

Safe and Readable Code: Monadic Operations in C++23



ROBERT SCHIMKOWITSCH

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

About Me



Senior Software Engineer @ Andritz Hydro



Multi-platform
desktop systems (Qt)

Client architecture
Requirement refinement
Mentoring junior colleagues

Robert Schimkowitsch

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

Co-Funder of
C++ User Group Vienna



Handling Potential Failure

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

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

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

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

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

Handling Potential Failure

```
bool getIntCellValueNegative  
    (CDb db, Key key, CLocation location, bool& out)  
{  
    CElement element;  
    if ( ! getElement(db, key, element)) { return false; }  
  
    CTable table;  
    if ( ! getTable(element, table)) { return false; }  
  
    CCell cell;  
    if ( ! getCell(table, location, cell)) { return false; }  
  
    int value;  
    if ( ! getNumericCellValue(cell, value)) { return false; }  
  
    result = (value < 0);  
    return true;  
}
```

...With Exceptions

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

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

Call stack
getCell
isIntCellValueNegative
tryCall<int> (__cdecl&...)
testTable
main
...

	test.cpp	25
getCell	test.cpp	60
isIntCellValueNegative	test.cpp	38
tryCall<int> (__cdecl&...)	test.cpp	107
testTable	test.cpp	9

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

But I Can't or Won't use Exceptions!

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

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

    CCell 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::getIntCellValueNegative
    (CDb db, Key key, 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::getIntCellValueNegative
    (CDb db, Key key, 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;
}
```

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



Does an Error Flag Help?

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

bool CMyClass::getIntCellValueNegative
    (CDb db, Key key, 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

```
optional<bool> isIntCellValueNegative(CDb db, Key key, 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!

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative);
}
```

Goals

1. Understand what functors and monads do
2. Use monadic operations from std without much trouble
3. Have leads & references for further exploration

Code on Slides

```
// Don't copy code from slides!  
// Use GitHub or Compiler Explorer links  
// (see end of presentation)
```



```
string s;           // Standard library symbols are lime  
foo();             // Callables are orange  
template<class TValue> // Template types are purple  
class CUser;        // User-defined types are blue
```

```
// Slides may contain traces of unicorn
```

Functors

MAP, WRAP, TRANSFORM

Is this a Functor?

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

// ...

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

A Simple Sequence

```
double calcArea(double radius);
string formatArea(double area);

string foo1(double radius)
{
    return formatArea(calcArea(radius));
}
```

A Simple Sequence

```
double calcArea(double radius);
string formatArea(double area);

string foo1(double radius)
{
    return formatArea(calcArea(radius));
}
```

Adding a Version with a std::vector of Input Values

```
double calcArea(double radius);
string formatArea(double area);

string foo1(double radius)
{
    return formatArea(calcArea(radius));
}

vector<string> fooVec(vector<double> vecRadii)
{
    vector<string> vecOutput;
    for(const double& radius : vecRadius)
    {
        resultVec.push_back(formatArea(calcArea(radius)));
    }
    return vecOutput;
}
```

Adding a Version with a std::vector of Input Values

```
double calcArea(double radius);
string formatArea(double area);

string foo1(double radius)
{
    return formatArea(calcArea(radius));
}

vector<string> fooVec(vector<double> vecRadii)
{
    vector<string> vecOutput;
    for(const double& radius : vecRadius)
    {
        resultVec.push_back(formatArea(calcArea(radius)));
    }
    return vecOutput;
}
```

Adding a Version with a std::vector of Input Values

```
double calcArea(double radius);
string formatArea(double area);

string foo1(double radius)
{
    return formatArea(calcArea(radius));
}

vector<string> fooVec(vector<double> vecRadii)
{
    vector<string> vecOutput;
    for(const double& radius : vecRadius)
    {
        resultVec.push_back(formatArea(calcArea(radius)));
    }
    return vecOutput;
}
```

Adding a Version with a std::vector of Input Values

```
string foo1(double radius)
{
    return formatArea(calcArea(radius));
}

vector<string> fooVec(vector<double> vecRadius)
{
    vector<string> vecOutput;
    for(const double& radius : vecRadius)
    {
        resultVec.push_back(formatArea(calcArea(radius)));
    }
    return vecOutput;
}
```

Handles vector part

Handles element part

Violation of Single Responsibility!
Vector-related code will be duplicated

A Minimal Functor

```
double calcArea(double radius);
string formatArea(double area);

vector<string> fooVec(vector<double> vecRadii)
{
    auto calcVecArea = liftVec<double>(calcArea);
    auto formatVecOutput = liftVec<double>(formatArea);
    return formatVecOutput(calcVecArea(vecRadius));
}
```

A Minimal Functor

```
double calcArea(double radius);
string formatArea(double area);

vector<string> fooVec(vector<double> vecRadii)
{
    auto calcVecArea = liftVec<double>(calcArea);
    auto formatVecOutput = liftVec<double>(formatArea);
    return formatVecOutput(calcVecArea(vecRadius));
}
```

New function
created by lifting
our old one

A Minimal Functor

```
double calcArea(double radius);  
string formatArea(double area);
```

We pass a callable here

```
vector<string> fooVec(vector<double> vecRadii)  
{  
    auto calcVecArea = liftVec<double>(calcArea);  
    auto formatVecOutput = liftVec<double>(formatArea);  
    return formatVecOutput(calcVecArea(vecRadius));  
}
```

New function
created by lifting
our old one

A Minimal Functor

```
double calcArea(double radius);
string formatArea(double area);

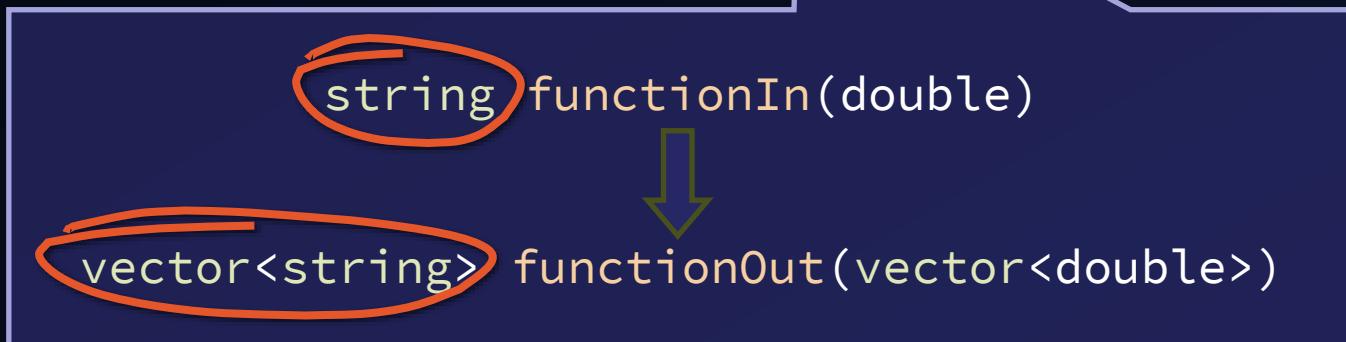
vector<string> fooVec(vector<double> vecRadii)
{
    auto calcVecArea = liftVec<double>(calcArea);
    auto formatVecOutput = liftVec<double>(formatArea);
    return formatVecOutput(calcVecArea(vecRadius));
}
```

```
double functionIn(double)
↓
vector<double> functionOut(vector<double>)
```

A Minimal Functor

```
double calcArea(double radius);
string formatArea(double area);

vector<string> fooVec(vector<double> vecRadii)
{
    auto calcVecArea = liftVec<double>(calcArea);
    auto formatVecOutput = liftVec<double>(formatArea);
    return formatVecOutput(calcVecArea(vecRadius));
}
```



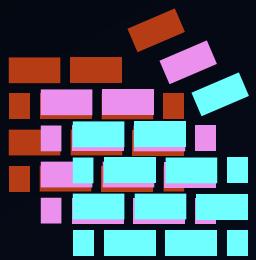
A Minimal Functor

```
double calcArea(double radius);
string formatArea(double area);

vector<string> fooVec(vector<double> vecRadii)
{
    auto calcVecArea = liftVec<double>(calcArea);
    auto formatVecOutput = liftVec<double>(formatArea);
    return formatVecOutput(calcVecArea(vecRadius));
}
```

'Vector' part is added by liftVec
and implemented once
We only define the sequence of calls

A Functor Wrapper as a Magic Box...



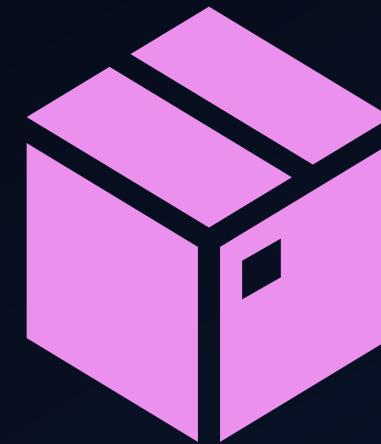
Put in a vector of bricks...



A Functor Wrapper as a Magic Box...



...send in a worker with a plan...
(still only one!)



A Functor Wrapper as a Magic Box...

...get a vector of houses



Using the Magic Box

```
double calcArea(double radius);
string formatArea(double area);

vector<string> fooVec(vector<double> vecRadii)
{
    return CFunctorVec{vecRadii}.transform(calcArea)
        .transform(formatArea)
        .result();
}
```

Using the Magic Box

```
double calcArea(double radius);
string formatArea(double area);

vector<string> fooVec(vector<double> vecRadii)
{
    return CFunctorVec{vecRadii}.transform(calcArea)
        .transform(formatArea)
        .result();
}
```

We pass a callable here

Using the Magic Box

```
double calcArea(double radius);
string formatArea(double area);

