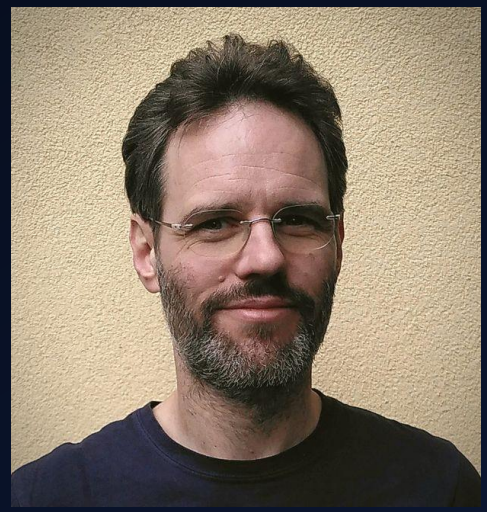
# C++23: Begone, Boilerplate!

```
std::optional<bool> isNumericTableCellValueNegative
    (const CDb& db, const Key& key, const CLocation& location)
{
    return getElement(db, key)
              .and_then(getTable)
              .and_then([location](const CTableData& table)
                        {return getCell(table, location);})
              .and_then(getNumericCellValue)
              .transform(isNegative);
}
```



## Robert Schimkowitsch

https://mastodon.social/@asperamanca
https://github.com/Asperamanca/
https://cppusergroupvienna.org/

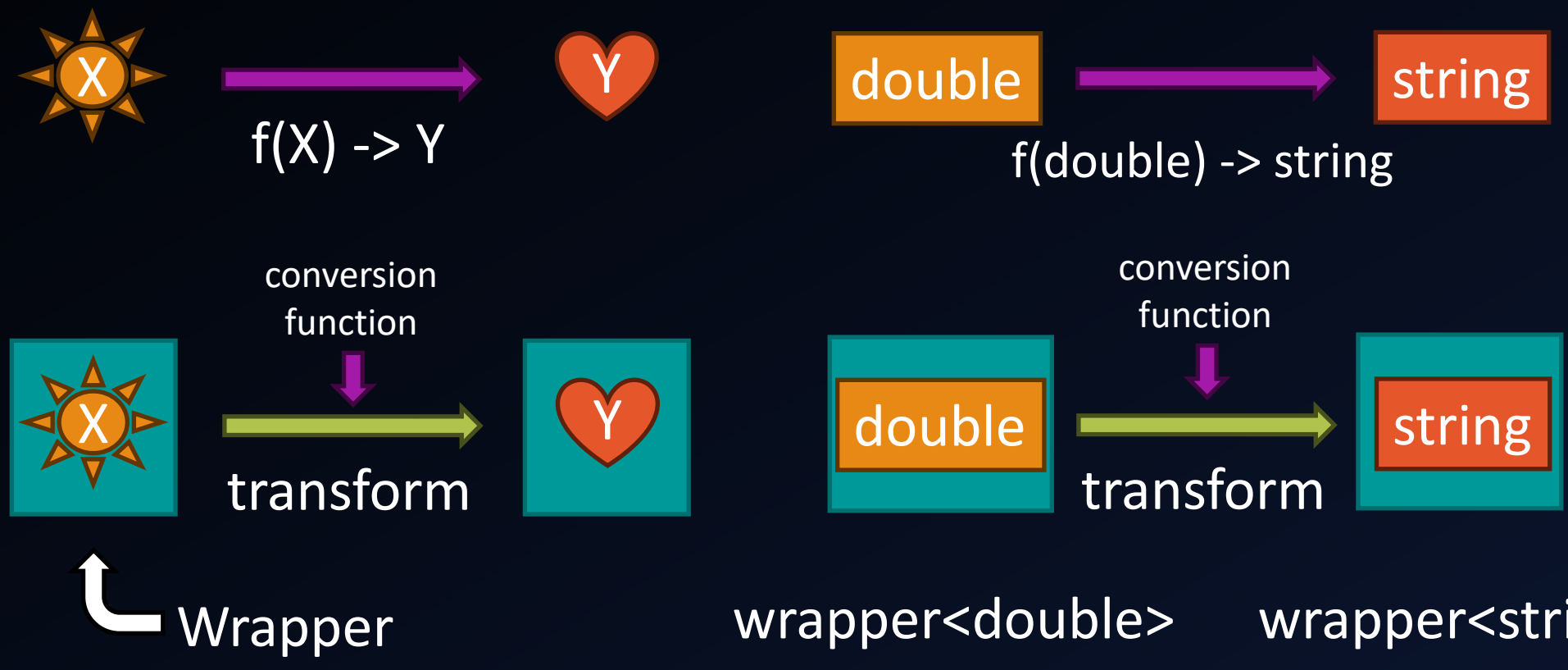
# Functors

WRAP & TRANSFORM

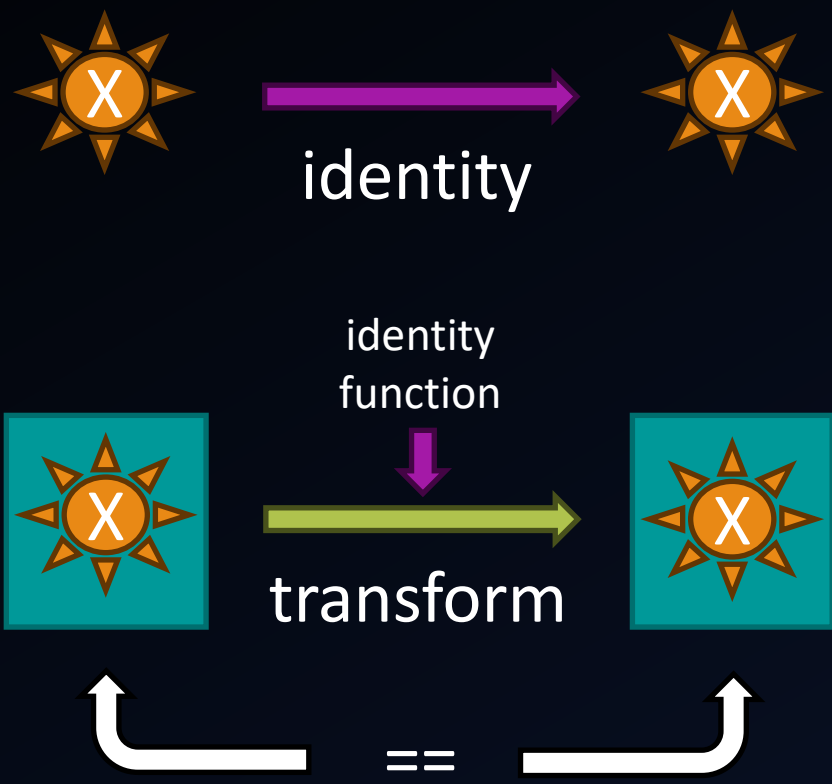
# What is a Functor?

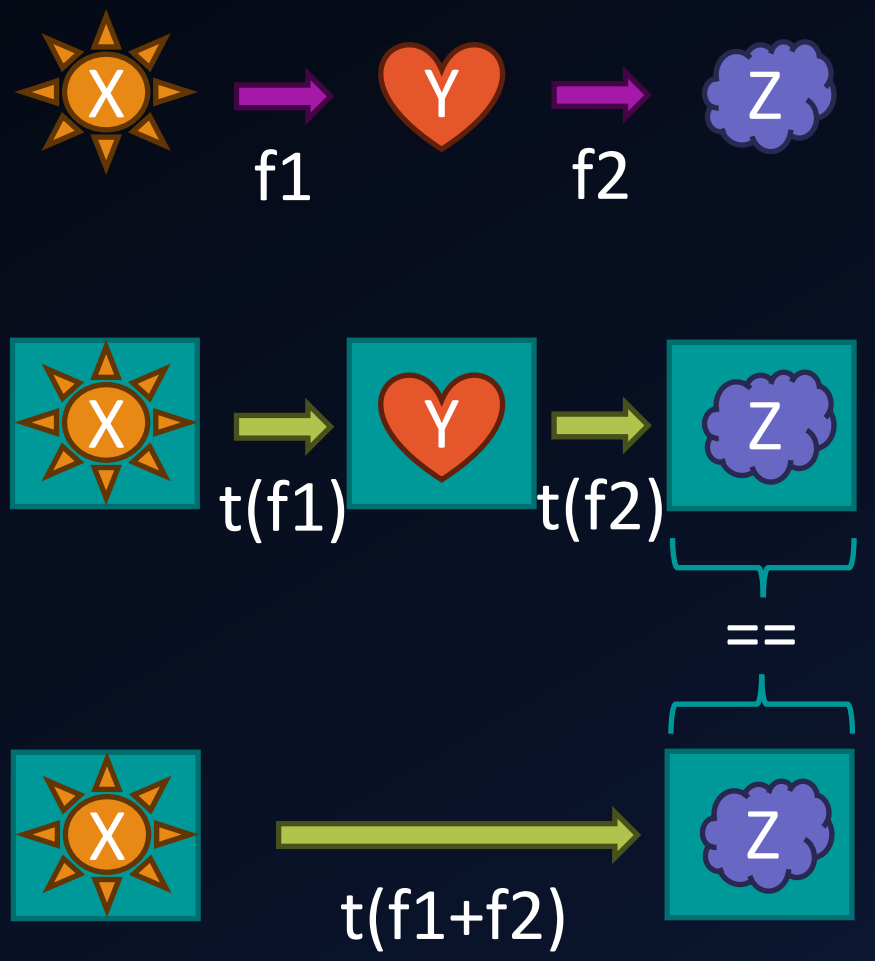
```
class CNegator
{
public:
    int operator()(const int value) const
    {
        return -value;
    }
};

// ...

CNegator negator;
auto x = negator(5); // -5
```

# Now really, what is a Functor?



# Two rules for the Transformation



# Example: From Classic Loop to Functor

```
auto v = std::vector{1.5,2.0,2.5};

std::vector<std::string> strings;
for(const auto& value : v)
{
  strings.push_back(std::format("{:}",
                    std::pow(value,2.0)));
}
// 'strings' now contains {"2.25","4","6.25"}
```

# Example: From Classic Loop to Functor

```
auto v = std::vector{1.5,2.0,2.5};

std::vector<std::string> strings;
for(const auto& value : v)
{
  strings.push_back(std::format("{:}",
                    std::pow(value,2.0)));
}
// 'strings' now contains {"2.25","4","6.25"}
```

# Example: From Classic Loop to Functor

```
auto v = std::vector{1.5,2.0,2.5};

std::vector<std::string> strings;
for(const auto& value : v)
{
  strings.push_back(std::format("{:}",
                    std::pow(value,2.0)));
}
// 'strings' now contains {"2.25","4","6.25"}
```

# Example: From Classic Loop to Functor

```
auto v = std::vector{1.5,2.0,2.5};

std::vector<std::string> strings;
for(const auto& value : v)
{
  strings.push_back(std::format("{:}",
                    std::pow(value,2.0)));
}
// 'strings' now contains {"2.25","4","6.25"}
```

# Example: From Classic Loop to Functor

```
#include <format>

double makeSquared(const double value)
{
    return std::pow(value,2.0);
}

std::string dblToStr(const double value)
{
    return std::format("{:}",value);
}
```

# Example: From Classic Loop to Functor

```
auto v = std::vector{1.5,2.0,2.5};

std::vector<std::string> strings;
for(const auto& value : v)
{
  strings.push_back(dblToStr(makeSquared(value)));
}

// 'strings' now contains {"2.25","4","6.25"}
```

# Example: From Classic Loop to Functor

```
#include <format>

double makeSquared(const double value)
{
    return std::pow(value,2.0);
}

std::string dblToStr(const double value)
{
    return std::format("{:}",value);
}
```

# Example: From Classic Loop to Functor

```
#include <format>

double makeSquared(const double value)
{
    return std::pow(value,2.0);
}
```

```
std::string dblToStr(const double value)
{
    return std::format("{:}",value);
}
```

# Example: From Classic Loop to Functor

```
#include <ranges>

auto v = std::vector{1.5,2.0,2.5};

auto strings = std::views::transform
                  (std::views::transform(v,makeSquared),
                                              dblToStr);

// {"2.25", "4", "6.25"}
```

# Example: From Classic Loop to Functor

```
#include <ranges>

auto v = std::vector{1.5,2.0,2.5};

auto strings = std::views::transform
 (std::views::transform(v,makeSquared),
                        dblToStr);

// {"2.25", "4", "6.25"}
```

# Example: From Classic Loop to Functor

```
#include <ranges>

auto v = std::vector{1.5,2.0,2.5};

auto strings = std::views::transform
                (std::views::transform(v,makeSquared),
                                       dblToStr);

// {"2.25", "4", "6.25"}
```

# The Pipe to the Rescue

```
#include <ranges>

auto v = std::vector{1.5,2.0,2.5};

auto strings = v
             | std::views::transform(makeSquared)
             | std::views::transform(dblToStr);

// {"2.25", "4", "6.25"}
```

# The Pipe to the Rescue

```
#include <ranges>

auto v = std::vector{1.5,2.0,2.5};

auto strings = v
             | std::views::transform(v,makeSquared)
             | std::views::transform(dblToStr);

// {"2.25", "4", "6.25"}
```

# Classic Solution vs. Functor Solution

```
std::vector<std::string> strings;
for(const auto& value : v)
{
  strings.push_back(dblToStr(makeSquared(value)));
}
```

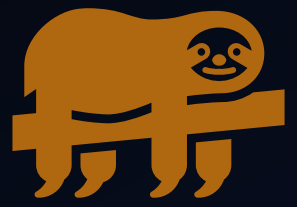
```
auto strings = v
             | std::views::transform(makeSquared)
             | std::views::transform(dblToStr);
```

View: Does not own or copy its data

# Classic Solution vs. Functor Solution

```
std::vector<std::string> strings;
for(const auto& value : v)
{
  strings.push_back(dblToStr(makeSquared(value)));
}
```

```
auto strings = v
             | std::views::transform(makeSquared)
             | std::views::transform(dblToStr);
```



# It's a Trap!

```
struct CEntry {
    int m_X{};
    int m_Y{};
    std::string m_Text{}; };

CEntry getNearestEntry(const int x) {...}

using namespace std;
auto v = vector{1,3,7};

auto strings = v
        | views::transform(getNearestEntry)
        | views::transform(&CEntry::m_Text);

printOutput(strings);
```

# It's a Trap!

```
struct CEntry {
    int m_X{};
    int m_Y{};
    std::string m_Text{}; };

CEntry getNearestEntry(const int x) {...}

using namespace std;
auto v = vector{1,3,7};

auto strings = v
        | views::transform(getNearestEntry)
        | views::transform(&CEntry::m_Text);

printOutput(strings);
```

# It's a Trap!

```
struct CEntry {
    int m_X{};
    int m_Y{};
    std::string m_Text{}; };

CEntry getNearestEntry(const int x) {...}

using namespace std;
auto v = vector{1,3,7};

auto strings = v
        | views::transform(getNearestEntry)
        | views::transform(&CEntry::m_Text);

printOutput(strings);
```

# The Type of the Resulting View

```
class std::ranges::transform_view
 <class std::ranges::transform_view
   <class std::ranges::ref_view
     <class std::vector
       <double,class std::allocator<double>>
     >,double (__cdecl*)(double)
   >,
  class std::basic_string
    <char,struct std::char_traits<char>,
     class std::allocator<char>
    > (__cdecl*)(double)
  >
```

auto

# Converting the output of views into a container

```
std::vector<std::string> convertViewResultToContainer()
{
   using namespace std;
   auto v = vector{1.5,2.0,2.5};

   auto strings = v
      | views::transform(makeSquared)
      | views::transform(dblToStr);

   return ranges::to<vector<string>>(strings);
}
```

# Inlining functions that return views

```
inline auto getSquaredNumerbersAsString
   (const std::vector<int>& vector)
{
   using namespace std;

   auto strings = v
      | views::transform(makeSquared)
      | views::transform(dblToStr);

   return strings;
}
```

# Callables

## DIFFERENT WAYS TO FEED YOUR FUNCTOR (OR MONAD)

# Callables: Free functions

```
double makeSquared(const double value) {...}

auto v = vector{1.5,2.0,2.5};
auto strings = v
   | std::views::transform(makeSquared)
   | std::views::transform(dblToStr);
```

# Callables: (Class) Static Functions

```
class CConv
{
public:
   static double makeSquaredStatic(const double value);
   //...
};
//...

auto v = vector{1.5,2.0,2.5};
auto strings = v
   | std::views::transform(CConv::makeSquaredStatic)
   | //...
```

# Callables: Inline Lambda

```
using namespace std;

auto v = vector{1.5,2.0,2.5};
auto strings = v
   | views::transform([](const double& value)
      {return value * value;})
   | //...
```

# Callables: Pick overload with Inline Lambda

```
double makeSquared(const double value) {...}
int makeSquared(const int value) {...}


auto v = vector{1.5,2.0,2.5};
auto strings = v
   | std::views::transform([](auto&& val)
                           {return makeSquared(val);})
   | std::views::transform(dblToStr);
```

# Callables: Inject Parameters via Inline Lambda

```
using namespace std;

const double power = 2.0;
auto v = vector{1.5,2.0,2.5};
auto strings = v
   | views::transform([power](const double& value)
      {return std::pow(value, power);})
   | //...
```

# Callables: Member Functions with Lambda

```
struct CValue
{
   double getValue() const;
   //...
};

auto strings = std::vector{CValue{1.5},CValue{2.0}}
   | std::views::transform(
      [](const CValue& obj){return obj.getValue();})
   | std::views::transform(makeSquared)
   | //...
```

# Callables: Member Functions from Params (Lambda)

```
class CConv
{
public:
   double makeSquared(const double value);
};
//...

using namespace std;
CConv conv;
auto v = vector{1.5,2.0,2.5};
auto strings = v
   | views::transform([&conv](const double& value)
      {return conv.makeSquaredMember(value);})
   | //...
```

# Callables: Named Lambda

```
auto makeSquaredLocalLambda = [](const double value)
   {
      return std::pow(value,2.0);
   };

auto v = vector{1.5,2.0,2.5};
auto strings = v
   | std::views::transform(makeSquaredLocalLambda)
   | //...
```

# Callables: Function Object

```
struct CMakeSquared
{
   double operator()(const double value) const;
};

CMakeSquared makeSquaredFunctionObject;
auto v = vector{1.5,2.0,2.5};
auto strings = v
   | std::views::transform(makeSquaredFunctionObject)
   | //...
```

# Callables: More options - Adapter functions

| Function | Use for | Example |
|---|---|---|
| std::mem_fn | Class member functions | `auto f = std::mem_fn(&CClass::foo);`<br>`f(pInstance, arg1, arg2);` |
| std::bind_front | Adapt functions with too many arguments | `void foo(int arg1, std::string arg2);`<br>`auto f = std::bind_front(&foo, 42);`<br>`f("OnlyStringNeeded");` |
| std::bind_back | | `void foo(int arg1, std::string arg2);`<br>`auto f = std::bind_back(&foo, "String");`<br>`f(123); // Only int needed` |
| std::bind | Not recommended | |

Full examples in the talk material.
See link at the end of the talk!

# Callables: std::function

```
#include <functional>

std::function<double(double)> fAnyFuncDblInDblRet
   = makeSquared;

//...
// Function could be a passed parameter, adding flexibility

auto v = vector{1.5,2.0,2.5};
auto strings = v
   | std::views::transform(fAnyFuncDblInDblRet)
   | //...
```

# Callables: Template Parameter

```
template<class TMakeSquared>

void foo(const TMakeSquared& fMakeSquaredTemplate)
{
   auto v = vector{1.5,2.0,2.5};
   auto strings = v
      | std::views::transform(fMakeSquaredTemplate)
      | //...
}
```

# Callables: Template Parameter

```
template<class TMakeSquared>
   requires std::invocable<TMakeSquared,double>
void foo(const TMakeSquared& fMakeSquaredTemplate)
{
   auto v = vector{1.5,2.0,2.5};
   auto strings = v
      | std::views::transform(fMakeSquaredTemplate)
      | //...
}
```

# Callables: Summary

Inline Code

```
[](const double& value)
      {return value * value;}
```

Extra Param

```
[power](const double& value)
   {return std::pow(value, power);})
```

Member from instance

```
[](const CValue& obj){return obj.getValue();})
```

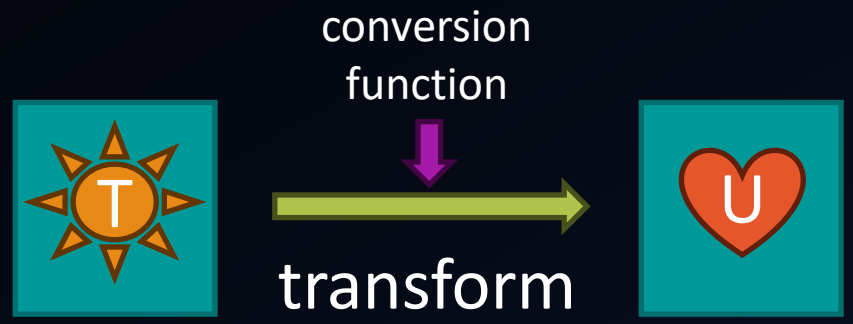
Member with param

```
[&conv](const double& value)
    {return conv.makeSquaredMember(value);}
```

Inject callable

```
std::function<double(double)> fMakeSquared
   = makeSquared;
```

# Functors: Summary



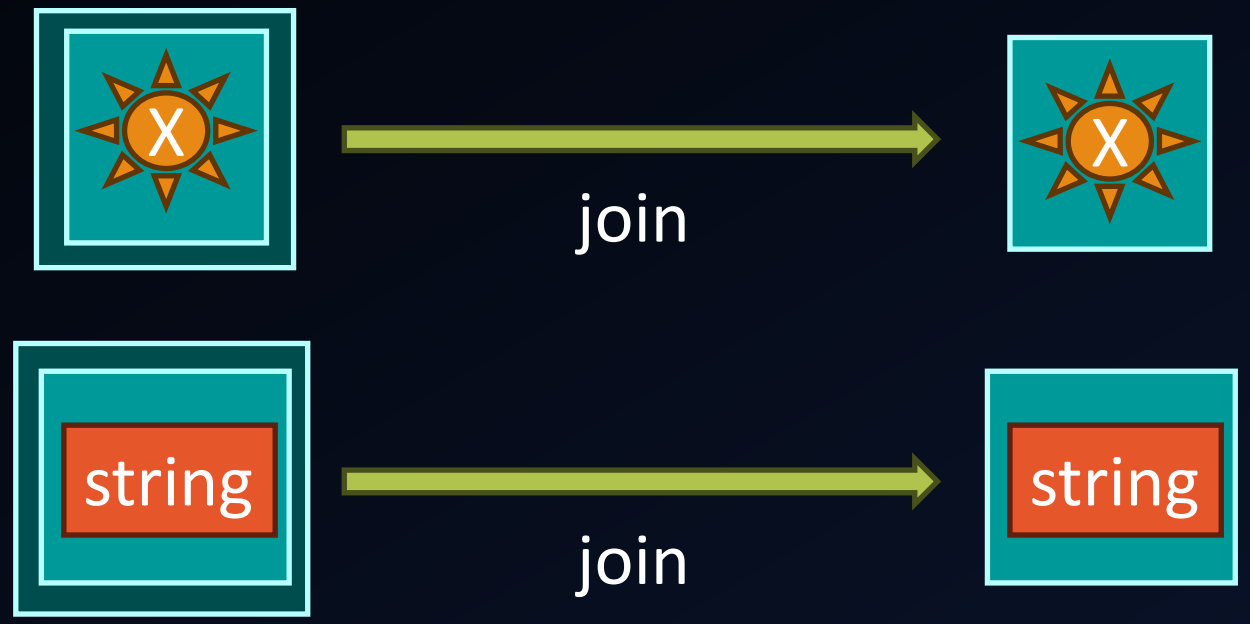

```
auto strings = v
             | std::views::transform(makeSquared)
             | std::views::transform(dblToStr);
```

Functions, Class Instances, Lambdas,
std::function, Template parameter,...