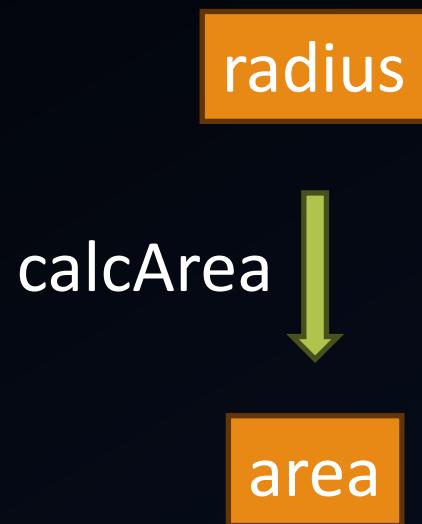
vector<string> fooVec(vector<double> vecRadii)
{
    return CFunctorVec{vecRadii}.transform(calcArea)
        .transform(formatArea)
        .result();
}
```

The 'Vector' part happens in CFunctorVec
and is implemented once
I only define the sequence of calls

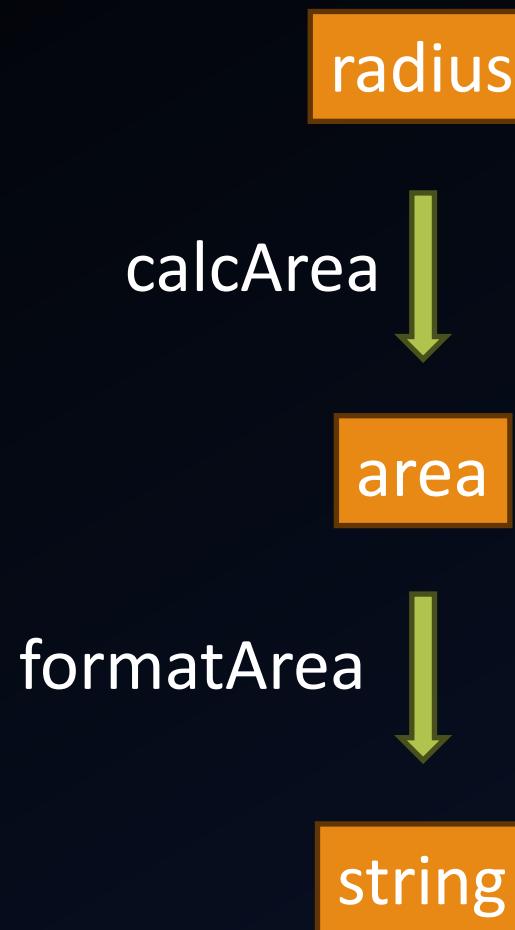
Functor, Step by Step

radius

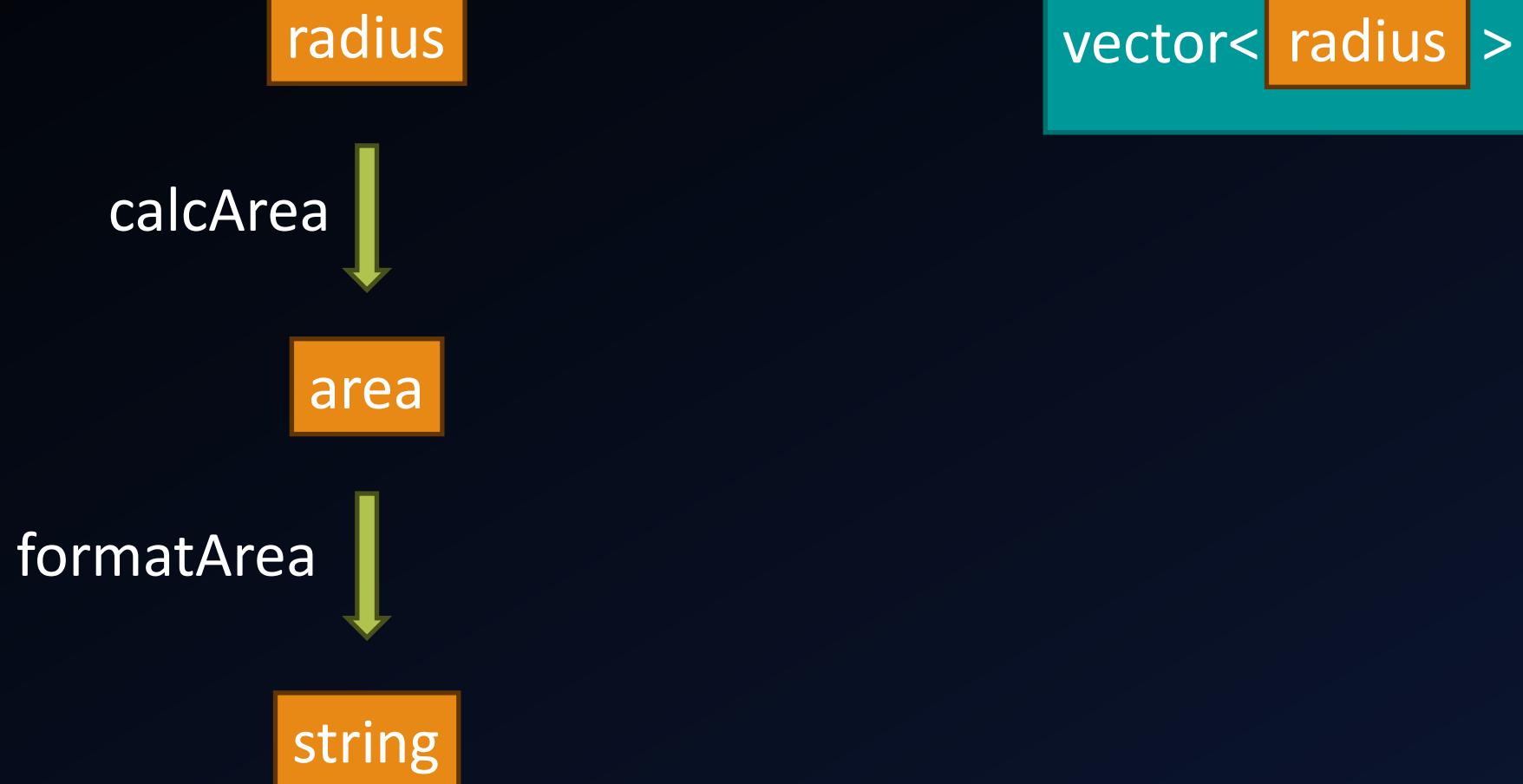
Functor, Step by Step



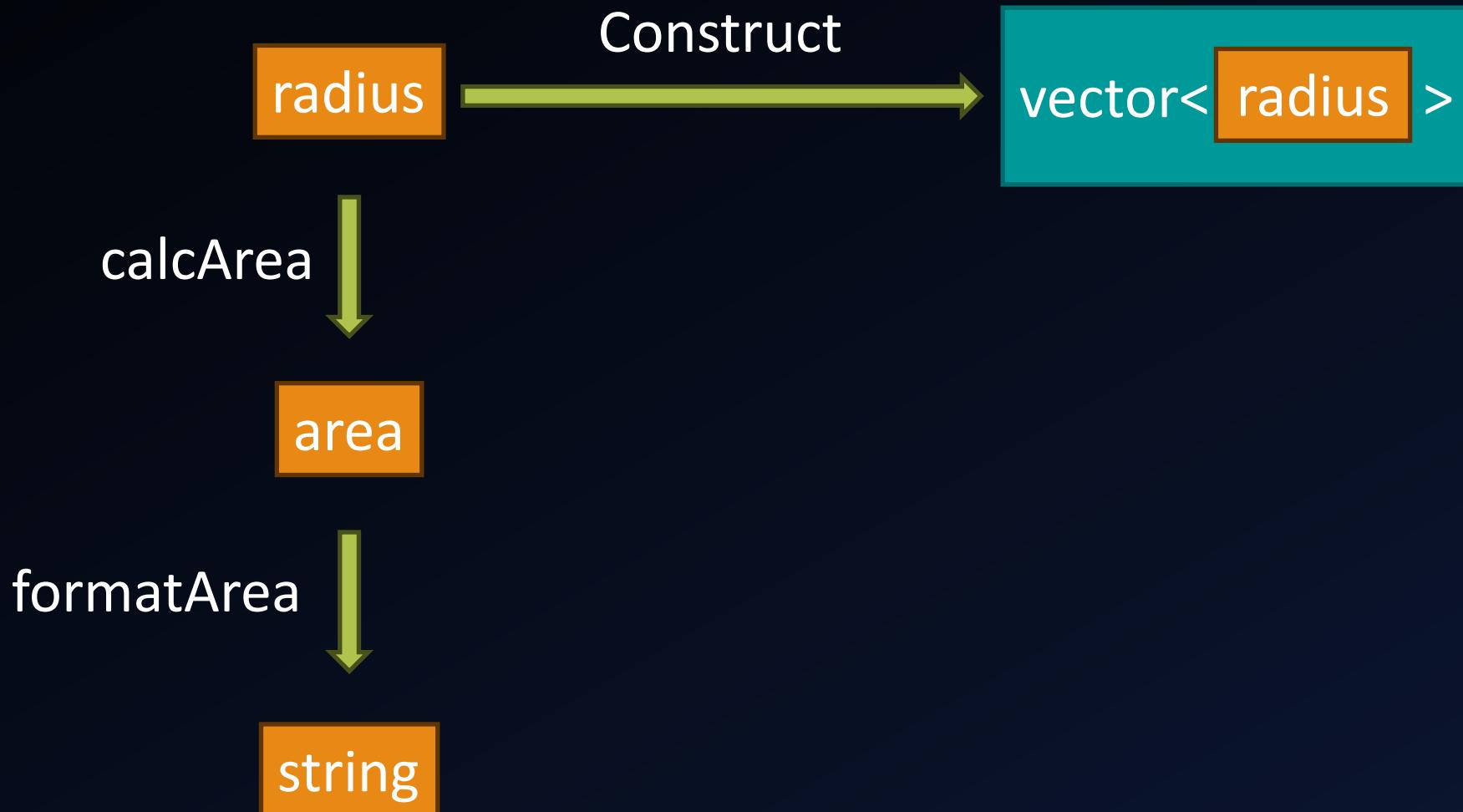
Functor, Step by Step



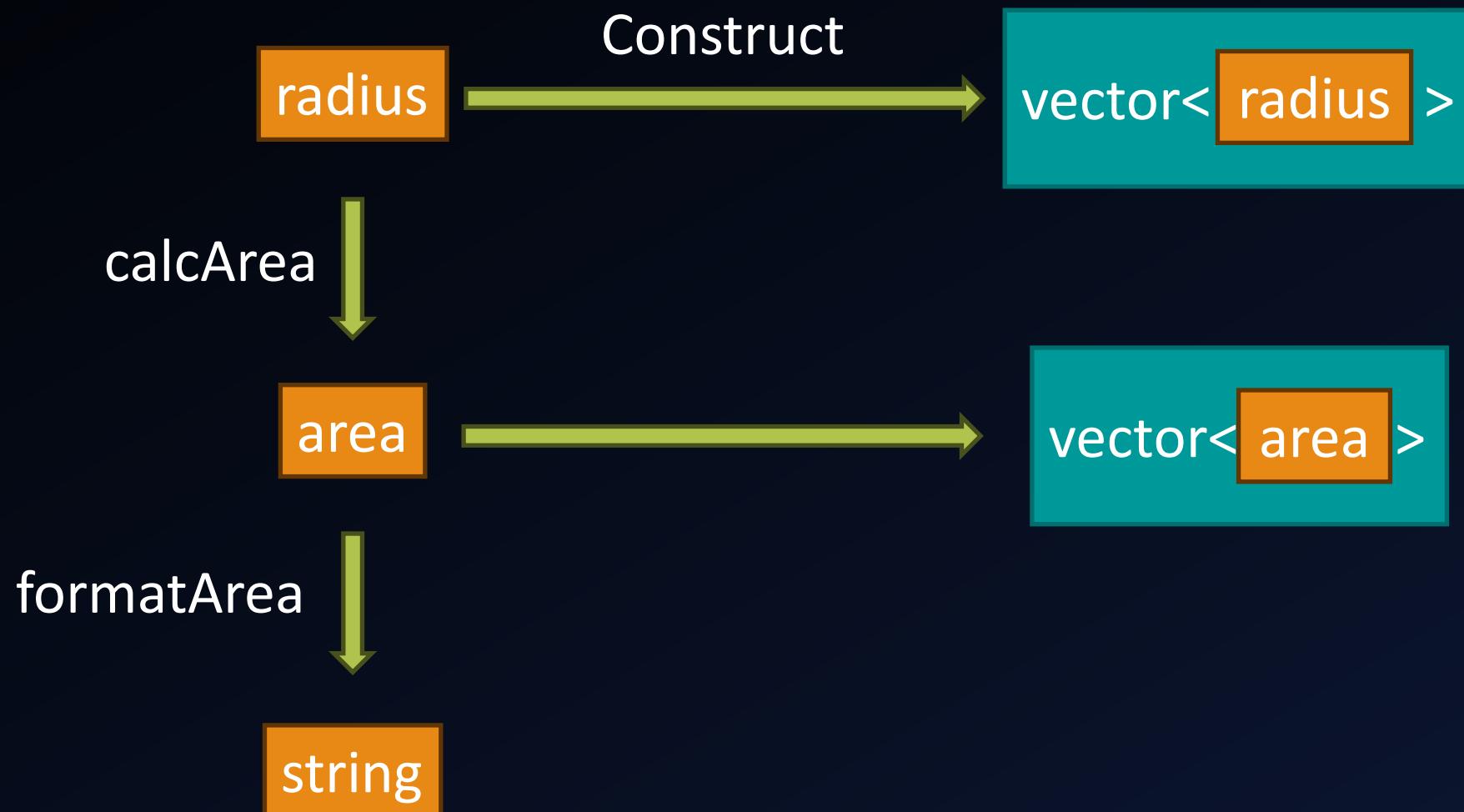
Functor, Step by Step



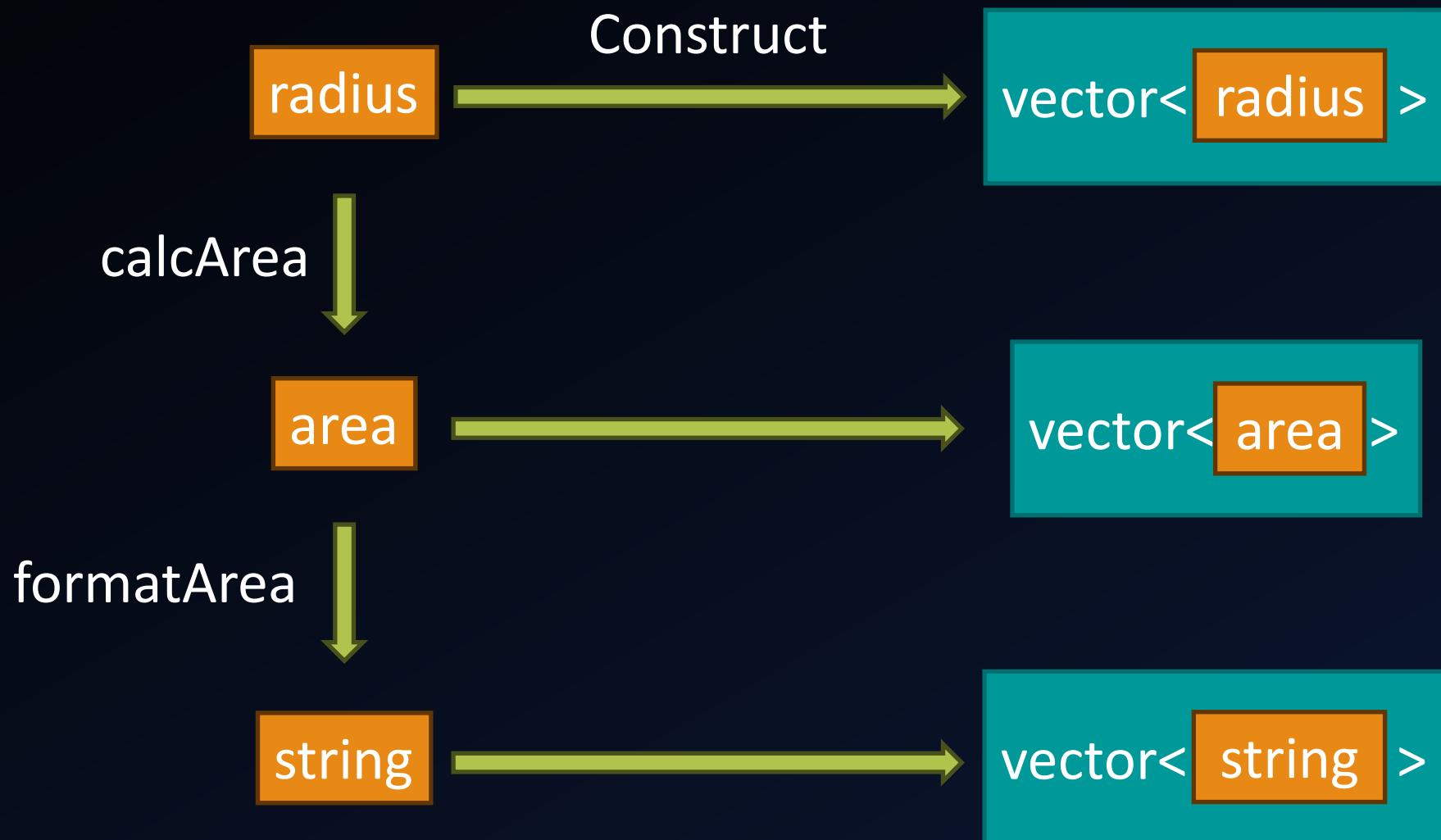