# Monads

## FUNCTORS+JOIN

# Monad = Functor + Join




wrapper<wrapper<string>>

wrapper<string>

# Usage of Join


# Printing Diagnostics: Conversion Functions

```
void printDiagnostic(const CDiagnosic& info);
```

```
std::vector<CDiagnosic> compile(const CFile& input);
```

```
std::vector<CFile> getFilesInProject(const CProject& input);
```

# Printing Diagnostics: Conversion Functions

```
void printDiagnostic(const CDiagnosic& info);

std::vector<CDiagnosic> compile(const CFile& input);

std::vector<CFile> getFilesInProject(const CProject& input);
```

# Printing Diagnostics: Conversion Functions

```
void printDiagnostic(const CDiagnosic& info);

std::vector<CDiagnosic> compile(const CFile& input);

std::vector<CFile> getFilesInProject(const CProject& input);
```

# Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
   auto files = getFilesInProject(project);

   for(const auto& file : files)
   {
      auto diagnostics = compile(file);
      for(const auto& diagnostic : diagnostics)
      {
         printDiagnostic(diagnostic);
      }
   }
}
```

# Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
   auto files = getFilesInProject(project);

   for(const auto& file : files)
   {
      auto diagnostics = compile(file);
      for(const auto& diagnostic : diagnostics)
      {
         printDiagnostic(diagnostic);
      }
   }
}
```

# Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
   auto files = getFilesInProject(project);

   for(const auto& file : files)
   {
      auto diagnostics = compile(file);
      for(const auto& diagnostic : diagnostics)
      {
         printDiagnostic(diagnostic);
      }
   }
}
```

# Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
   auto files = getFilesInProject(project);

   for(const auto& file : files)
   {
      auto diagnostics = compile(file);
      for(const auto& diagnostic : diagnostics)
      {
         printDiagnostic(diagnostic);
      }
   }
}
```

# Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
   auto files = getFilesInProject(project);

   for(const auto& file : files)
   {
      auto diagnostics = compile(file);
      for(const auto& diagnostic : diagnostics)
      {
         printDiagnostic(diagnostic);
      }
   }
}
```

# Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
   auto files = getFilesInProject(project);

   for(const auto& file : files)
   {
      auto diagnostics = compile(file);
      for(const auto& diagnostic : diagnostics)
      {
         printDiagnostic(diagnostic);
      }
   }
}
```

# Printing Diagnostics: Range Monad

```
namespace vw = std::views;
auto diagnostics = projects
     | vw::transform(getFilesInProject)  | vw::join
     | vw::transform(compile)            | vw::join;

std::ranges::for_each(diagnostics,printDiagnostic);
```

# Printing Diagnostics: Range Monad

```
namespace vw = std::views;
auto diagnostics = projects
     | vw::transform(getFilesInProject) | vw::join
     | vw::transform(compile)           | vw::join;

std::ranges::for_each(diagnostics,printDiagnostic);
```

# Printing Diagnostics: Range Monad

```
namespace vw = std::views;
auto diagnostics = projects
     | vw::transform(getFilesInProject) | vw::join
     | vw::transform(compile)           | vw::join;

std::ranges::for_each(diagnostics,printDiagnostic);
```

# Printing Diagnostics: Range Monad

```
namespace vw = std::views;
auto diagnostics = projects
     | vw::transform(getFilesInProject) | vw::join
     | vw::transform(compile)           | vw::join;

std::ranges::for_each(diagnostics,printDiagnostic);
```

# Printing Diagnostics: Range Monad

```
namespace vw = std::views;
auto diagnostics = projects
     | vw::transform(getFilesInProject) | vw::join
     | vw::transform(compile)           | vw::join;

std::ranges::for_each(diagnostics,printDiagnostic);
```

# Printing Diagnostics: Range Monad

```
namespace vw = std::views;
auto diagnostics = projects
     | vw::transform(getFilesInProject)  | vw::join
     | vw::transform(compile)            | vw::join;

std::ranges::for_each(diagnostics,printDiagnostic);
```

# Printing Diagnostics: Range Monad

```
namespace vw = std::views;
auto diagnostics = projects
    | vw::transform(getFilesInProject) | vw::join
    | vw::transform(compile)           | vw::join;
```

```
std::ranges::for_each(diagnostics,printDiagnostic);
```

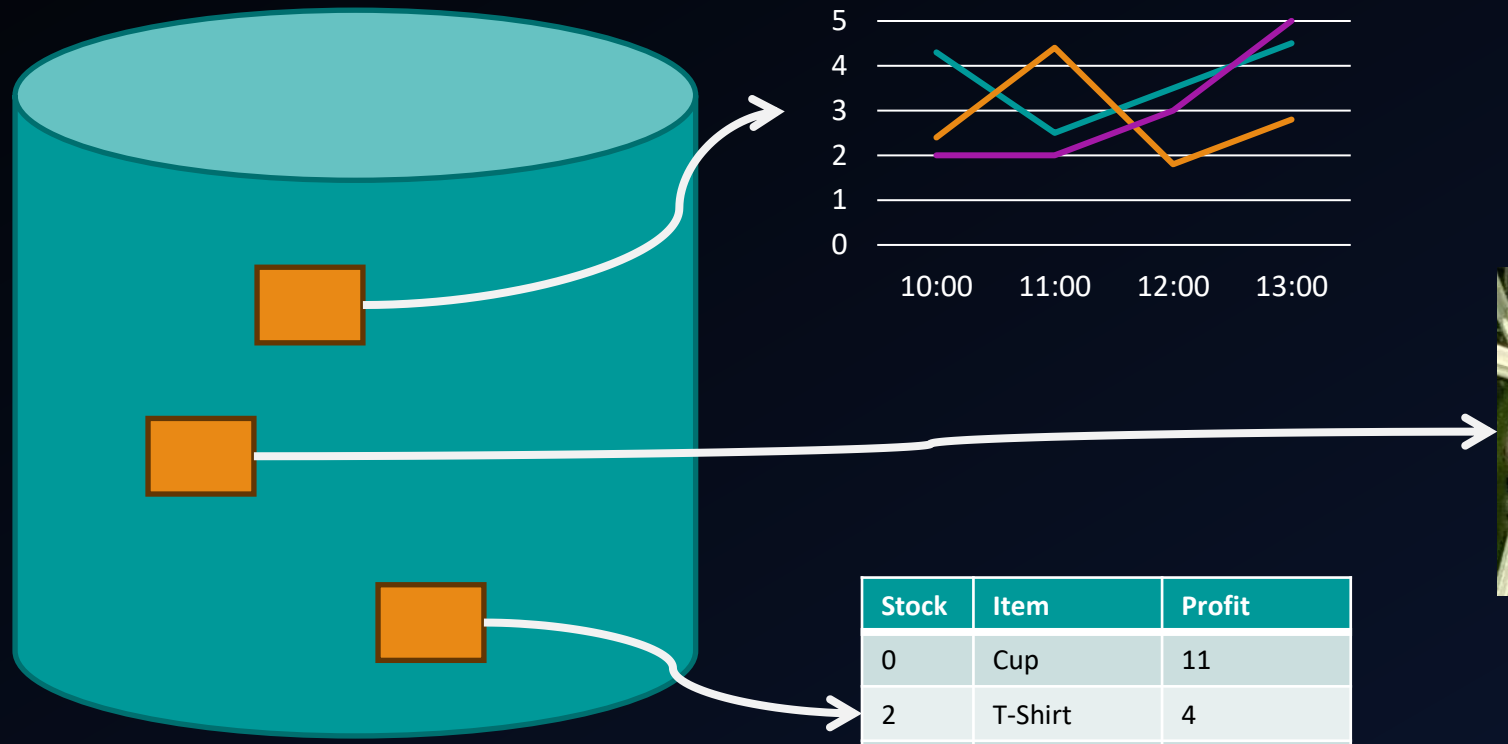
# Printing Diagnostics: Code comparison

```
for(const auto& project : projects)
{
    auto files = getFilesInProject(project);
    for(const auto& file : files)
    {
        auto diagnostics = compile(file);
        for(const auto& diagnostic : diagnostics)
        {
            printDiagnostic(diagnostic);
        }
    }
}

namespace vw = std::views;
auto diagnostics = projects
          | vw::transform(getFilesInProject) | vw::join
          | vw::transform(compile)           | vw::join;
std::ranges::for_each(diagnostics,printDiagnostic);
```

# Handling Failure

## THE OPTIONAL AND EXPECTED MONADS

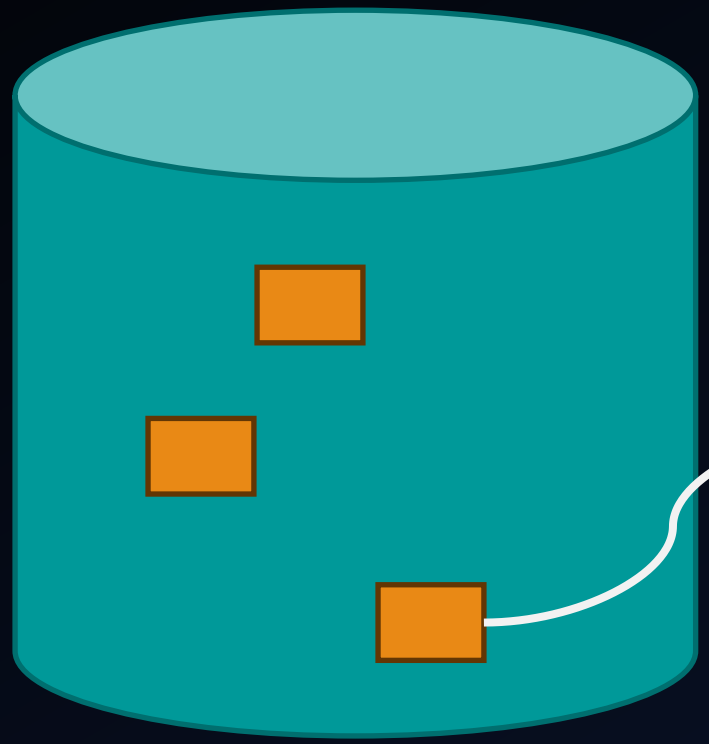
# Chaining Functions That Can Fail





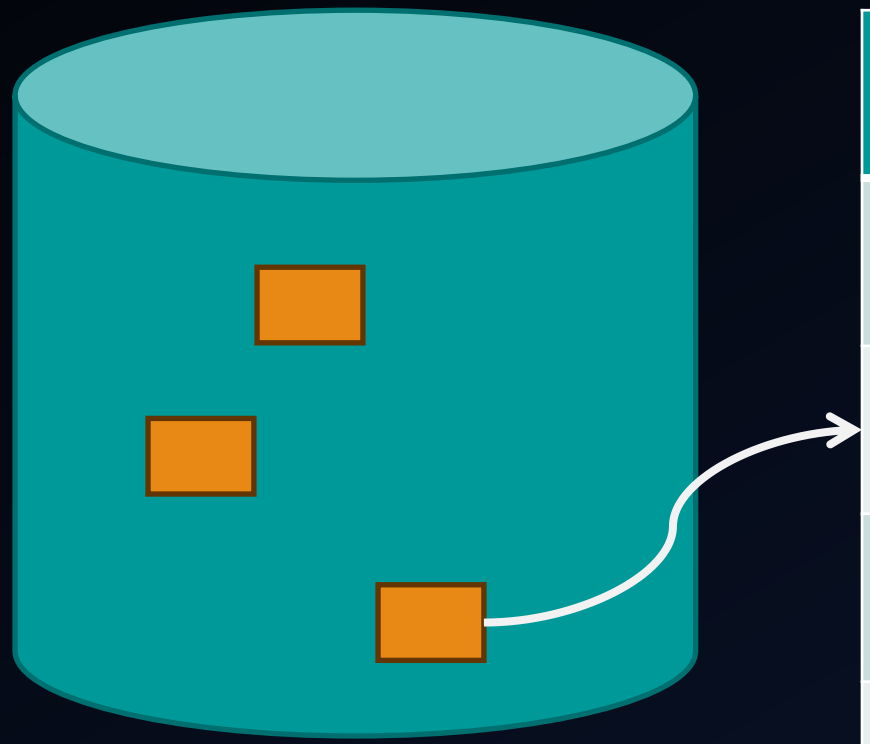

| Stock | Item | Profit |
|---|---|---|
| 0 | Cup | 11 |
| 2 | T-Shirt | 4 |
| 0 | Poster | 16 |
| 0 | Statuette | -8 |

# Chaining Functions That Can Fail



| Stock | Item | Profit |
|---|---|---|
| 0 | Cup | 11 |
| 2 | T-Shirt | 4 |
| 0 | Poster | 16 |
| 0 | Statuette | -8 |

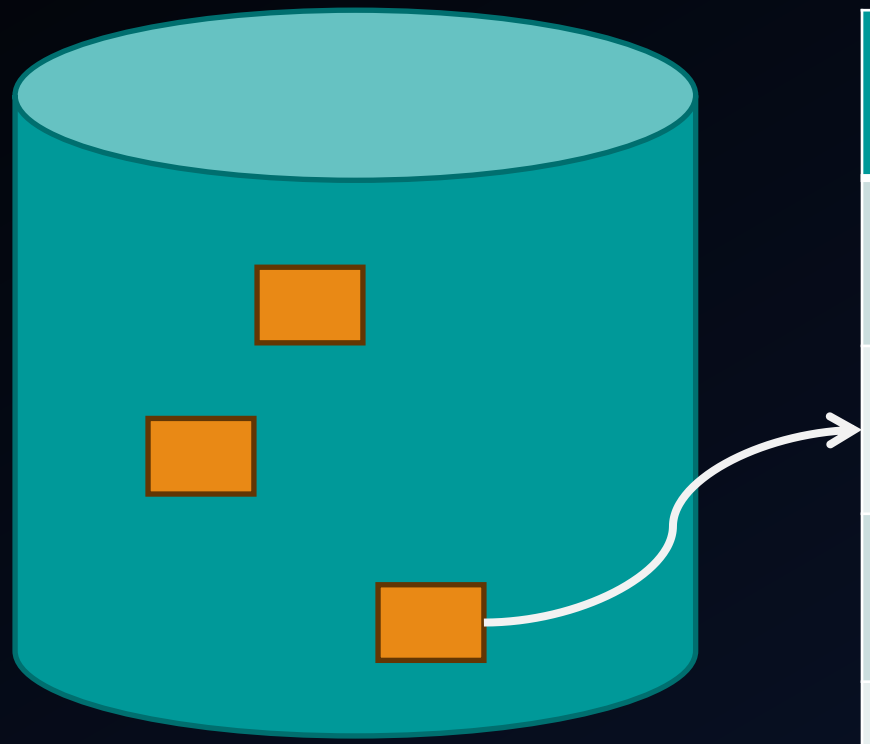
# Chaining Functions That Can Fail



| Stock | Item | Profit |
|---|---|---|
| 0 | Cup | |
| 2 | T-Shirt | |
| 0 | Poster | |
| 0 | Statuette | -8 |

Number?

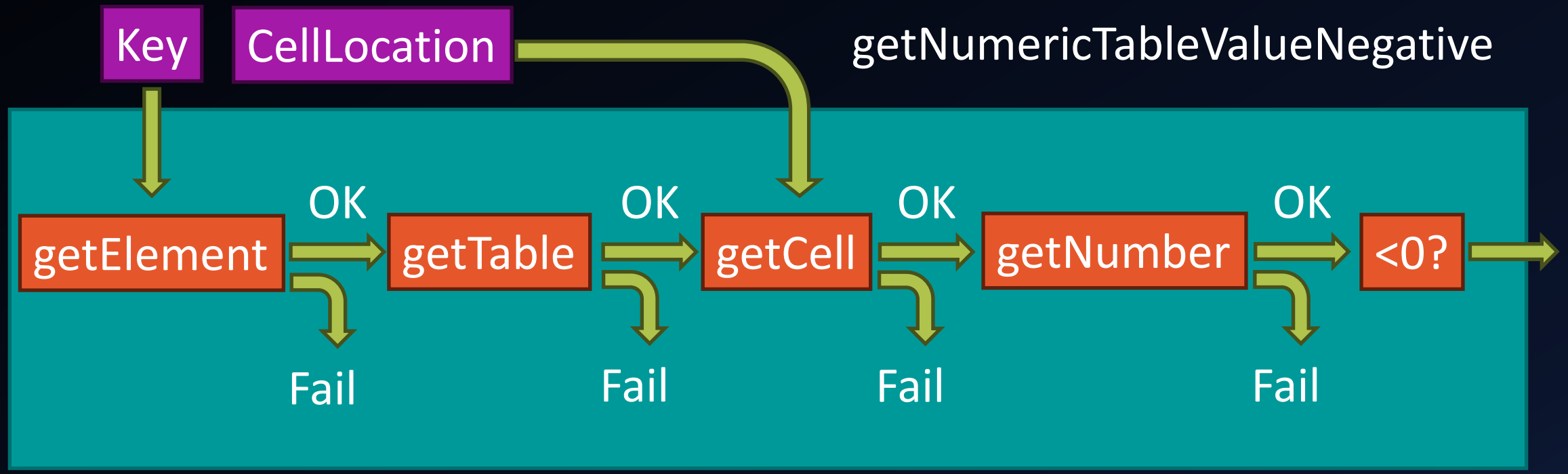
# Chaining Functions That Can Fail



| Stock | Item | Profit |
|---|---|---|
| 0 | Cup | |
| 2 | T-Shirt | |
| 0 | Poster | |
| 0 | Statuette | -8 |

Negative?

# Chaining Functions That Can Fail



# Chaining Functions That Can Fail: Classic Way

```
bool getElement(const CDb& db, const ElementKey& key,
                CElement& out);

bool getTable(const CElement& element, CTable& out);

bool getCell(const CTable& table,
             const CLocation& location CCell& out);

bool getNumericCellValue(const CCell& cell, int& out)
```

# Chaining Functions That Can Fail: Classic Way

```
bool getElement(const CDb& db, const ElementKey& key,
                CElement& out);

bool getTable(const CElement& element, CTable& out);

bool getCell(const CTable& table,
             const CLocation& location CCell& out);

bool getNumericCellValue(const CCell& cell, int& out)
```

# Chaining Functions That Can Fail: Classic Way

```
bool getElement(const CDb& db, const ElementKey& key,
                CElement& out);

bool getTable(const CElement& element, CTable& out);

bool getCell(const CTable& table,
             const CLocation& location CCell& out);

bool getNumericCellValue(const CCell& cell, int& out)
```

# Chaining Functions That Can Fail: Classic Way

```
bool getElement(const CDb& db, const ElementKey& key,
                CElement& out);

bool getTable(const CElement& element, CTable& out);

bool getCell(const CTable& table,
             const CLocation& location CCell& out);

bool getNumericCellValue(const CCell& cell, int& out)
```

# Chaining Functions That Can Fail: Classic Way

```
bool getNumericTableValueNegative
    (const CDb& db, const Key& key,
     const CLocation& location, bool& out)
{
    CElement element;
    if ( ! getElement(db, key, element))          { return false; }

    CTableData table;
    if ( ! getTable(element, table))              { return false; }

    CTableCell cell;
    if ( ! getCell(table, location, cell))        { return false; }

    int value;
    if ( ! getNumericCellValue(cell, value))      { return false; }

    result = (value < 0);
    return true;
}
```

# Chaining Functions That Can Fail: Classic Way

```
bool getNumericTableValueNegative
    (const CDb& db, const Key& key,
     const CLocation& location, bool& out)
{
    CElement element;
    if ( ! getElement(db, key, element))          { return false; }

    CTableData table;
    if ( ! getTable(element, table))              { return false; }

    CTableCell cell;
    if ( ! getCell(table, location, cell))        { return false; }

    int value;
    if ( ! getNumericCellValue(cell, value))      { return false; }

    result = (value < 0);
    return true;
}
```

# Chaining Functions That Can Fail: Classic Way

```
bool getNumericTableValueNegative
    (const CDb& db, const Key& key,
     const CLocation& location, bool& out)
{
    CElement element;
    if ( ! getElement(db, key, element))          { return false; }

    CTableData table;
    if ( ! getTable(element, table))              { return false; }

    CTableCell cell;
    if ( ! getCell(table, location, cell))        { return false; }

    int value;
    if ( ! getNumericCellValue(cell, value))      { return false; }

    result = (value < 0);
    return true;
}
```

# Chaining Functions That Can Fail: Classic Way

```
bool getNumericTableValueNegative
    (const CDb& db, const Key& key,
     const CLocation& location, bool& out)
{
    CElement element;
    if ( ! getElement(db, key, element))          { return false; }

    CTableData table;
    if ( ! getTable(element, table))              { return false; }

    CTableCell cell;
    if ( ! getCell(table, location, cell))        { return false; }

    int value;
    if ( ! getNumericCellValue(cell, value))      { return false; }

    result = (value < 0);
    return true;
}
```