Functor, Step by Step



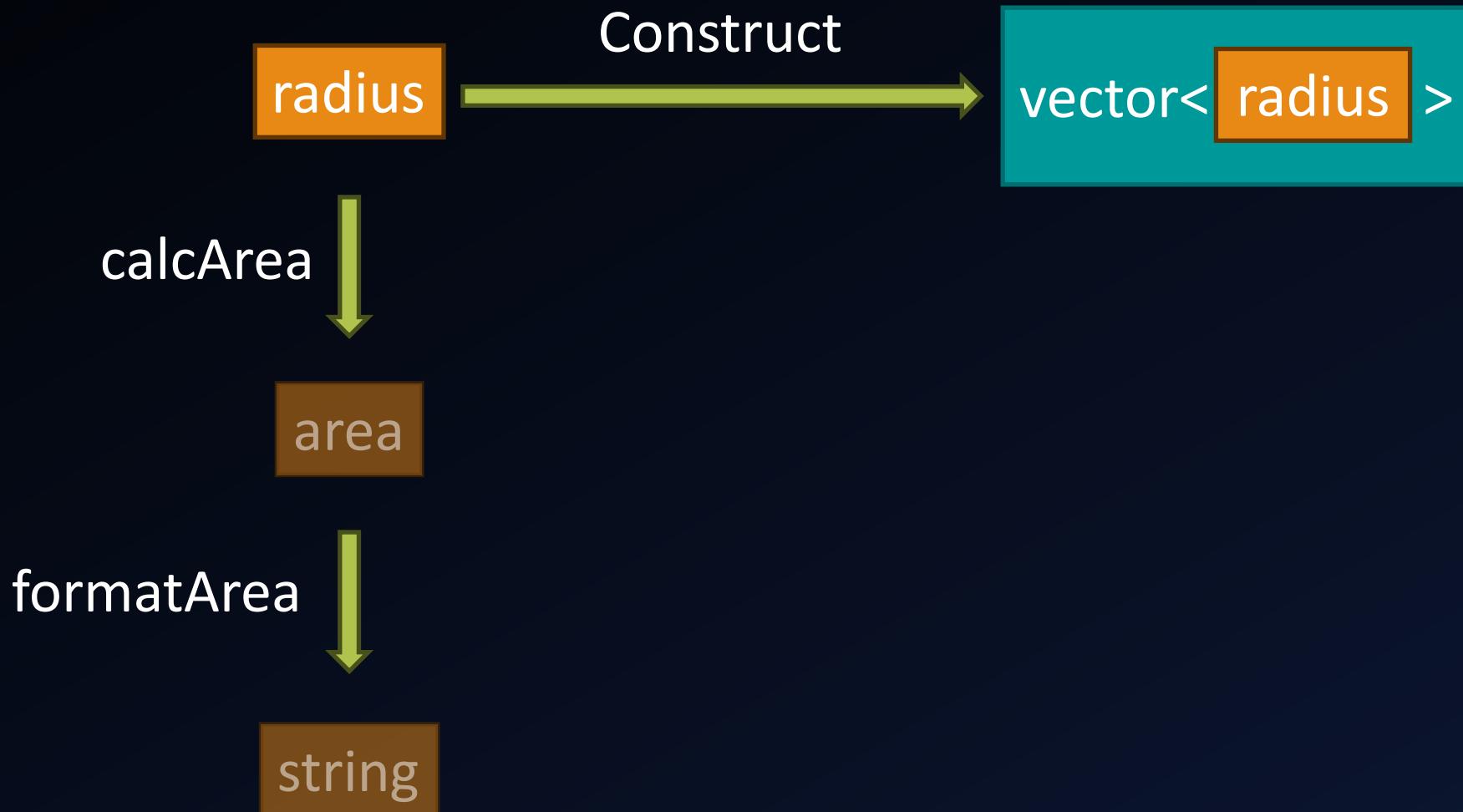
Functor, Step by Step



Functor, Step by Step



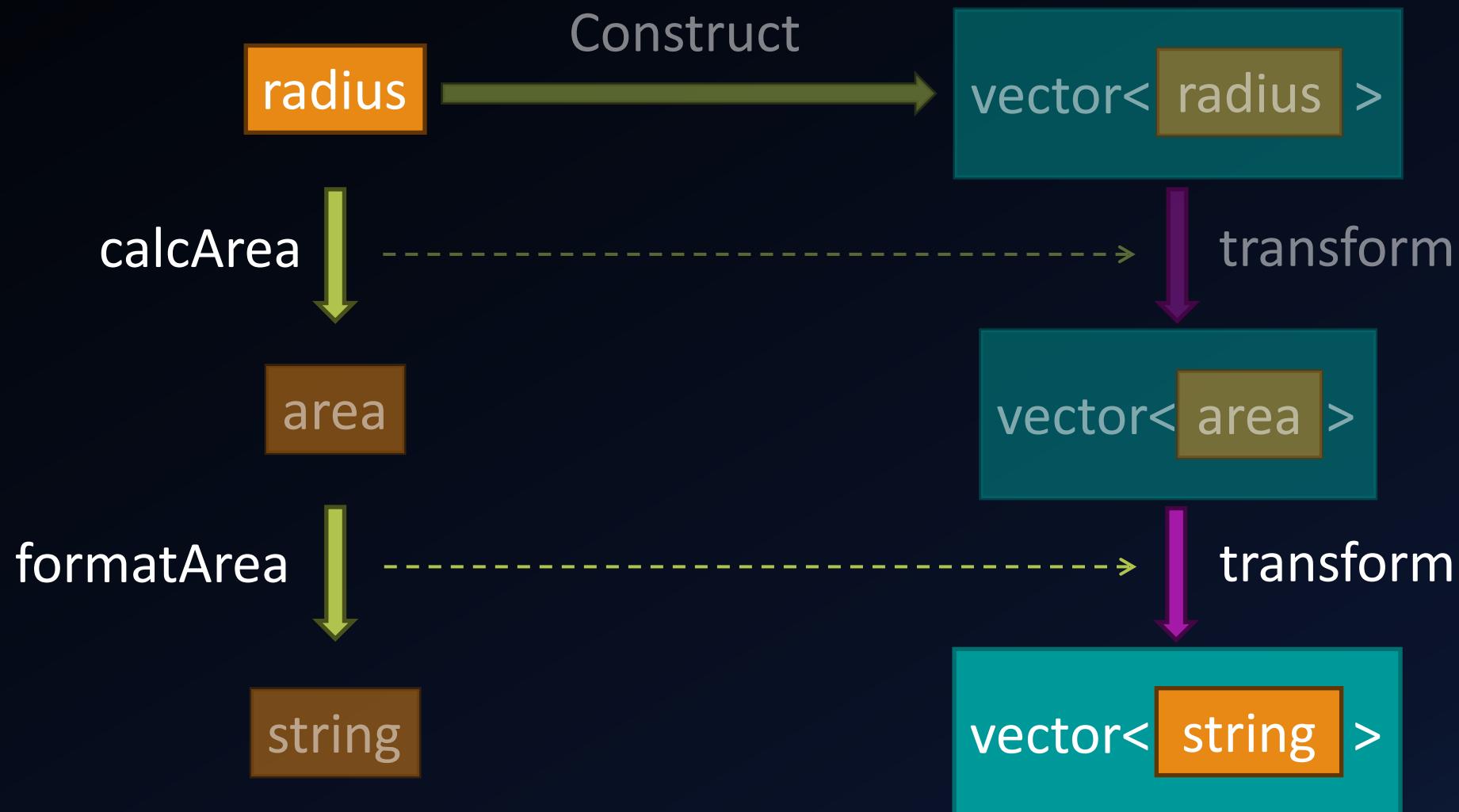
Functor, Step by Step



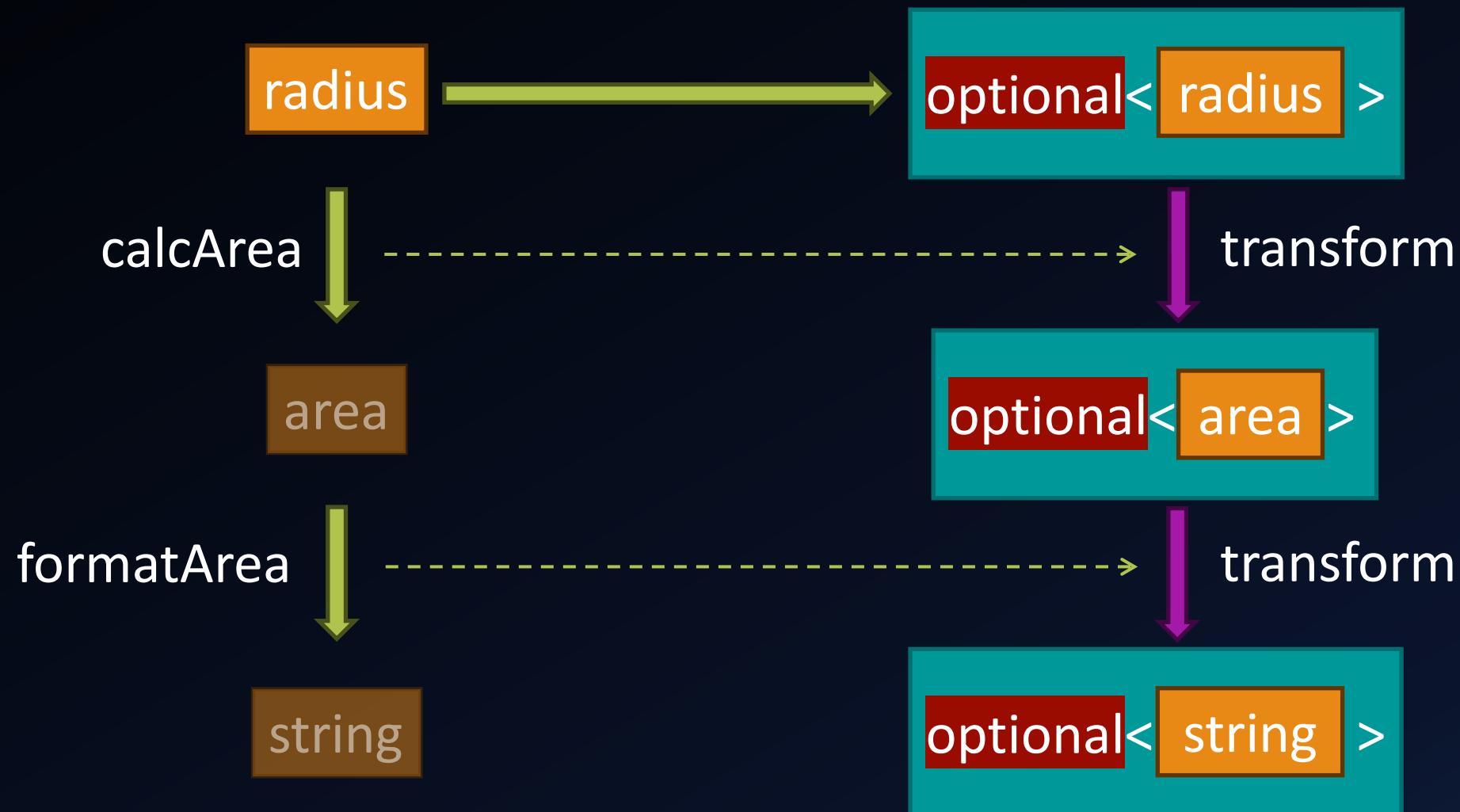
Now it's a Functor!



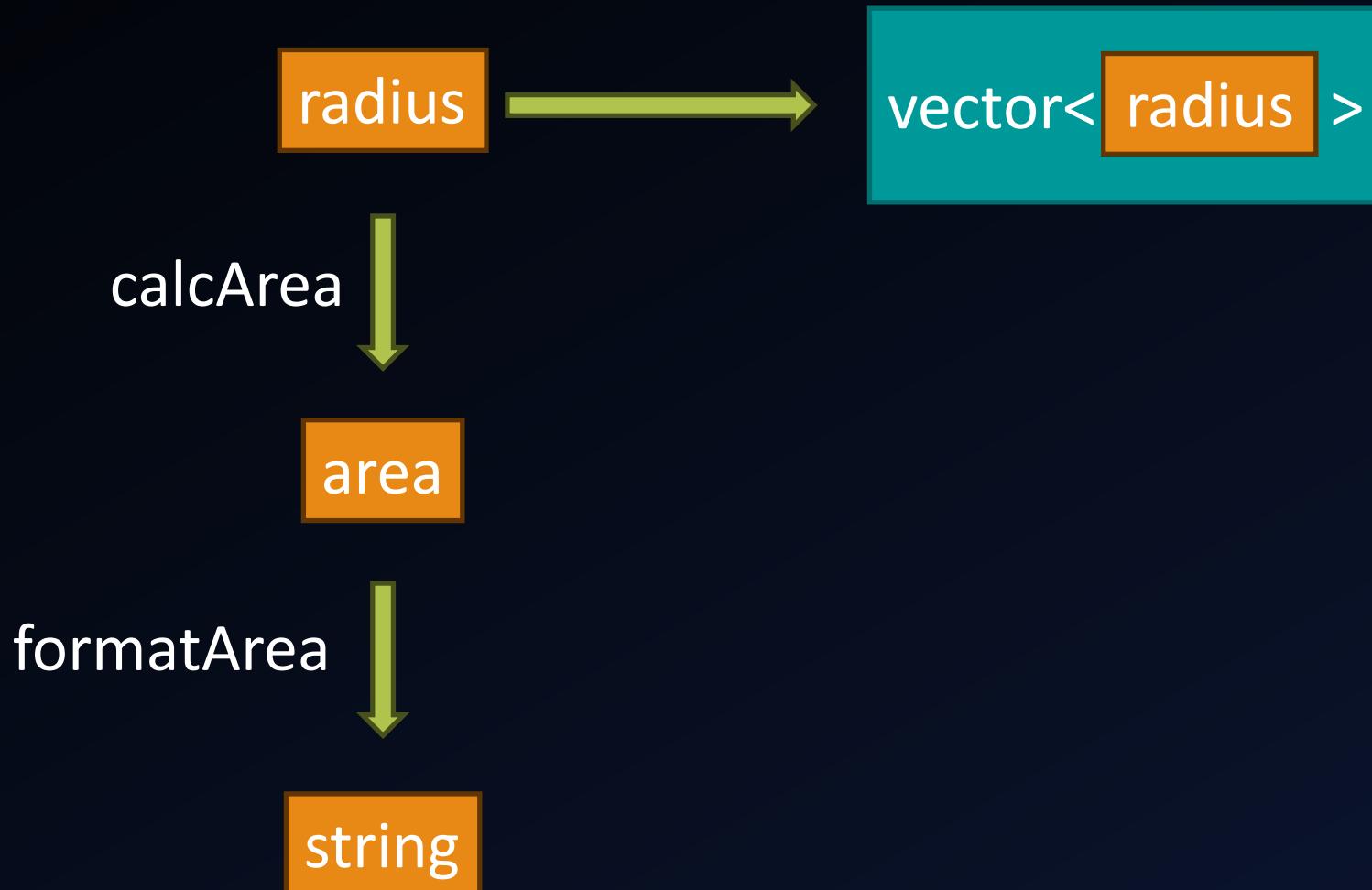
Now it's a Functor!



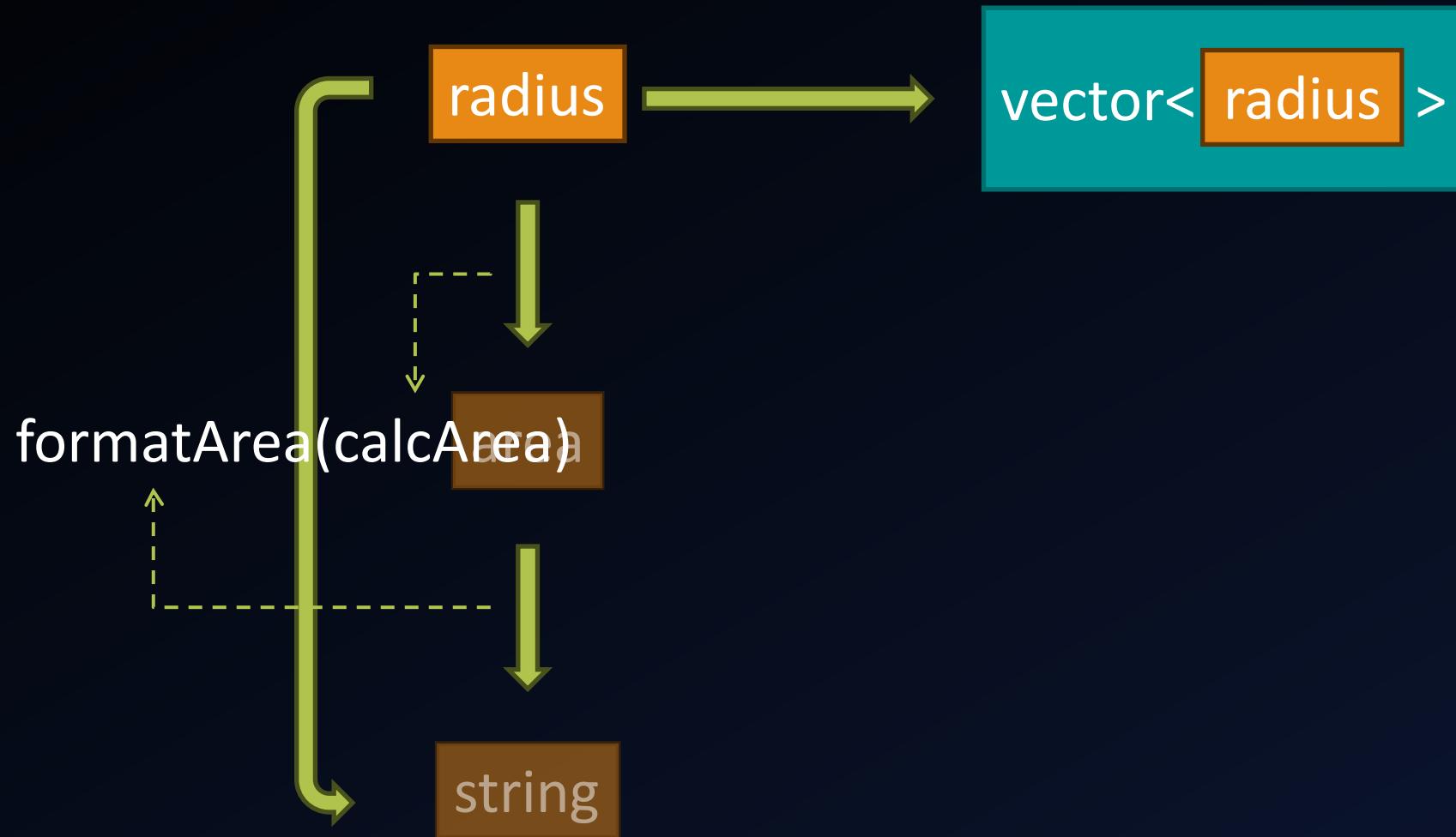
Replace the Functor Using the same Functions



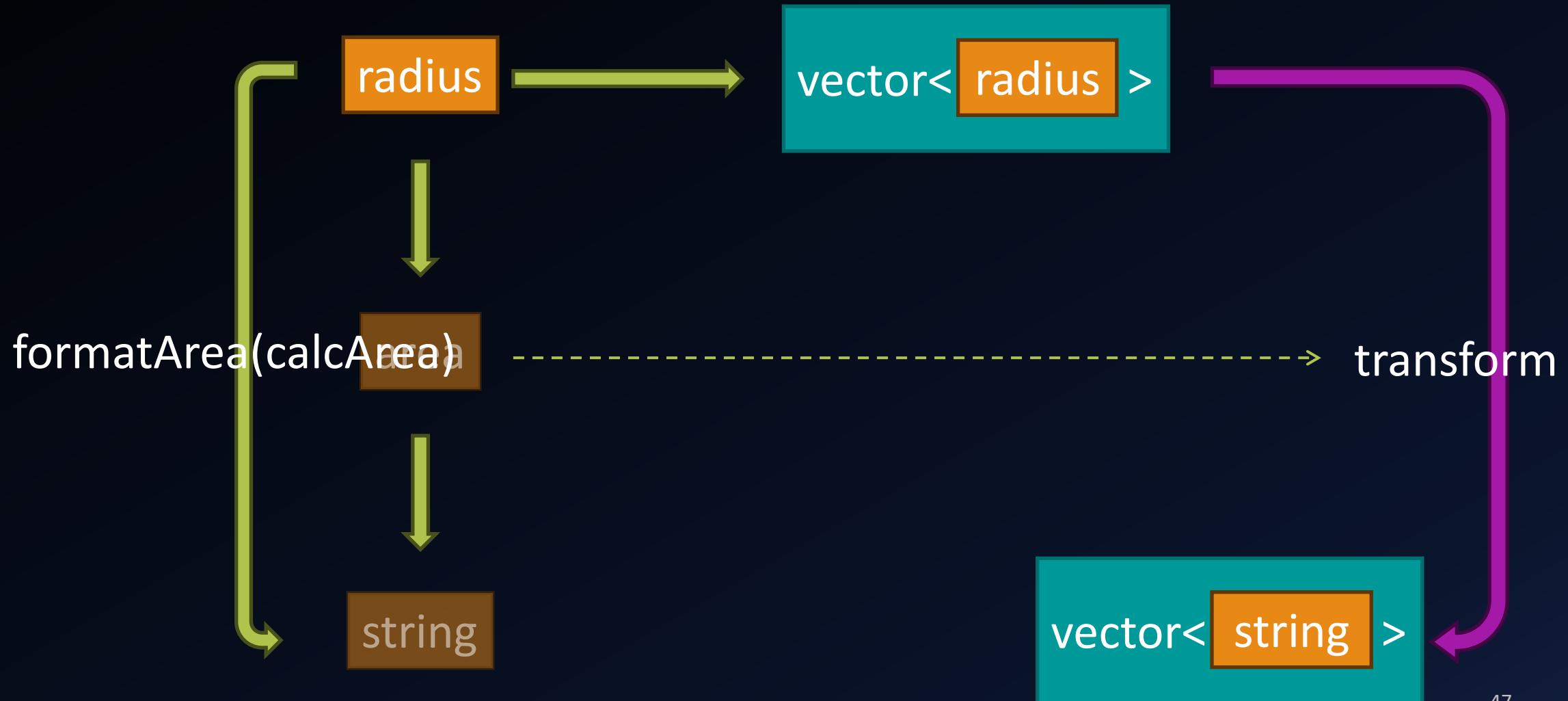
Two Rules for Functors



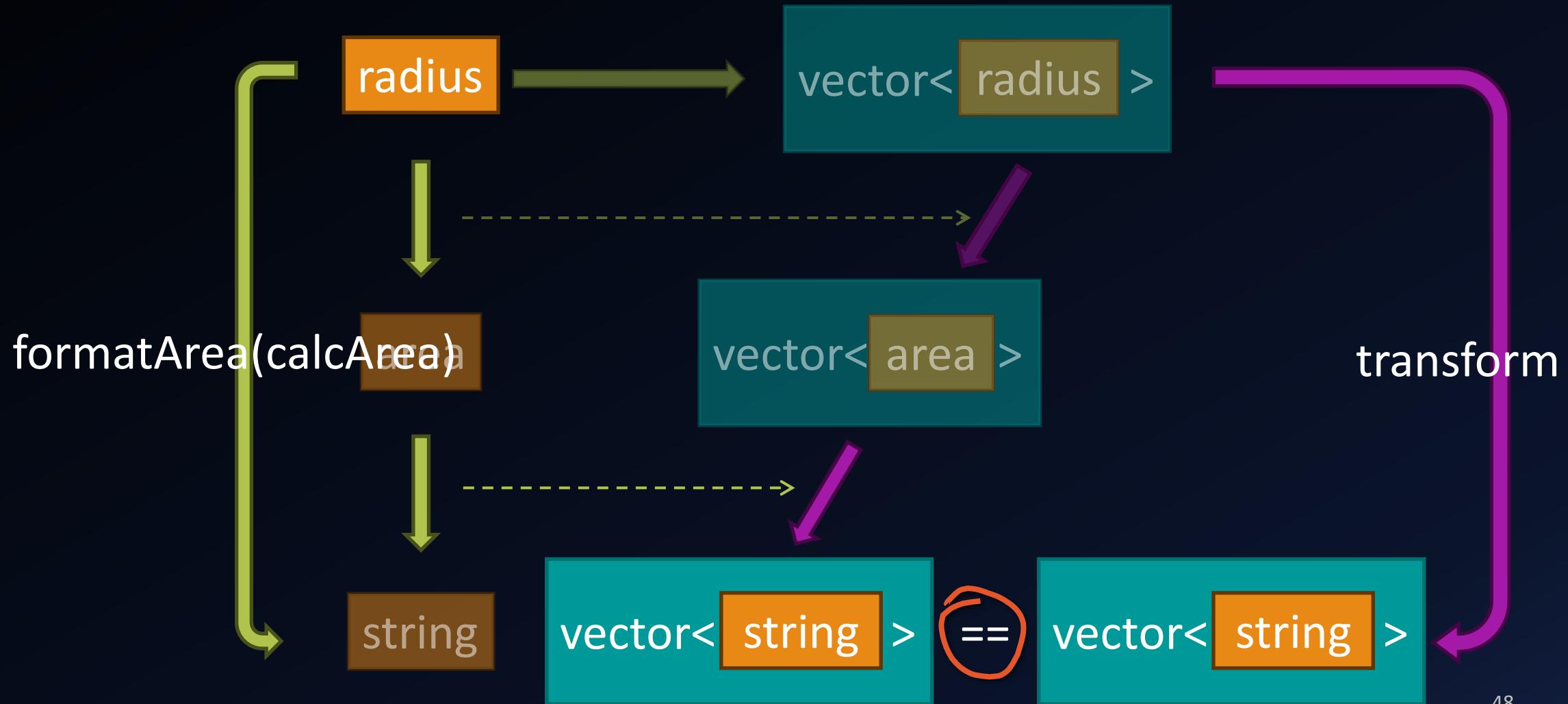
Rule #1: Composition Reflects in the Functor



Rule #1: Composition Reflects in the Functor