# Chaining Functions That Can Fail: Classic Way

```
bool getNumericTableValueNegative
    (const CDb& db, const Key& key,
     const CLocation& location, bool& out)
{
    CElement element;
    if ( ! getElement(db, key, element))          { return false; }

    CTableData table;
    if ( ! getTable(element, table))              { return false; }

    CTableCell cell;
    if ( ! getCell(table, location, cell))        { return false; }

    int value;
    if ( ! getNumericCellValue(cell, value))      { return false; }

    result = (value < 0);
    return true;
}
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<CElement> getElement(const CDb& db,
                                   const ElementKey& key);

std::optional<CTable> getTable(const CElement& element);

std::optional<CCell> getCell(const CTable& tableData,
                             const CLocation& location);

std::optional<int> getNumericCellValue(const CCell& cell);

bool isNegative(const int value);
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<CElement> getElement(const CDb& db,
                                   const ElementKey& key)
{
    //...
    if (/* Key not found */)
    {
        return{}; // or: return std::nullopt;
    }
    CElement elem = //...
    return elem;
}
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<CElement> getElement(const CDb& db,
                                   const ElementKey& key)
{
    //...
    if (/* Key not found */)
    {
        return{}; // or: return std::nullopt;
    }
    CElement elem = //...
    return elem;
}
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<CElement> getElement(const CDb& db,
                                   const ElementKey& key);

std::optional<CTable> getTable(const CElement& element);

std::optional<CCell> getCell(const CTable& tableData,
                             const CLocation& location);

std::optional<int> getNumericCellValue(const CCell& cell);

bool isNegative(const int value);
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<CElement> getElement(const CDb& db,
                                   const ElementKey& key);

std::optional<CTable> getTable(const CElement& element);

std::optional<CCell> getCell(const CTable& tableData,
                             const CLocation& location);

std::optional<int> getNumericCellValue(const CCell& cell);

bool isNegative(const int value);
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<CElement> getElement(const CDb& db,
                                   const ElementKey& key);

std::optional<CTable> getTable(const CElement& element);

std::optional<CCell> getCell(const CTable& tableData,
                             const CLocation& location);

std::optional<int> getNumericCellValue(const CCell& cell);

bool isNegative(const int value);
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<CElement> getElement(const CDb& db,
                                   const ElementKey& key);

std::optional<CTable> getTable(const CElement& element);

std::optional<CCell> getCell(const CTable& tableData,
                             const CLocation& location);

std::optional<int> getNumericCellValue(const CCell& cell);

bool isNegative(const int value);
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<bool> isNumericTableCellValueNegative
  (const CDb& db, const Key& key, const CLocation& location)
{
    return getElement(db, key)
              .and_then(getTable)
              .and_then([location](const CTableData& table)
                        {return getCell(table, location);})
              .and_then(getNumericCellValue)
              .transform(isNegative);
}
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<bool> isNumericTableCellValueNegative
  (const CDb& db, const Key& key, const CLocation& location)
{
    return getElement(db, key)
             .and_then(getTable)
             .and_then([location](const CTableData& table)
                       {return getCell(table, location);})
             .and_then(getNumericCellValue)
             .transform(isNegative);
}
```

# Quiz Time! – What makes this a monad?

```
getElement(...) -> std::optional<CElement>
```

```
getElement(db, key)
    .and_then(getTable)
    //...
```

```
getTable(std::optional<CElement>) -> std::optional<CTable>
```

```
auto wrapped = std::optional<std::optional<int>>{5};

std::optional<int> unwrapped =
    wrapped.and_then(std::identity{});
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<bool> isNumericTableCellValueNegative
  (const CDb& db, const Key& key, const CLocation& location)
{
    return getElement(db, key)
              .and_then(getTable)
              .and_then([location](const CTableData& table)
                        {return getCell(table, location);})
              .and_then(getNumericCellValue)
              .transform(isNegative);
}
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<bool> isNumericTableCellValueNegative
  (const CDb& db, const Key& key, const CLocation& location)
{
    return getElement(db, key)
             .and_then(getTable)
             .and_then([location](const CTableData& table)
                       {return getCell(table, location);})
             .and_then(getNumericCellValue)
             .transform(isNegative);
}
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<bool> isNumericTableCellValueNegative
  (const CDb& db, const Key& key, const CLocation& location)
{
    return getElement(db, key)
             .and_then(getTable)
             .and_then([location](const CTableData& table)
                       {return getCell(table, location);})
             .and_then(getNumericCellValue)
             .transform(isNegative);
}
```

# What you can do with an std::optional

```
std::optional<TRet> result = foo();

if (result.has_value())
{
    auto value = result.value();
    //...
}
else
{
    //...
}
```

# What you can do with an std::optional

```
std::optional<TRet> result = foo();

if (result.has_value())
{
    auto value = result.value();
    //...
}
else
{
    //...
}
```

# Chaining Functions That Can Fail: Code Comparison

```
CElement element;
if ( ! getElement(db, key, element))                { return false; }
CTableData table;
if ( ! getTable(element, table))                    { return false; }
CTableCell cell;
if ( ! getCell(table, location, cell))              { return false; }
int value;
if ( ! getNumericCellValue(cell, value))            { return false; }
result = (value < 0); return true;
```

```
return getElement(db, key)
    .and_then(getTable)
    .and_then([location](const CTableData& table)
              {return getCell(table, location);})
    .and_then(getNumericCellValue)
    .transform(isNegative);
```

# Chaining Functions That Can Fail: The Optional Monad

```
std::optional<bool> isNumericTableCellValueNegative
  (const CDb& db, const Key& key, const CLocation& location)
{
    return getElement(db, key)
             .and_then(getTable)
             .and_then([location](const CTableData& table)
                       {return getCell(table, location);})
             .and_then(getNumericCellValue)
             .transform(isNegative)
           .or_else(log<bool>);
}
```

# Chaining Functions That Can Fail: The Optional Monad

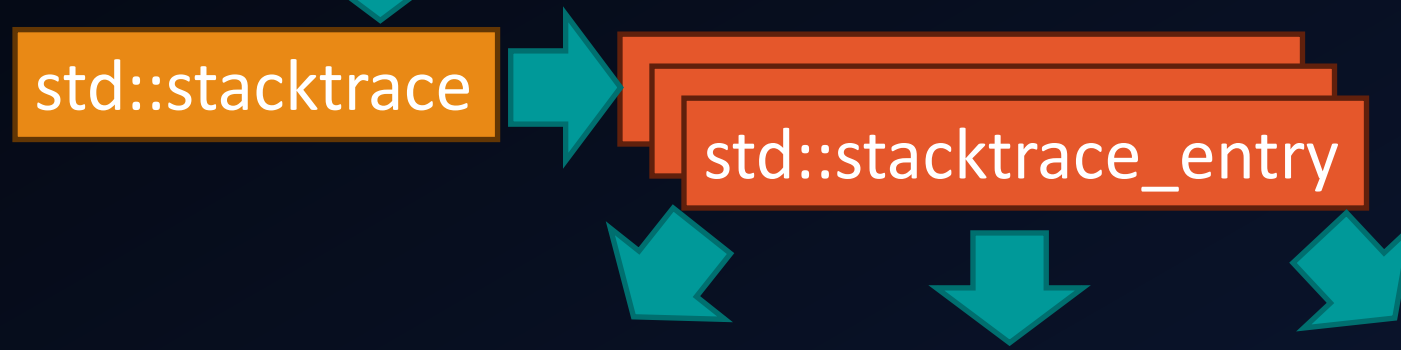
```
std::optional<bool> isNumericTableCellValueNegative
  (const CDb& db, const Key& key, const CLocation& location)
{
    return getElement(db, key)
             .and_then(getTable)
             .and_then([location](const CTableData& table)
                       {return getCell(table, location);})
             .and_then(getNumericCellValue)
             .transform(isNegative)
           .or_else(log<bool>);
}
template<class TRet>
std::optional<TRet> log();
```

# Catching failure: or_else

```
template<class TRet>
std::optional<TRet> COptionalMonad::log()
{
    std::println("Error:");
    dumpStack(std::stacktrace::current());
    return std::nullopt;
}
```

# Catching failure: or_else

```
template<class TRet>
std::optional<TRet> COptionalMonad::log()
{
    std::println("Error at:");
    dumpStack(std::stacktrace::current());
    return std::nul pt;
}
```

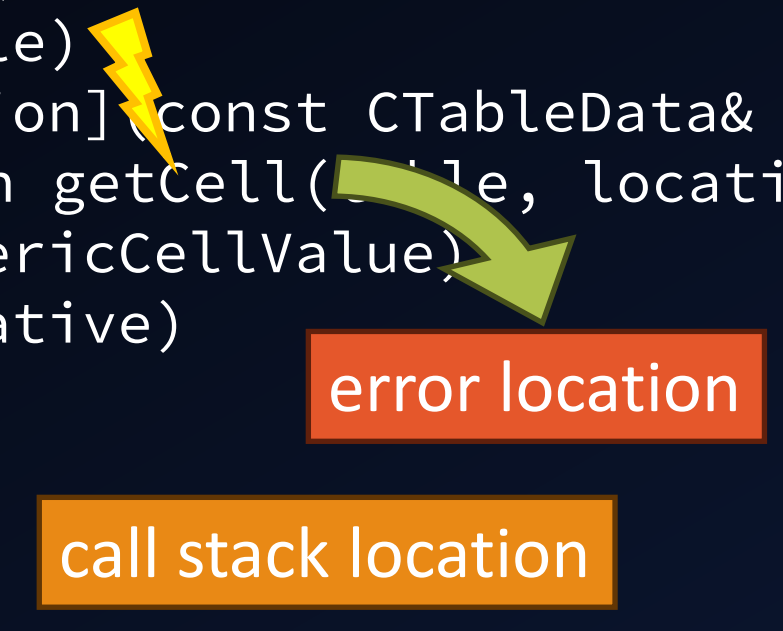


# Catching failure: or_else

```
template<class TRet>
std::optional<TRet> COptionalMonad::log()
{
    std::println("Error:");
    dumpStack(std::stacktrace::current());
    return std::nullopt;
}
```

# Catching failure: Lack of error context

```
std::optional<bool> isNumericTableCellValueNegative
  (const CDb& db, const Key& key, const CLocation& location)
{
    return getElement(db, key)
             .and_then(getTable)
             .and_then([location](const CTableData& table)
                       {return getCell(table, location);})
             .and_then(getNumericCellValue)
             .transform(isNegative)
           .or_else(log<bool>);
}
```



# Returning Error State: The Expected Monad

```
std::optional<CElement> getElement
    (const CDb& db,const ElementKey& key);

std::optional<CTable> getTable
    (const CElement& element);

std::optional<CCell> getCell
    (const CTable& tableData, const CLocation& location);

std::optional<int> getNumericCellValue
    (const CCell& cell);

bool isNegative(const int value);
```

# Returning Error State: The Expected Monad

```
std::expected<CElement,CErr> getElement
    (const CDb& db,const ElementKey& key);

std::expected<CTable,CErr> getTable
    (const CElement& element);

std::expected<CCell,CErr> getCell
    (const CTable& tableData, const CLocation& location);

std::expected<int,CErr> getNumericCellValue
    (const CCell& cell);

bool isNegative(const int value);
```

# Returning Error State: The Expected Monad

```
std::expected<CElement,CErr> getElement(const CDb& db,
                                        const ElementKey& key)
{
    //...
    if (/* Key not found */)
    {
        return std::unexpected{CErr("Key not found")};
    }
    CElement elem = //...
    return elem;
}
```

# Returning Error State: The Expected Monad

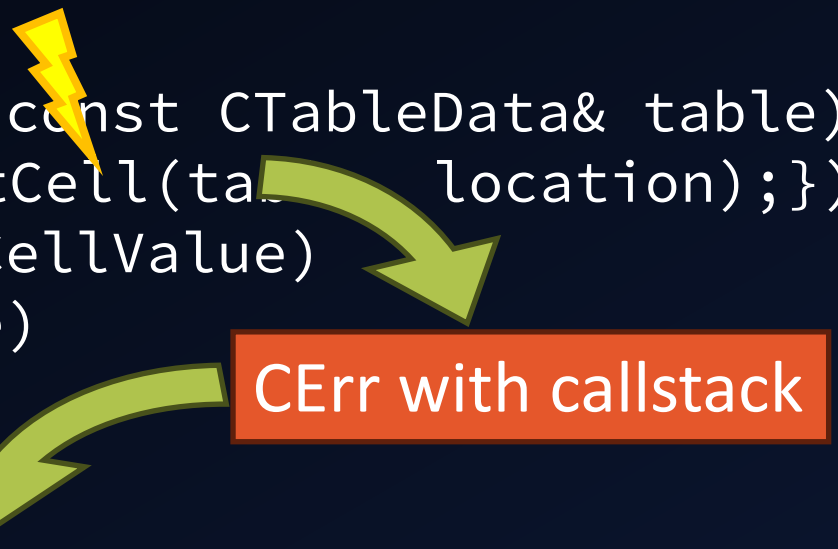
```
std::expected<CElement,CErr> getElement(const CDb& db,
                                        const ElementKey& key)
{
    //...
    if (/* Key not found */)
    {
        return std::unexpected{CErr("Key not found")};
    }
    CElement elem = //...
    return elem;
}
```

# Returning Error State: The Expected Monad

```
std::expected<bool> isNumericTableCellValueNegative
  (const CDb& db, const Key& key, const CLocation& location)
{
    return getElement(db, key)
              .and_then(getTable)
              .and_then([location](const CTableData& table)
                        {return getCell(table, location);})
              .and_then(getNumericCellValue)
              .transform(isNegative);
}
```

# Returning Error State: The Expected Monad

```
std::expected<bool, CErr> isNumericTableCellValueNegative
  (const CDb& db, const Key& key, const CLocation& location)
{
    return getElement(db, key)
             .and_then(getTable)
             .and_then([location](const CTableData& table)
                       {return getCell(ta      location);})
             .and_then(getNumericCellValue)
             .transform(isNegative)
           .or_else(log<bool>);
}
template<class TRet>
std::expected<TRet> log(const CErr& errorInfo);
```



# Returning Error State: The Expected Monad

```
template<class TRet>
std::expected<TRet, CErr> CExpectedMonad::log
                                (const CErr& errorInfo)
{
    std::println("Error: {:}", errorInfo.m_Message);
    dumpStack(errorInfo.m_Stack);
    return std::unexpected{errorInfo};
}
```

# Returning Error State: The Expected Monad

```
template<class TRet>
std::expected<TRet, CErr> CExpectedMonad::log
                                (const CErr& errorInfo)
{
    std::println("Error: {:}", errorInfo.m_Message);
    dumpStack(errorInfo.m_Stack);
    return std::unexpected{errorInfo};
}
```

# Returning Error State: The Expected Monad

```
std::expected<TRet, CErr> result = foo();

if (result.has_value())
{
    auto value = result.value();
    //...
}
else
{
    auto errorInfo = result.error();
    //...
}
```

# Returning Error State: The Expected Monad

```
std::expected<TRet, CErr> result = foo();

if (result.has_value())
{
    auto value = result.value();
    //...
}
else
{
    auto errorInfo = result.error();
    //...
}
```

# Returning Error State: The Expected Monad

```
std::expected<TRet, CErr> result = foo();

if (result.has_value())
{
    auto value = result.value();
    //...
}
else
{
    auto errorInfo = result.error();
    //...
}
```

# The Default Monad

## THE OTHER SIDE OF STD::OPTIONAL

# Picking First Success: The Default Monad

```
std::optional<ELanguage> getLanguageFromCommandLine();

std::optional<ELanguage> getLanguageFromRegistry();

std::optional<ELanguage> getLanguageFromEnvironment();

//Fallback: ELanguage::English
```

# Picking First Success: The Default Monad

```
std::optional<ELanguage> getLanguageFromCommandLine();

std::optional<ELanguage> getLanguageFromRegistry();

std::optional<ELanguage> getLanguageFromEnvironment();

//Fallback: ELanguage::English
```

# Picking First Success: The Default Monad

```
std::optional<ELanguage> getLanguageFromCommandLine();

std::optional<ELanguage> getLanguageFromRegistry();

std::optional<ELanguage> getLanguageFromEnvironment();

//Fallback: ELanguage::English
```

# Picking First Success: The Default Monad

```
std::optional<ELanguage> getLanguageFromCommandLine();

std::optional<ELanguage> getLanguageFromRegistry();

std::optional<ELanguage> getLanguageFromEnvironment();

//Fallback: ELanguage::English
```

# Picking First Success: The Default Monad

```
std::optional<ELanguage> getLanguageFromCommandLine();

std::optional<ELanguage> getLanguageFromRegistry();

std::optional<ELanguage> getLanguageFromEnvironment();

//Fallback: ELanguage::English
```

# Picking First Success: The Default Monad

```
ELanguage getStartupLanguage()
{
    return getLanguageFromCommandLine()
             .or_else(getLanguageFromRegistry)
             .or_else(getLanguageFromEnvironment)
             .value_or(ELanguage::English);
}
```

# Picking First Success: The Default Monad

```
ELanguage getStartupLanguage()
{
    return getLanguageFromCommandLine()
            .or_else(getLanguageFromRegistry)
            .or_else(getLanguageFromEnvironment)
            .value_or(ELanguage::English);
}
```

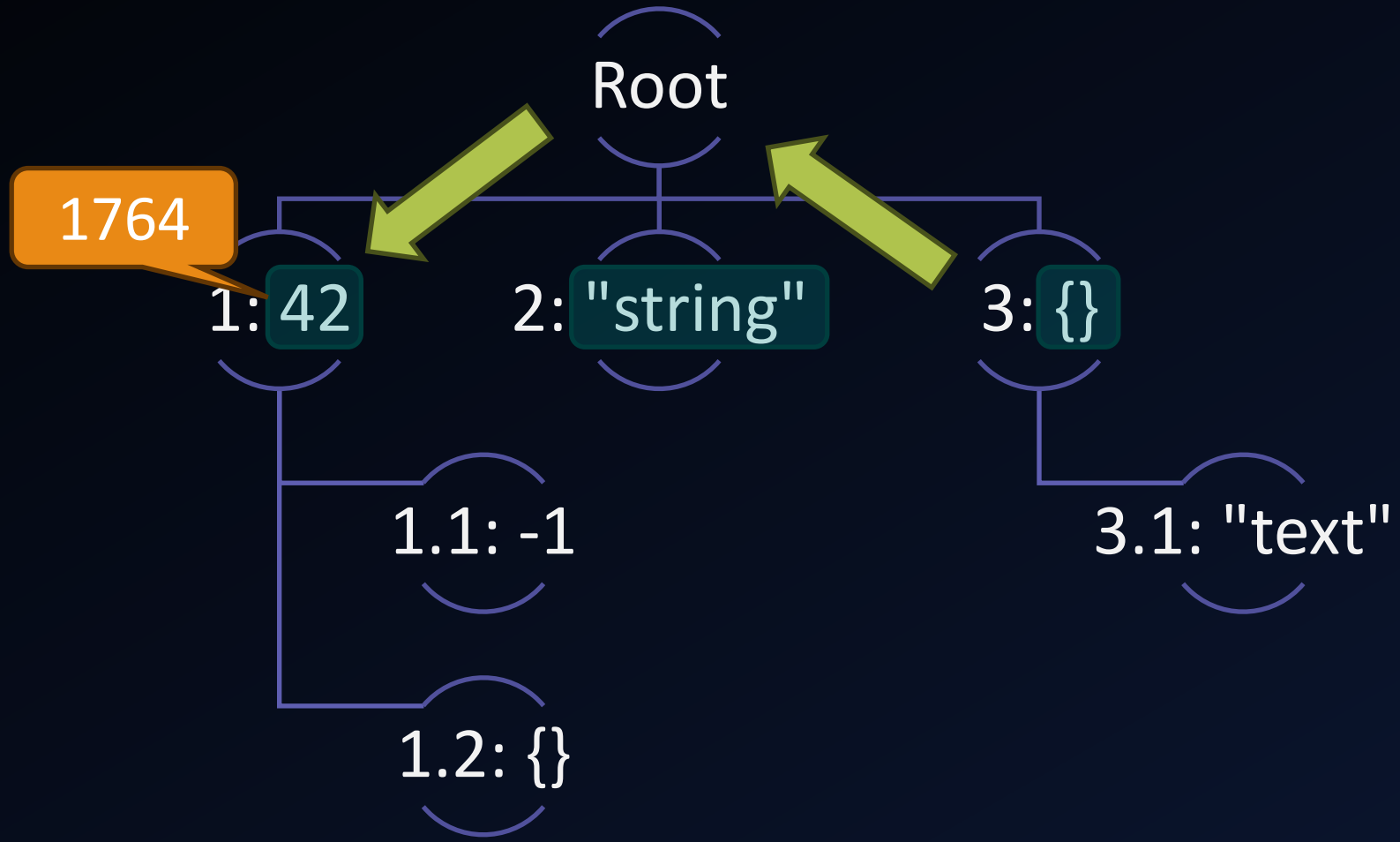
# Compiler Support (Minimum Required Version)

| Feature | C++ Version | GCC | Clang | MSVC |
|---|---|---|---|---|
| std::format | 20 | 13 | 17 | 19.29 |
| std::views::transform | 20 | 10 | 15 | 19.29 |
| std::ranges::to | 23 | 14 | 17 | 19.34 |
| std::bind_front | 20 | 9 | 13 | 19.25 |
| std::bind_back | 23 | 14 | 19 | 19.34 |
| std::optional::and_then / or_else | 23 | 12 | 14 | 19.32 |
| std::print / std::println | 23 | 14 | 18 | 19.37 |
| std::stacktrace | 23 | - | - | 19.34 |
| std::expected | 23 | 12 | 16 | 19.33 |
| Current compiler version | | 14.2 | 19.1 | 19.40 |

# Beyond C++23

## MORE MONADS

# Maybe Null: Chaining Functions That Return Pointers



# Maybe Null: Chaining Functions That Return Pointers

```
using CContent = std::variant<std::monostate, int, std::string>;

class CNode
{
public:
     //...
     CNode* getParent();
     CNode* getChild(const CNodeKey nodeKey);

     CContent getContent() const;
     void setContent(const CContent& arg);
     //...
};
```

# Maybe Null: Chaining Functions That Return Pointers

```
std::optional<int> getSiblingValueSquared
        (CNode* pNodeStart, const CNodeKey& siblingKey)
{
   if ( ! pNodeStart)      { return{}; }

   auto* const pParent = pNodeStart->getParent();
   if ( ! pParent)         { return{}; }

   auto* const pSibling = pParent->getChild(siblingKey);
   auto oNumericValue = getNodeNumericValue(pSibling);
   return oNumericValue.transform(makeSquared);
}
```

# Maybe Null: Chaining Functions That Return Pointers

```
std::optional<int> getSiblingValueSquared
        (CNode* pNodeStart, const CNodeKey& siblingKey)
{
   if ( ! pNodeStart)      { return{}; }

   auto* const pParent = pNodeStart->getParent();
   if ( ! pParent)         { return{}; }

   auto* const pSibling = pParent->getChild(siblingKey);
   auto oNumericValue = getNodeNumericValue(pSibling);
   return oNumericValue.transform(makeSquared);
}
```

# Maybe Null: Chaining Functions That Return Pointers

```
std::optional<int> getSiblingValueSquared
        (CNode* pNodeStart, const CNodeKey& siblingKey)
{
   if ( ! pNodeStart)      { return{}; }

   auto* const pParent = pNodeStart->getParent();
   if ( ! pParent)         { return{}; }

   auto* const pSibling = pParent->getChild(siblingKey);
   auto oNumericValue = getNodeNumericValue(pSibling);
   return oNumericValue.transform(makeSquared);
}
```

```
if ( ! pNode)
{
   return {};
}
```