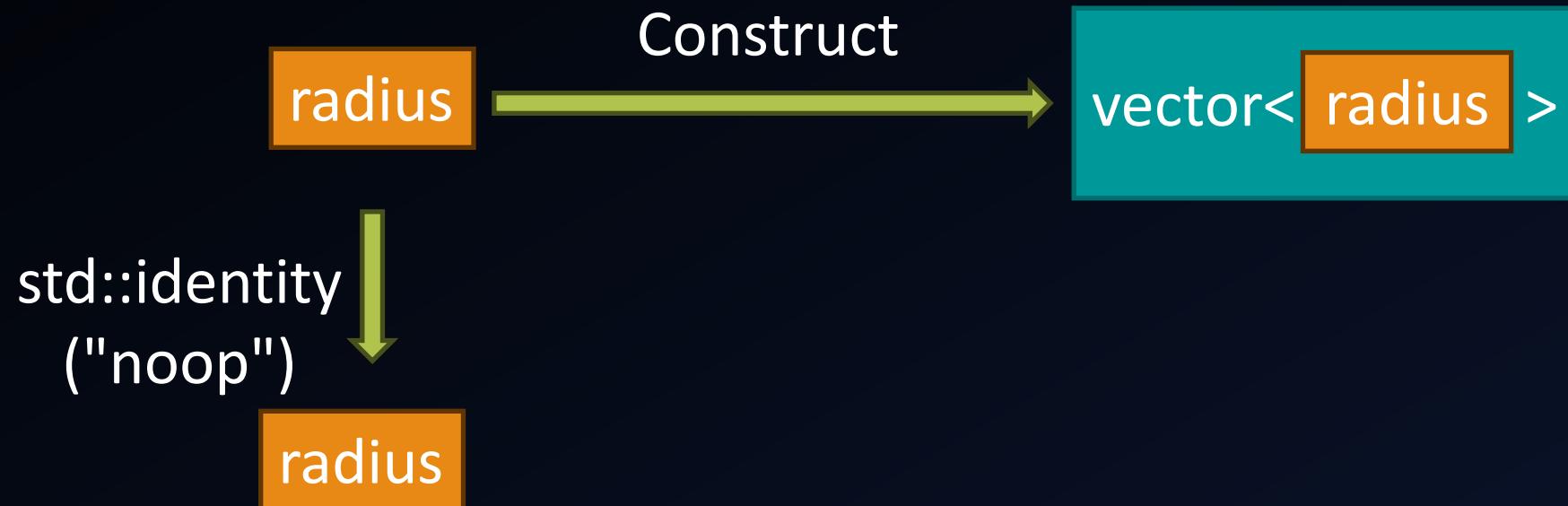
Rule #1: Composition Reflects in the Functor



Rule #2: Identity is Preserved

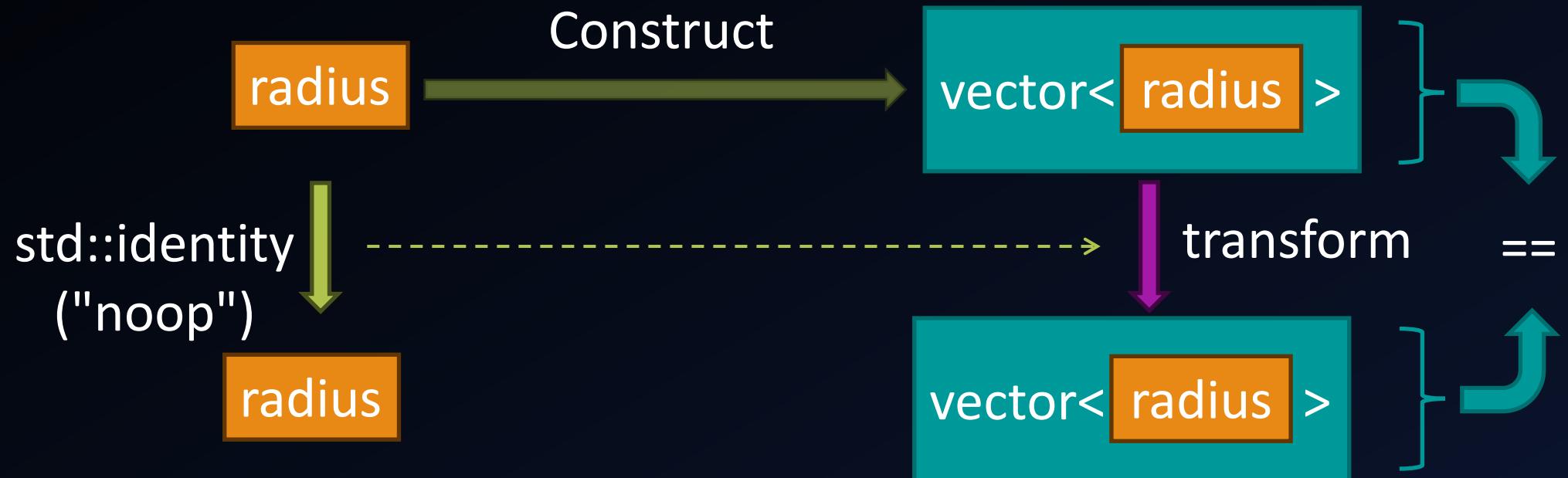


Rule #2: Identity is Preserved



(`std::identity` simply returns the input parameter)

Rule #2: Identity is Preserved



(`std::identity` simply returns the input parameter)

A functor
provides a
mapping
that preserves
composition
and
identity



...can be implemented in different ways

...does not have to be an object

...does not have to hold a value

A Functor with a Twist

STD::RANGES::VIEWS

A Look Back at our Classic Vector Example