# Maybe Null: Chaining Functions That Return Pointers

```
std::optional<int> getSiblingValueSquared
        (CNode* pNodeStart, const CNodeKey& siblingKey)
{
   if ( ! pNodeStart)      { return{}; }

   auto* const pParent = pNodeStart->getParent();
   if ( ! pParent)         { return{}; }

   auto* const pSibling = pParent->getChild(siblingKey);
   auto oNumericValue = getNodeNumericValue(pSibling);
   return oNumericValue.transform(makeSquared);
}
```

# Maybe Null: Chaining Functions That Return Pointers

```
std::optional<int> getSiblingValueSquared
        (CNode* pNodeStart, const CNodeKey& siblingKey)
{
   return CPtr(pNodeStart)
      .and_then([](CNode& node)
                  {return node.getParent();})
      .and_then([&siblingKey](CNode& node)
                  {return node.getChild(siblingKey);})
      .and_then(getNodeNumericValue)
      .transform(makeSquared);
}
```

# Maybe Null: Chaining Functions That Return Pointers

```
std::optional<int> getSiblingValueSquared
        (CNode* pNodeStart, const CNodeKey& siblingKey)
{
   return CPtr(pNodeStart)
      .and_then([](CNode& node)
                  {return node.getParent();})
      .and_then([&siblingKey](CNode& node)
                  {return node.getChild(siblingKey);})
      .and_then(getNodeNumericValue)
      .transform(makeSquared);
}
```

Alternative:
`std::mem_fn(&CNode::getParent)`

# Maybe Null: Chaining Functions That Return Pointers

```
std::optional<int> getNodeNumericValue(CNode& node)
{
    auto content = node.getContent();
    if ( ! std::holds_alternative<int>(content)) { return{}; }

    return std::get<int>(content);
}
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
struct CPtr {
   CPtr() = default;
   explicit CPtr(const TPtr& ptr);

   TPtr operator->() const;
   operator bool() const;
   bool operator!() const;

   //...
private:
   TPtr m_Ptr{};
};
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
struct CPtr {
   CPtr() = default;
   explicit CPtr(const TPtr& ptr);

   TPtr operator->() const;
   operator bool() const;
   bool operator!() const;

   //...
private:
   TPtr m_Ptr{};
};
```

```
template<class T>
concept cIsPointer = std::is_pointer_v<T>;
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
struct CPtr {
    CPtr() = default;
    explicit CPtr(const TPtr& ptr);

    TPtr operator->() const;
    operator bool() const;
    bool operator!() const;

    //...
private:
    TPtr m_Ptr{};
};
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
CPtr<TPtr >::CPtr(const TPtr& ptr)
    : m_Ptr(ptr)
{}
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
struct CPtr {
   CPtr() = default;
   explicit CPtr(const TPtr& ptr);

   TPtr operator->() const;
   operator bool() const;
   bool operator!() const;

   //...
private:
   TPtr m_Ptr{};
};
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
T CPtr<TPtr>::operator->() const
{ return m_Ptr; }

template<cIsPointer TPtr>
CPtr<TPtr>::operator bool() const
{ return m_Ptr; }

template<cIsPointer TPtr>
bool CPtr<TPtr>::operator!() const
{ return ! m_Ptr; }
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
struct CPtr {
   //...

   template<class TCall>
   auto and_then(TCall&& fInvoke);
private:
   TPtr m_Ptr{};
};
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
template<class TCall>
auto CPtr<TPtr>::and_then(TCall&& fInvoke)
{
    using TRef = std::add_lvalue_reference_t<std::remove_pointer_t<TPtr>>;
    using TRet = std::invoke_result_t<TCall,TRef>;

    if constexpr (cIsPointer<TRet>)
    {
        //...return type is a pointer
    }
    else
    {
        //...return type is NOT a pointer
    }
}
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
template<class TCall>
auto CPtr<TPtr>::and_then(TCall&& fInvoke)
{
    using TRef = std::add_lvalue_reference_t<std::remove_pointer_t<TPtr>>;
    using TRet = std::invoke_result_t<TCall,TRef>;

    if constexpr (cIsPointer<TRet>)
    {
        //...return type is a pointer
    }
    else
    {
        //...return type is NOT a pointer
    }
}
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
template<class TCall>
auto CPtr<TPtr>::and_then(TCall&& fInvoke)
{
    using TRef = std::add_lvalue_reference_t<std::remove_pointer_t<TPtr>>;
    using TRet = std::invoke_result_t<TCall,TRef>;

    if constexpr (cIsPointer<TRet>)
    {
        //...return type is a pointer
    }
    else
    {
        //...return type is NOT a pointer
    }
}
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
template<class TCall>
auto CPtr<TPtr>::and_then(TCall&& fInvoke)
{
    using TRef = std::add_lvalue_reference_t<std::remove_pointer_t<TPtr>>;
    using TRet = std::invoke_result_t<TCall,TRef>;

    if constexpr (cIsPointer<TRet>)
    {
        //...return type is a pointer
    }
    else
    {
        //...return type is NOT a pointer
    }
}
```

# Implementing A Pointer Monad

```
if constexpr (cIsPointer<TRet>)
{
   if (m_Ptr)
   {
      return CPtr<TRet>(std::invoke(
                              std::forward<TCall>(fInvoke),*m_Ptr);
   }
   return CPtr<TRet>{};
}
```

# Implementing A Pointer Monad

```
if constexpr (cIsPointer<TRet>)
{
   if (m_Ptr)
   {
      return CPtr<TRet>(std::invoke(
                              std::forward<TCall>(fInvoke),*m_Ptr);
   }
   return CPtr<TRet>{};
}
```

# Implementing A Pointer Monad

```
if constexpr (cIsPointer<TRet>)
{
   if (m_Ptr)
   {
      return CPtr<TRet>(std::invoke(
                              std::forward<TCall>(fInvoke),*m_Ptr));
   }
   return CPtr<TRet>{};
}
```

# Implementing A Pointer Monad

```
if constexpr (cIsPointer<TRet>)
{
   if (m_Ptr)
   {
      return CPtr<TRet>(std::invoke(
                              std::forward<TCall>(fInvoke),*m_Ptr);
   }
   return CPtr<TRet>{};
}
```

# Implementing A Pointer Monad

```
if constexpr (cIsPointer<TRet>)
{
   if (m_Ptr)
   {
      return CPtr<TRet>(std::invoke(
                              std::forward<TCall>(fInvoke),*m_Ptr);
   }
   return CPtr<TRet>{};
}
```

# Implementing A Pointer Monad

```
template<cIsPointer TPtr>
template<class TCall>
auto CPtr<TPtr>::and_then(TCall&& fInvoke)
{
    using TRef = std::add_lvalue_reference_t<std::remove_pointer_t<TPtr>>;
    using TRet = std::invoke_result_t<TCall,TRef>;

    if constexpr (cIsPointer<TRet>)
    {
        //...return type is a pointer
    }
    else
    {
        //...return type is NOT a pointer
    }
}
```

# Implementing A Pointer Monad

```
else // Return type is not a pointer
{
   using TValue = TRet::value_type;
   static_assert(std::is_same_v<std::optional<TValue>,TRet>,
          "Return type is neither pointer nor std::optional");
   if (m_Ptr)
   {
      return std::invoke(std::forward<TCall>(fInvoke),*m_Ptr);
   }

   return std::remove_cvref_t<TRet>{};
}
```

# Implementing A Pointer Monad

```
else // Return type is not a pointer
{
   using TValue = TRet::value_type;
   static_assert(std::is_same_v<std::optional<TValue>,TRet>,
          "Return type is neither pointer nor std::optional");
   if (m_Ptr)
   {
      return std::invoke(std::forward<TCall>(fInvoke),*m_Ptr);
   }

   return std::remove_cvref_t<TRet>{};
}
```

# Implementing A Pointer Monad

```
else // Return type is not a pointer
{
   using TValue = TRet::value_type;
   static_assert(std::is_same_v<std::optional<TValue>,TRet>,
          "Return type is neither pointer nor std::optional");
   if (m_Ptr)
   {
      return std::invoke(std::forward<TCall>(fInvoke),*m_Ptr);
   }

   return std::remove_cvref_t<TRet>{};
}
```

# Implementing A Pointer Monad

```
else // Return type is not a pointer
{
   using TValue = TRet::value_type;
   static_assert(std::is_same_v<std::optional<TValue>,TRet>,
          "Return type is neither pointer nor std::optional");
   if (m_Ptr)
   {
      return std::invoke(std::forward<TCall>(fInvoke),*m_Ptr);
   }

   return std::remove_cvref_t<TRet>{};
}
```

# Implementing A Pointer Monad

```
if constexpr (cIsPointer<TRet>)
{
   if (m_Ptr)
   {
      return CPtr<TRet>(std::invoke(std::forward<TCall>(fInvoke), //...
   }
   return CPtr<TRet>{};
}
else // Return type is not a pointer
{
   if (m_Ptr)
   {
      return std::invoke(std::forward<TCall>(fInvoke), //...
   }
   return std::remove_cvref_t<TRet>{};
}
```

# Implementing A Pointer Monad

```
std::optional<int> getSiblingValueSquared
        (CNode* pNodeStart, const CNodeKey& siblingKey)
{
    return CPtr(pNodeStart)
        .and_then([](CNode& node)
                  {return node.getParent();})
        .and_then([&siblingKey](CNode& node)
                  {return node.getChild(siblingKey);})
        .and_then(getNodeNumericValue)
        .transform(makeSquared);
}
```

# Implementing A Pointer Monad

```
std::optional<int> getSiblingValueSquared
        (CNode* pNodeStart, const CNodeKey& siblingKey)
{
    return CPtr(pNodeStart)
        .and_then([](CNode& node)
                  {return node.getParent();})
        .and_then([&siblingKey](CNode& node)
                  {return node.getChild(siblingKey);})
        .and_then(getNodeNumericValue)
        .transform(makeSquared);
}
```

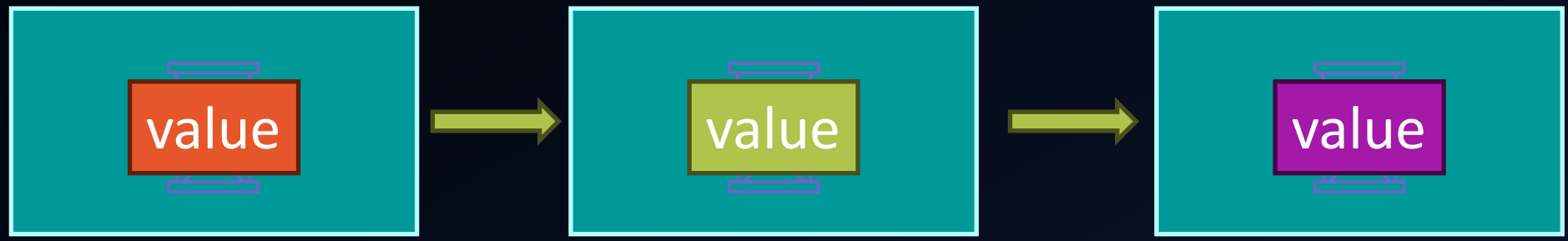
# Pointer Monad – Summary

- Writing monads does not need a lot of code
- General advice for writing template code applies
  - Test for specific types
  - Think about how different value categories affect your code
  - Use concepts and static_assert to prevent misuse at compile time