```
double calcArea(double radius);
string formatArea(double area);

string foo1(double radius)
{
    return formatArea(calcArea(radius));
}

vector<string> fooVec(vector<double> vecRadii)
{
    vector<string> vecOutput;
    for(const double& radius : vecRadius)
    {
        resultVec.push_back(formatArea(calcArea(radius)));
    }
    return vecOutput;
}
```

Using std::views::transform

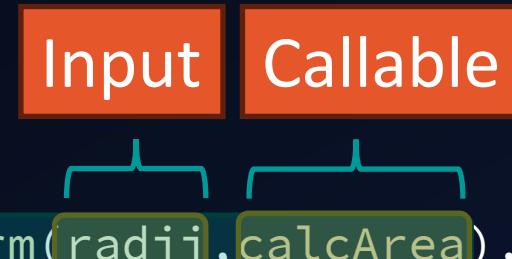
```
vector<string> fooVec(vector<double> vecRadii)
{
    vector<string> vecOutput;
    for(const double& radius : vecRadius)
    {
        resultVec.push_back(formatArea(calcArea(radius)));
    }
    return vecOutput;
}

vector<string> fooView(vector<double> vecRadii)
{
    auto output = views::transform(views::transform(vecRadii, calcArea),
                                  formatArea);
    // {"2.25", "4", "6.25"}
    //...
}
```

It's a Functor!

```
vector<string> fooVec(vector<double> vecRadii)
{
    vector<string> vecOutput;
    for(const double& radius : vecRadius)
    {
        resultVec.push_back(formatArea(calcArea(radius)));
    }
    return vecOutput;
}

vector<string> fooView(vector<double> vecRadii)
{
    auto output = views::transform(views::transform(vecRadii, calcArea),
                                  formatArea);
    // {"2.25", "4", "6.25"}
    //...
}
```



The diagram illustrates the components of a transform operation. It shows a sequence of nested function calls: `views::transform`, `views::transform`, and `formatArea`. The first `views::transform` takes an `Input` (the input vector `vecRadii`) and a `Callable` (the function `calcArea`). Its result is then passed as an `Input` to the second `views::transform`, which takes this result and another `Callable` (`formatArea`).

It's a Functor!

```
vector<string> fooVec(vector<double> vecRadii)
{
    vector<string> vecOutput;
    for(const double& radius : vecRadius)
    {
        resultVec.push_back(formatArea(calcArea(radius)));
    }
    return vecOutput;
}

vector<string> fooView(vector<double> vecRadii)
{
    auto output = views::transform(views::transform(vecRadii, calcArea),
                                  formatArea);
    // {"2.25", "4", "6.25"}
    //...
}
```

The diagram illustrates the execution flow of the `fooView` function. A blue arrow points from the `vecRadii` parameter to the first `views::transform` call. Another blue arrow points from the result of that transform to the second `views::transform` call. A green box labeled "Input" encloses the `vecRadii` parameter and the first `views::transform` call. A green box labeled "Callable" encloses the second `views::transform` call and its arguments (`calcArea` and `formatArea`). The final output is shown as a list of strings: {"2.25", "4", "6.25"}.

It's a Functor!

```
vector<string> fooVec(vector<double> vecRadii)
{
    vector<string> vecOutput;
    for(const double& radius : vecRadius)
    {
        resultVec.push_back(formatArea(calcArea(radius)));
    }
    return vecOutput;
}

vector<string> fooView(vector<double> vecRadii)
{
    auto output = views::transform(views::transform(vecRadii, calcArea),
                                  formatArea);
    // {"2.25", "4", "6.25"}
    //...
}
```

The Pipe to the Rescue

```
vector<string> fooView(vector<double> vecRadii)
{
    auto output = views::transform(views::transform(vecRadii, calcArea),
                                  formatArea);
    // {"2.25", "4", "6.25"}
    //...
}

vector<string> fooPipe(vector<double> vecRadii)
{
    auto output = vecRadii
        | views::transform(calcArea)
        | views::transform(formatArea);
    // {"2.25", "4", "6.25"}
    //...
}
```

The Pipe to the Rescue

```
vector<string> fooView(vector<double> vecRadii)
{
    auto output = views::transform(views::transform(vecRadii, calcArea),
                                  formatArea);
    // {"2.25", "4", "6.25"}
    //...
}

vector<string> fooPipe(vector<double> vecRadii)
{
    auto output = radii
                  | views::transform(calcArea)
                  | views::transform(formatArea);
    // {"2.25", "4", "6.25"}
    //...
}
```

The Pipe to the Rescue

```
vector<string> fooView(vector<double> vecRadii)
{
    auto output = views::transform(views::transform(vecRadii, calcArea),
                                  formatArea);
    // {"2.25", "4", "6.25"}
    //...
}

vector<string> fooPipe(vector<double> vecRadii)
{
    auto output = radii | views::transform(calcArea)
                    | views::transform(formatArea);
    // {"2.25", "4", "6.25"}
    //...
}
```

The Pipe to the Rescue

```
vector<string> fooView(vector<double> vecRadii)
{
    auto output = views::transform(views::transform(vecRadii, calcArea),
                                  formatArea);
    // {"2.25", "4", "6.25"}
    //...
}

vector<string> fooPipe(vector<double> vecRadii)
{
    auto output = vecRadii
        | views::transform(calcArea)
        | views::transform(formatArea);
    // {"2.25", "4", "6.25"}
    //...
}
```

The Pipe to the Rescue

```
vector<string> fooView(vector<double> vecRadii)
{
    auto output = views::transform(views::transform(vecRadii, calcArea),
                                  formatArea);
    // {"2.25", "4", "6.25"}
    //...
}

vector<string> fooPipe(vector<double> vecRadii)
{
    auto output = vecRadii
        | views::transform(calcArea)
        | views::transform(formatArea);
    // {"2.25", "4", "6.25"}
    //...
}
```

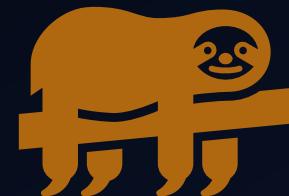
What do we Return?

```
vector<string> fooView(vector<double> vecRadii)
{
    auto output = views::transform(views::transform(vecRadii, calcArea),
                                  formatArea);
    // {"2.25", "4", "6.25"}
    // ...
}

vector<string> fooPipe(vector<double> vecRadii)
{
    auto output = vecRadii
        | views::transform(calcArea)
        | views::transform(formatArea);
    // {"2.25", "4", "6.25"}
    // ...
}
```

What Type is 'output'?

```
vector<string> fooPipe(vector<double> vecRadii)
{
    auto output = radii
        | views::transform(calcArea)
        | views::transform(formatArea);
    // {"2.25", "4", "6.25"}
    //...
}
```



'output' is not the result data
'output' is a view
(with building instructions)

views
do not own
the source data

What Type is 'output'?

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



Returning the View

```
inline auto fooPipe(vector<double> vecRadii)
{
    auto output = radii
        | views::transform(calcArea)
        | views::transform(formatArea);
    // {"2.25", "4", "6.25"}
    return output;
}
```

Not a template function!
inline manually!

Returning a Container

```
vector<string> fooPipe(vector<double> vecRadii)
{
    auto output = radii
        | views::transform(calcArea)
        | views::transform(formatArea);
    // {"2.25", "4", "6.25"}
    ranges::to<vector<string>>(output);
}
```



Views...

...are building
instructions,
not results

...typically do
not own
the source data

...are used
with 'auto'

...evaluate lazily

It's a Trap!

```
struct CEntry {  
    int m_X{};  
    int m_Y{};  
    string m_Text{};  
};  
  
CEntry getNearestEntry(const int x) {...}  
  
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{};  
    string m_Text{};  
};  
  
CEEntry getNearestEntry(const int x) {...}  
  
auto v = vector{1,3,7};  
  
auto strings = v  
    | views::transform(getNearestEntry)  
    | views::transform(&CEEntry::m_Text);  
  
printOutput(strings);
```



It's a Trap!

```
struct CEntry {  
    int m_X{};  
    int m_Y{};  
    string m_Text{};  
};  
  
CEEntry getNearestEntry(const int x) {...}  
  
auto v = vector{1,3,7};  
  
auto strings = v  
    | views::transform(getNearestEntry)  
    | views::transform(&CEEntry::m_Text);  
  
printOutput(strings);
```



Callables

FEED YOUR FUNCTOR

Callables: Free functions

```
double calcArea      (double radius);  
  
auto vecInput = vector{1.5,2.0,2.5};  
auto viewOutput = vecInput  
| views::transform(calcArea)  
| //...
```

Callables: (Class) Static Functions

```
class CConv
{
public:
    static double calcAreaStatic(double radius);
    //...
};

//...

auto vecInput = vector{1.5,2.0,2.5};
auto viewOutput = vecInput
    | views::transform(CConv::calcAreaStatic)

    | //...
```

Callables: Inline Lambda

```
auto vecInput = vector{1.5,2.0,2.5};  
auto viewOutput = vecInput  
| views::transform([](double radius)  
    { return pow(radius, 2.0) * numbers::pi; })  
| //...
```

Callables: Pick Overload with Inline Lambda

```
double calcArea(const double radius);
int    calcArea(const int value);

auto vecInput = vector{1.5,2.0,2.5};
auto viewOutput = vecInput
    | views::transform([](double radius)
        { return calcArea(radius); })
    | //...
```

Callables: Inject Parameters via Inline Lambda

```
const double power = 3.0;
auto vecInput = vector{1.5,2.0,2.5};
auto viewOutput = vecInput
    | views::transform([power](double value)
        { return pow(value, power); })
    | //...
```

Callables: Pass Object, Call Certain Member Function

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

auto vecInput = vector{CValue{1.5},CValue{2.0}};
auto viewOutput = vecInput
    | views::transform([](CValue obj)
        { return obj.getValue(); })
    | //...
```

Callables: Pass Value, Call Member of Certain Object

```
class CConv
{
public:
    double calcAreaMember(const double value);
};

//...

CConv conv;
auto vecInput = vector{1.5,2.0,2.5};
auto viewOutput = vecInput
    | views::transform([&conv](double value)
        { return conv.calcAreaMember(value); })
    | //...
```

Callables: Named Lambda

```
auto calcAreaLambda = [](double value)
{
    return pow(value,2.0);
};

auto vecInput = vector{1.5,2.0,2.5};
auto viewOutput = vecInput
    | views::transform(calcAreaLambda)
    | //...
```

Callables: Function Object

```
struct CCalcArea
{
    double operator()(double radius) const;
};

CCalcArea calcAreaFunctionObject;
auto vecInput = vector{1.5,2.0,2.5};
auto viewOutput = vecInput
    | views::transform(calcAreaFunctionObject)
    | //...
```

Callables: std::function

```
#include <functional>

function<double(double)> fAnyFuncDblInDblRet {calcArea};

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

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

Callables: Template Parameter

```
template<class TCallable>

void foo(TCallable&& fCallable)
{
    auto vecInput = vector{1.5,2.0,2.5};
    auto viewOutput = vecInput
        | views::transform(forward<TCallable>(fCallable))
        | //...
}
```

Callables: Template Parameter

```
template<class TCallable>
    requires invocable<TCallable,double> &&
        is_same_v<double, invoke_result_t<TCallable,double>>
void foo(TCallable&& fCallable)
{
    auto vecInput = vector{1.5,2.0,2.5};
    auto viewOutput = vecInput
        | views::transform(forward<TCallable>(fCallable))
        | //...
}
```

Callables: Summary

Inline Code /
Pick Overload

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

Extra
Param

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

Member of
Passed Value

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

Pass value
to Member

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

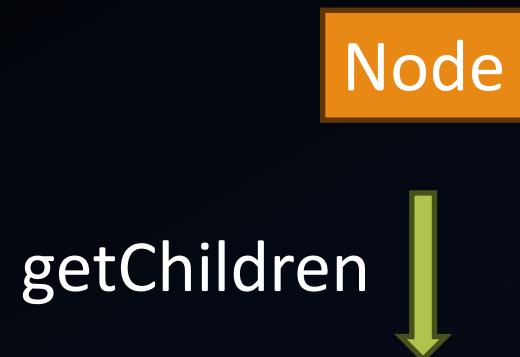
Inject
callable