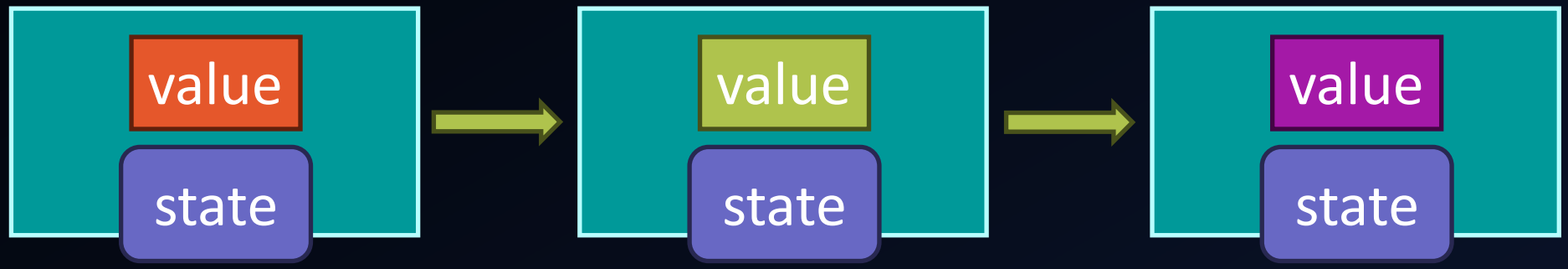
# Monad Use Cases

## WHAT ELSE YOU CAN DO WITH THEM

# Continuation Monad

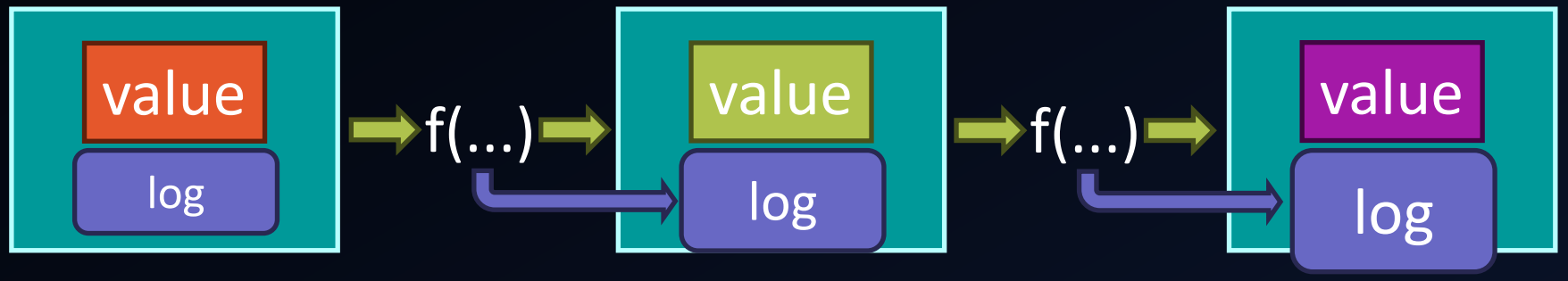


# State Monad

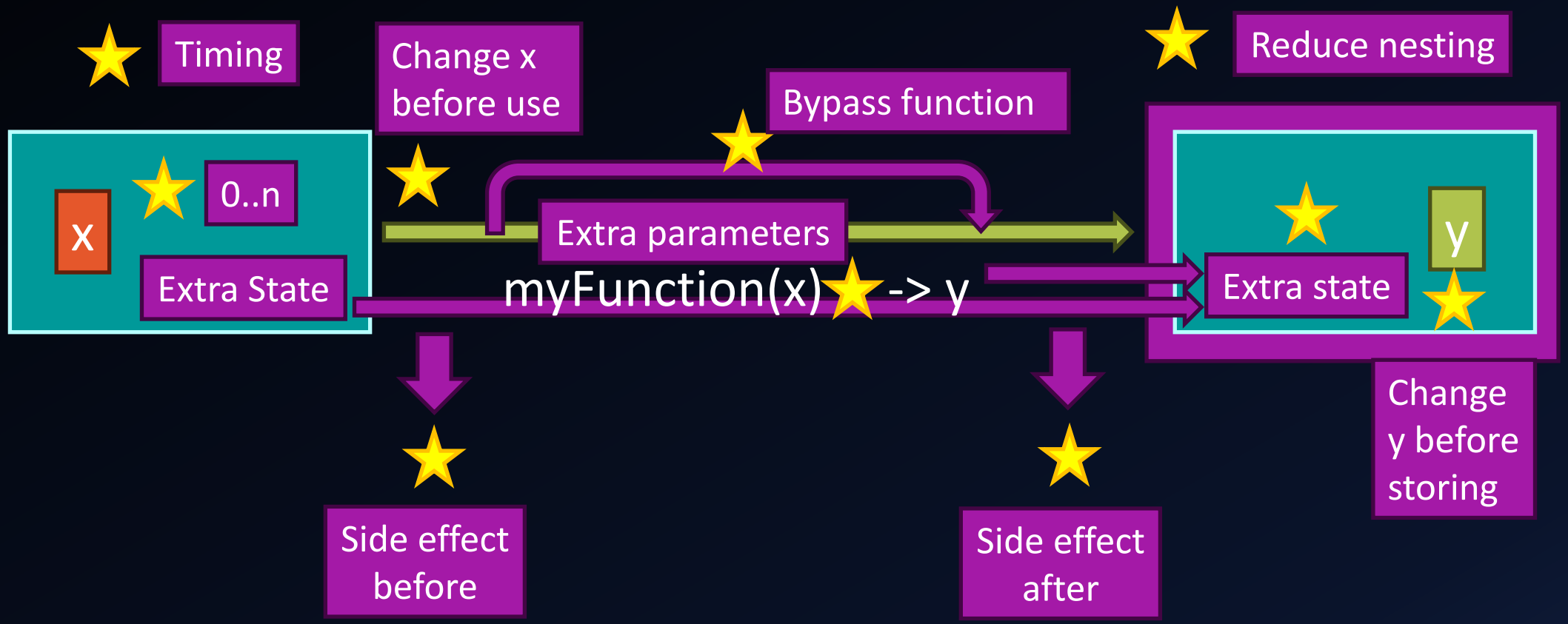



newValue = f(value, state);

# Writer Monad



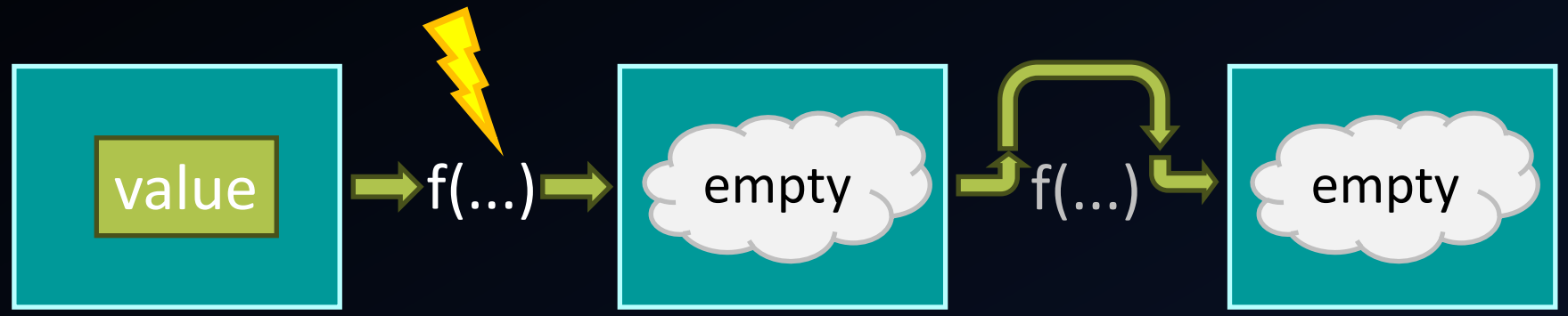
# The Power of Monads



# Ranges Monad

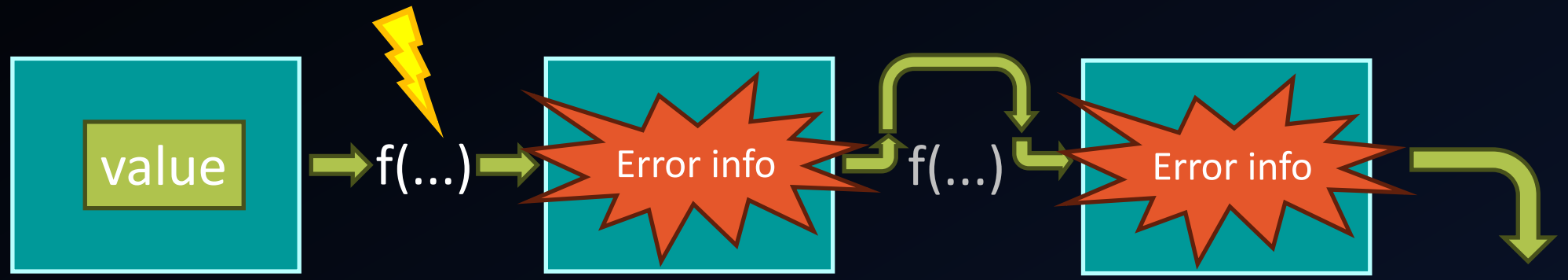

```
auto diagnostics = projects
     | vw::transform(getFilesInProject)
     | vw::transform(compile)
     | vw::join
     | vw::join;
```

# Optional ("Maybe") Monad



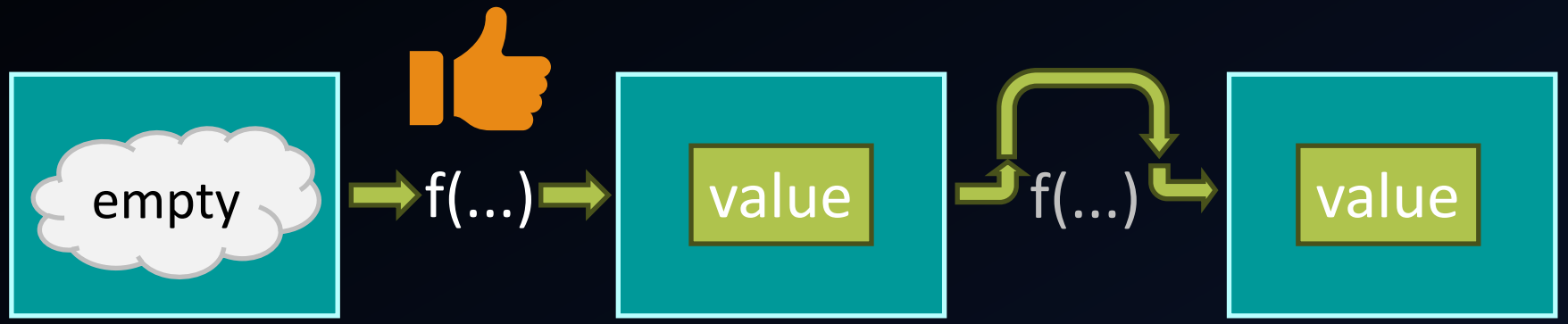

```
return getElement(db, key)
          .and_then(getTable)
          .and_then([location](const CTableData& table)
                    {return getCell(table, location);})
          .and_then(getNumericCellValue)
          .transform(isNegative);
```

# Expected Monad




```
return getElement(db, key)
          .and_then(getTable)
          .and_then([location](const CTableData& table)
                    {return getCell(table, location);})
          .and_then(getNumericCellValue)
          .transform(isNegative).or_else(log<bool>);
```

# Default Monad




```
return getLanguageFromCommandLine()
            .or_else(getLanguageFromRegistry)
            .or_else(getLanguageFromEnvironment)
            .value_or(ELanguage::English);
```

# Monadic Operations in C++23


ROBERT SCHIMKOWITSCH

https://mastodon.social/@asperamanca
https://github.com/Asperamanca/
https://cppusergroupvienna.org/

# References

https://github.com/Asperamanca/monadic_operations_cpp23