```
function<double(double)> fCalcArea = calcArea;
```

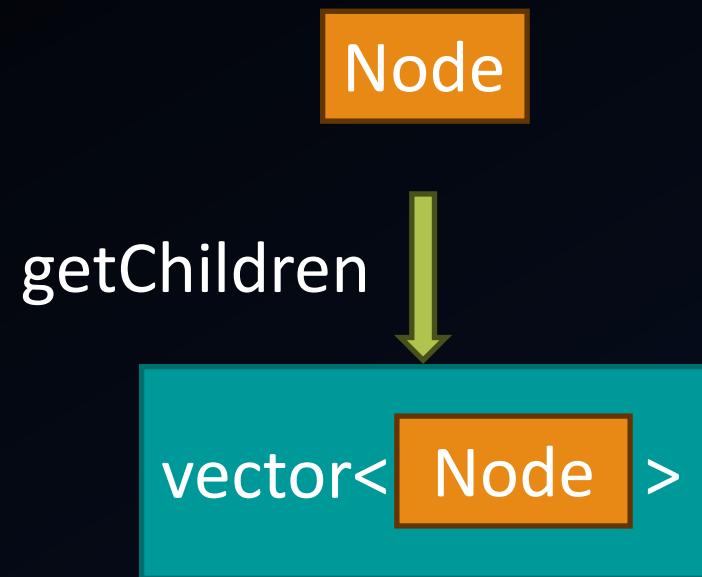
Monads

FUNCTIONS+JOIN

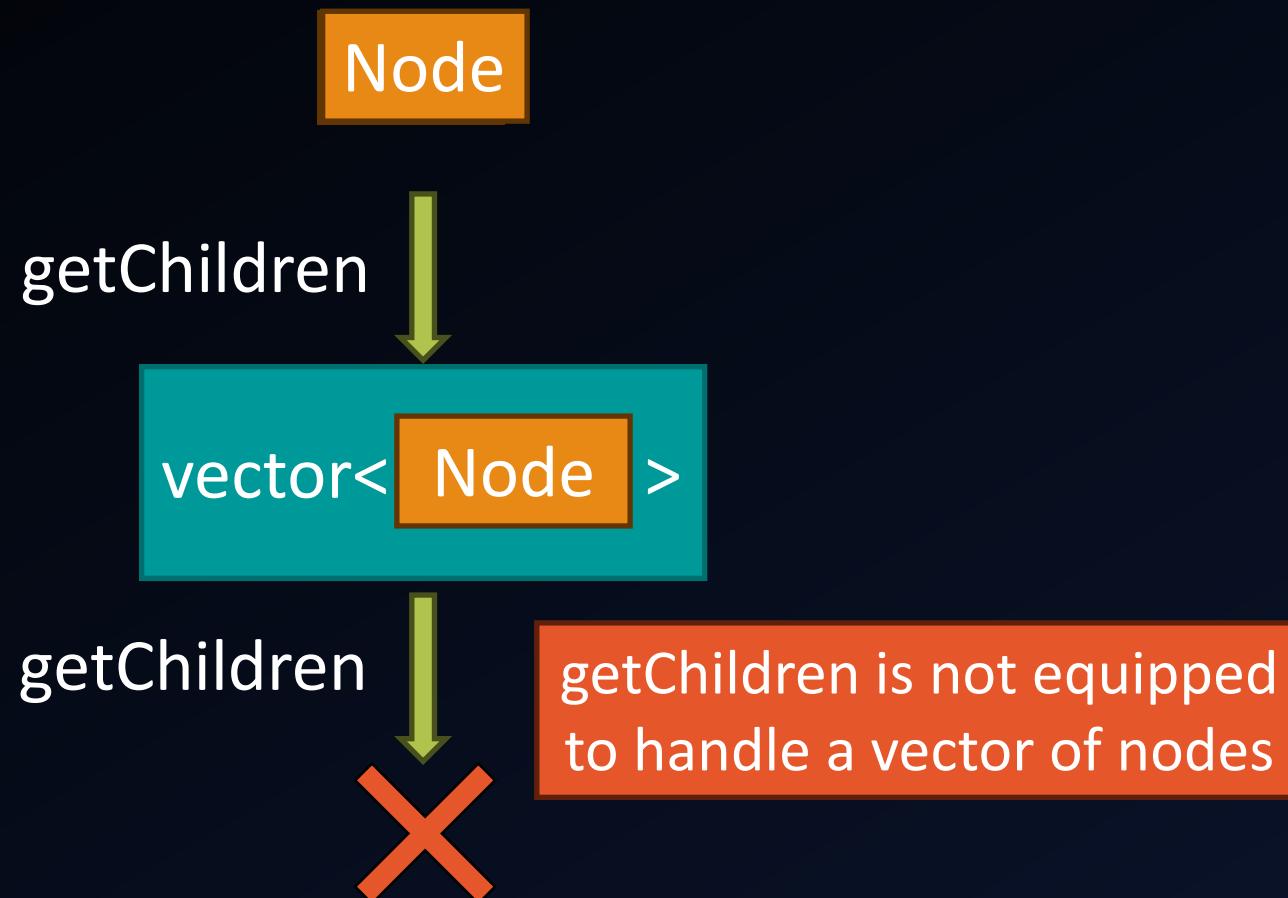
Getting the Children of a Node



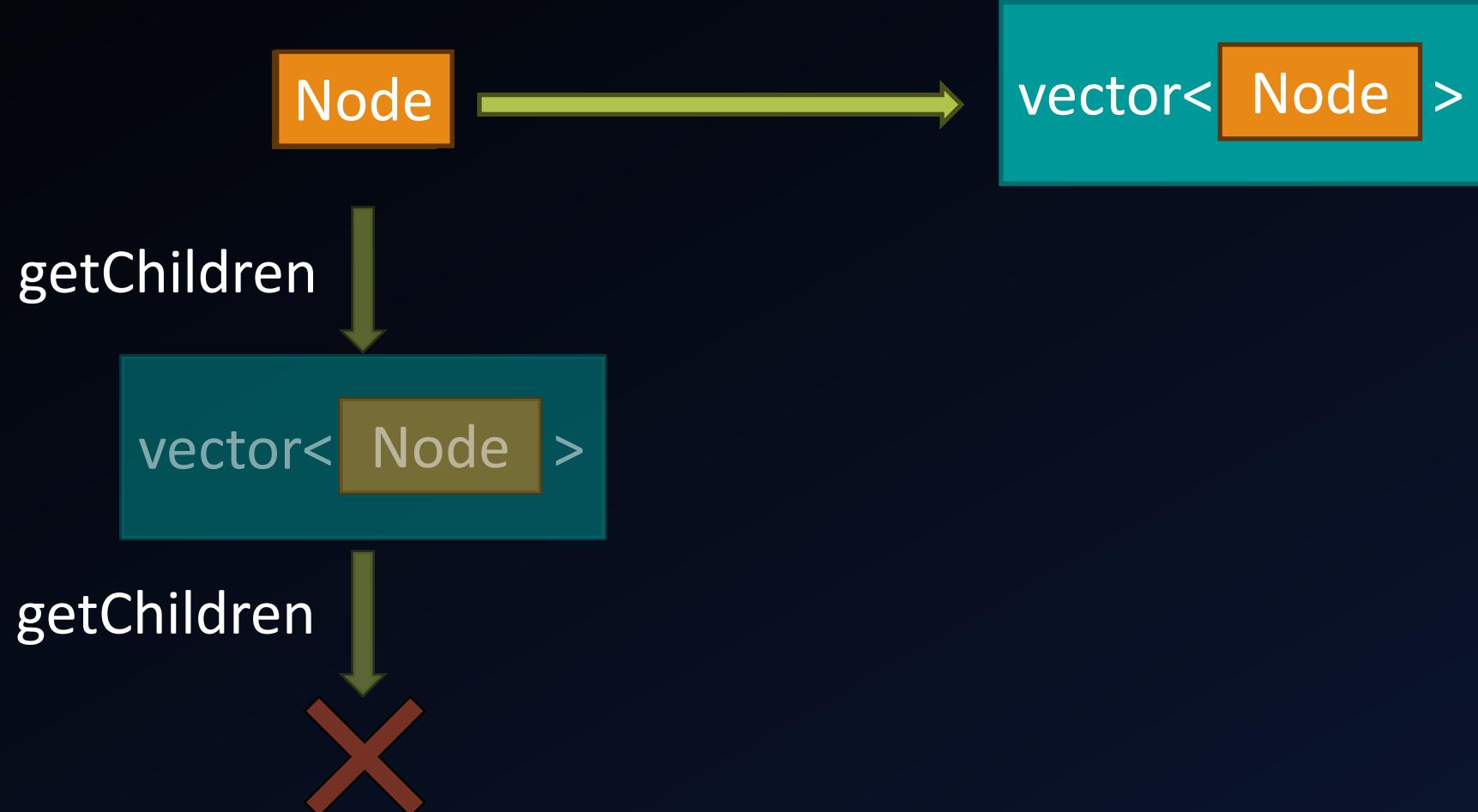
Getting the Children of a Node



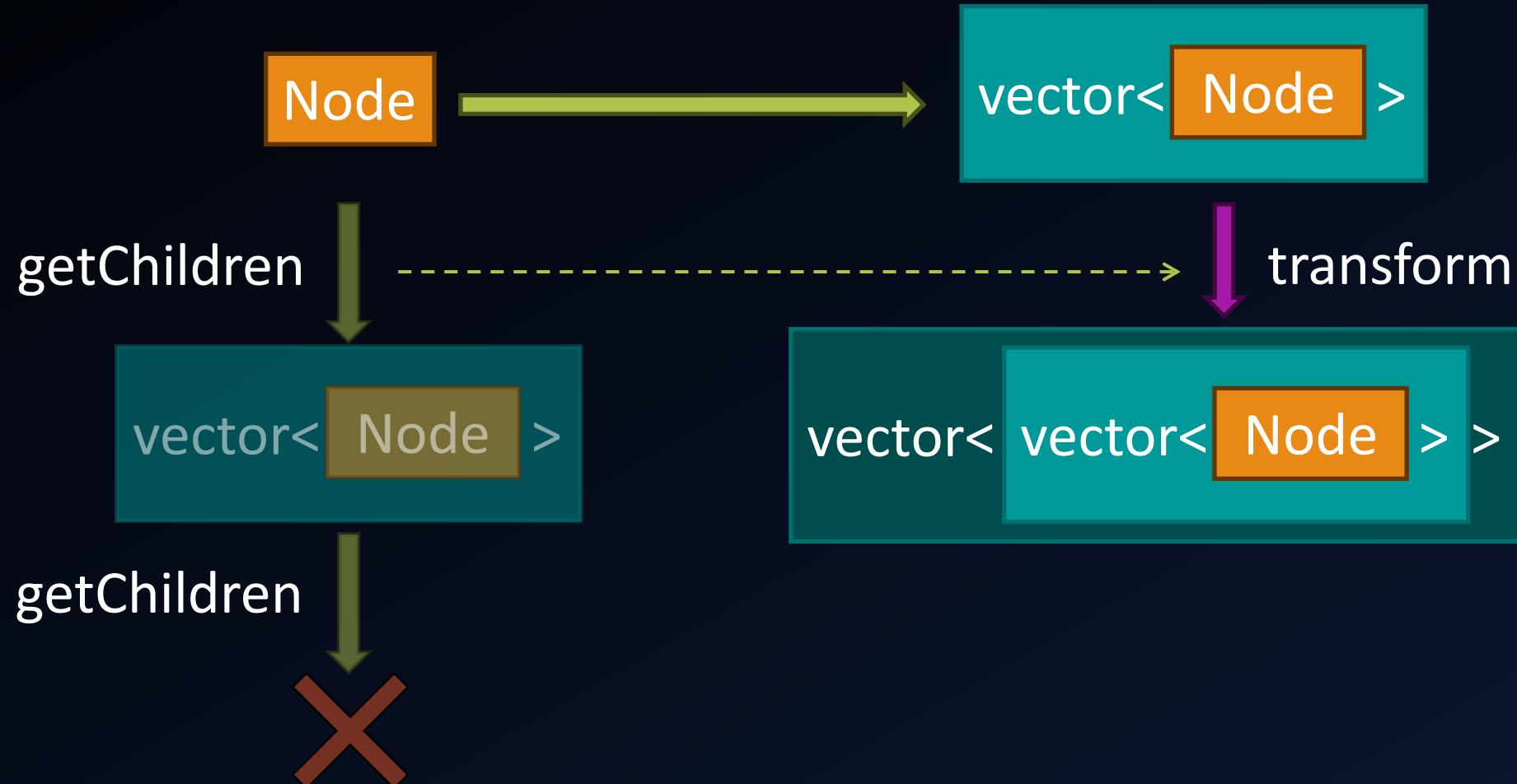
Getting the Children of a Node



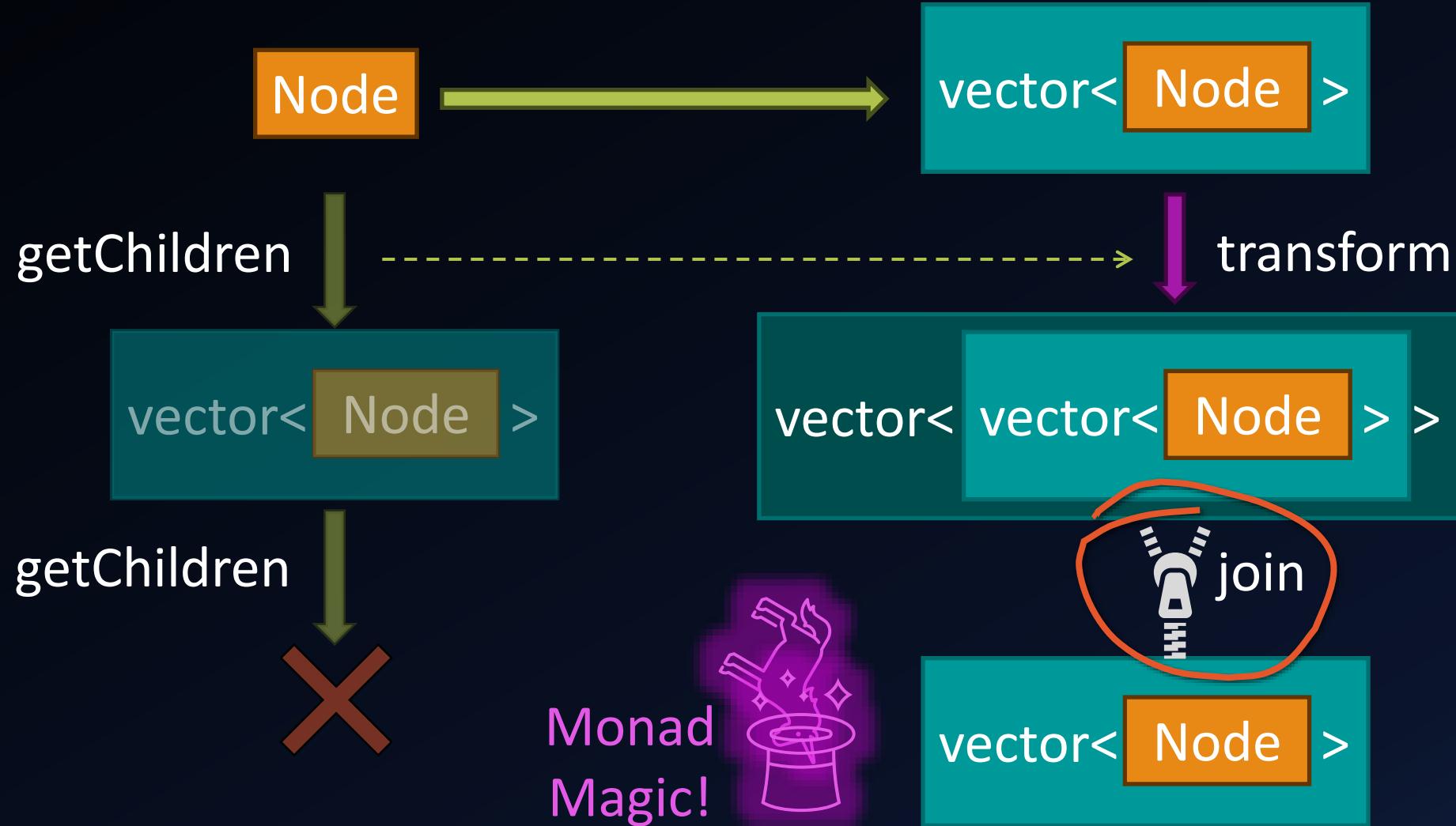
Let's try a Functor Approach



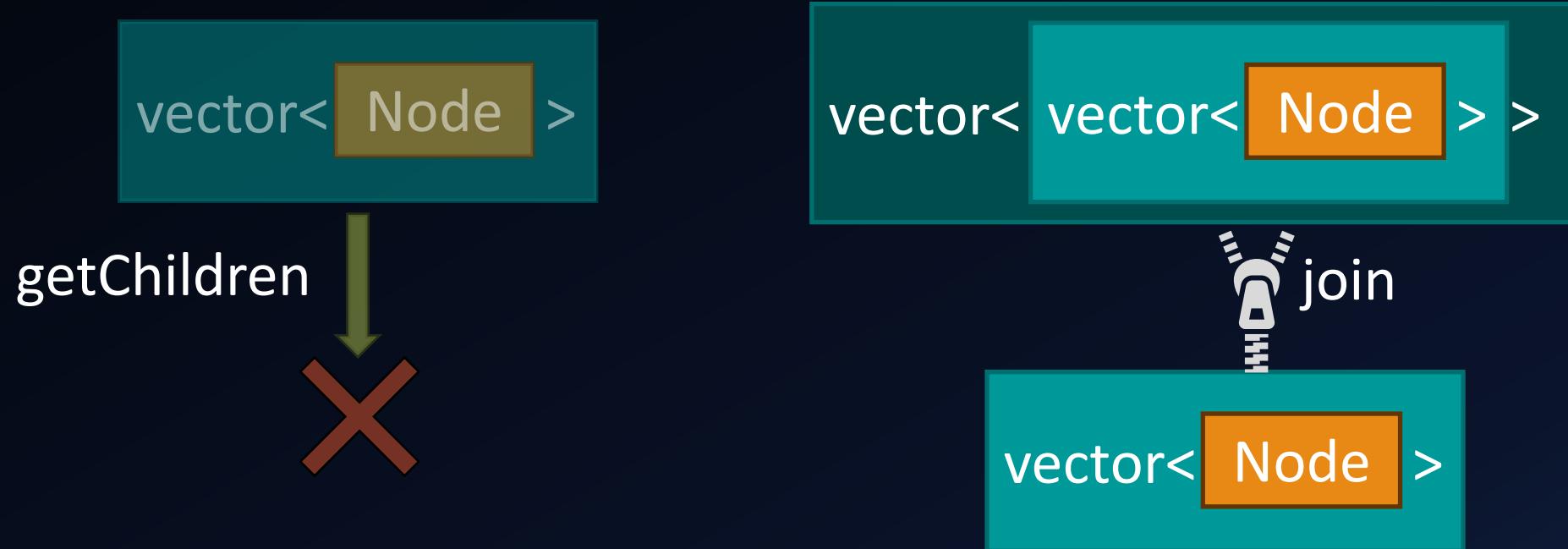
Let's try a Functor Approach



Monad \approx Functor + Join

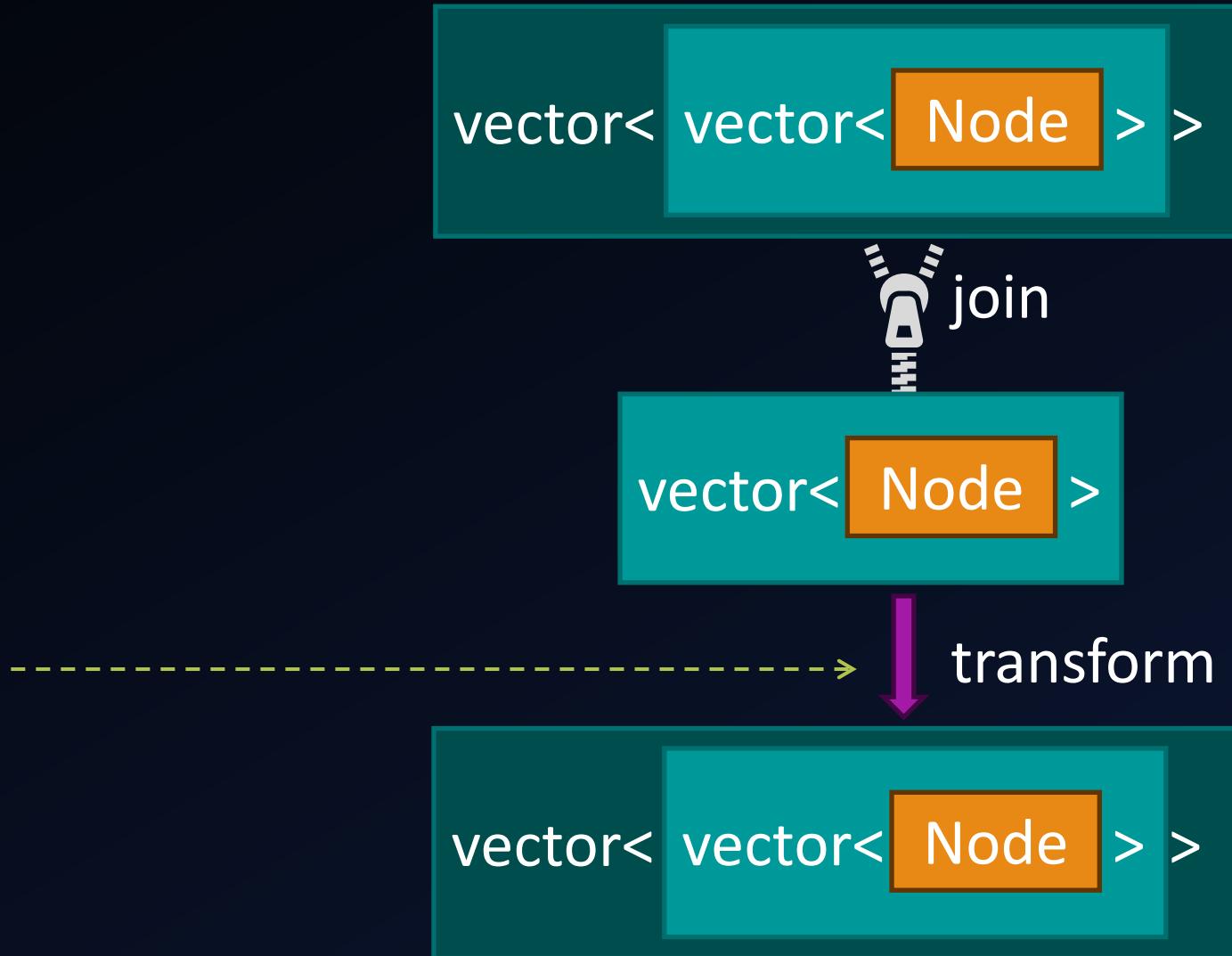


Monad \approx Functor + Join



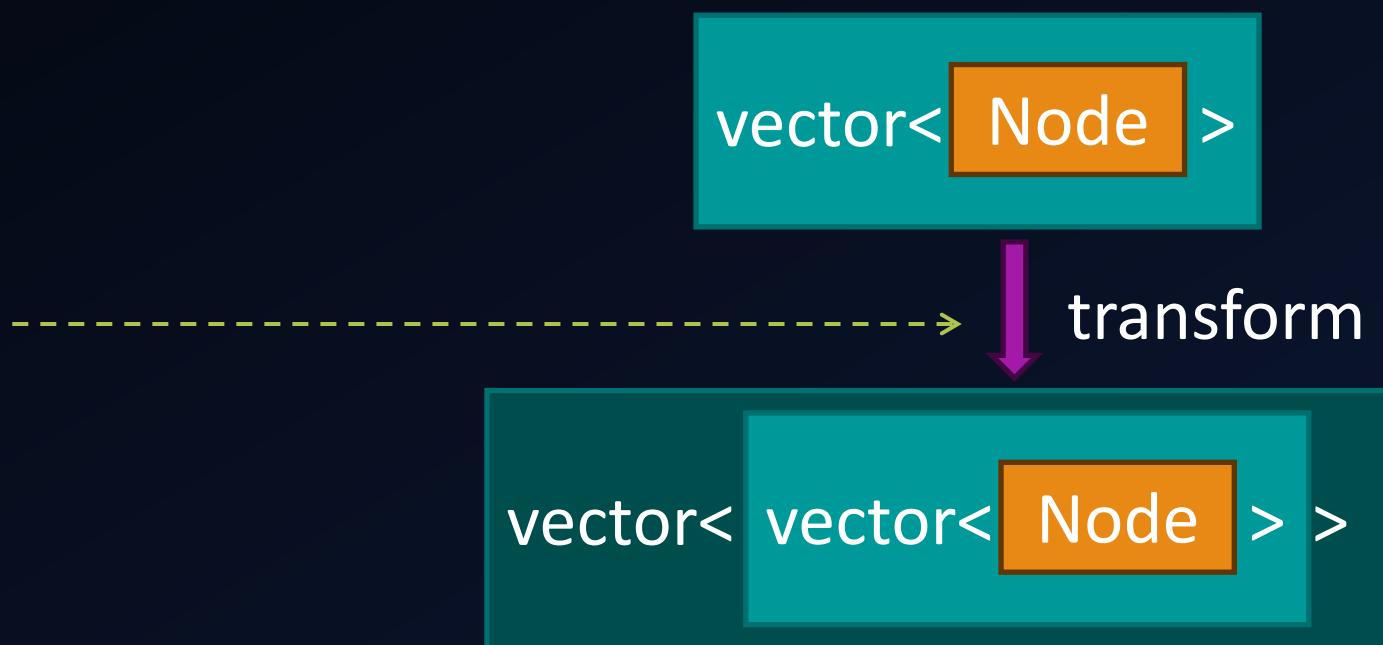
Monad \approx Functor + Join

getChildren



Monad \approx Functor + Join

getChildren



Monad \approx Functor + Join

getChildren



vector< Node >

Node

transform

vector< vector< Node > >

Node

join

vector< Node >

Node

For our purposes a
monad
is a
functor
with the ability to
unwrap
one level of
nesting



Typically cannot unwrap
the last level of nesting

Rules about composition and
identity still apply

'Transform' and 'Join' are sometimes
combined into one function

A Ranges / Views Monad

LET'S JOIN THEM

Printing Diagnostics: Conversion Functions

```
void printDiagnostic(const CDiagnostic& info);  
vector<CDiagnostic> compile(const CFile& input);  
vector<CFile> getFilesInProject(const CProject& input);
```

Printing Diagnostics: Conversion Functions

```
void printDiagnostic(const CDiagnostic& info);  
vector<CDiagnostic> compile(const CFile& input);  
vector<CFile> getFilesInProject(const CProject& input);
```

Printing Diagnostics: Conversion Functions

```
void printDiagnostic(const CDiagnostic& info);  
vector<CDiagnostic> compile(const CFile& input);  
vector<CFile> getFilesInProject(const CProject& input);
```

Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
    vector<CFile> files = getFilesInProject(project);

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

Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
    vector<CFile> files = getFilesInProject(project);

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

Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
    vector<CFile> files = getFilesInProject(project);

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

Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
    vector<CFile> files = getFilesInProject(project);

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

Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
    vector<CFile> files = getFilesInProject(project);

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

Printing Diagnostics: Classic Loop

```
for(const auto& project : projects)
{
    vector<CFile> files = getFilesInProject(project);

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

Printing Diagnostics: Ranges/View Monad

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

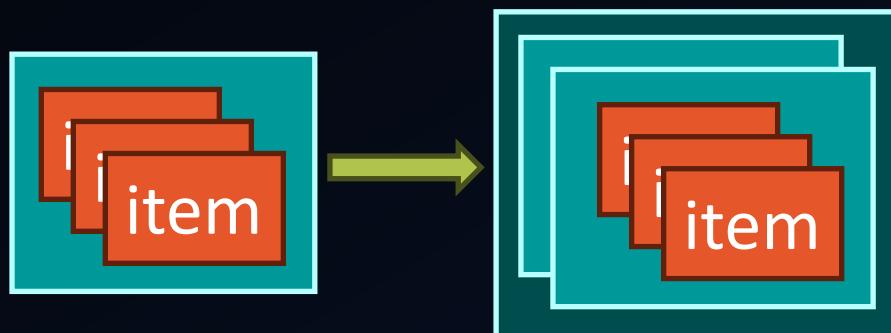
ranges::for_each(diagnostics, printDiagnostic);
```



Printing Diagnostics: Ranges/View Monad

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

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



Printing Diagnostics: Ranges/View Monad

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

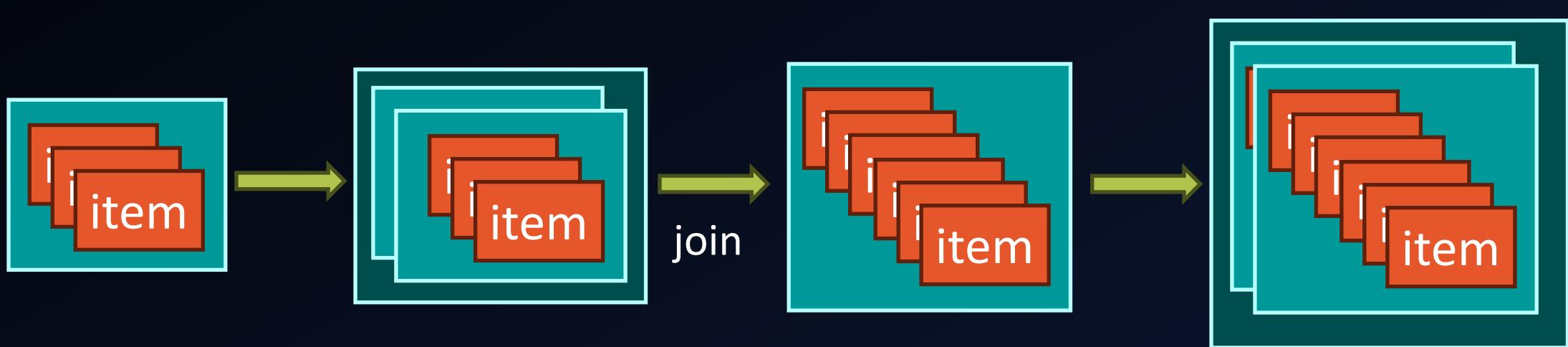
ranges::for_each(diagnostics, printDiagnostic);
```



Printing Diagnostics: Ranges/View Monad

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

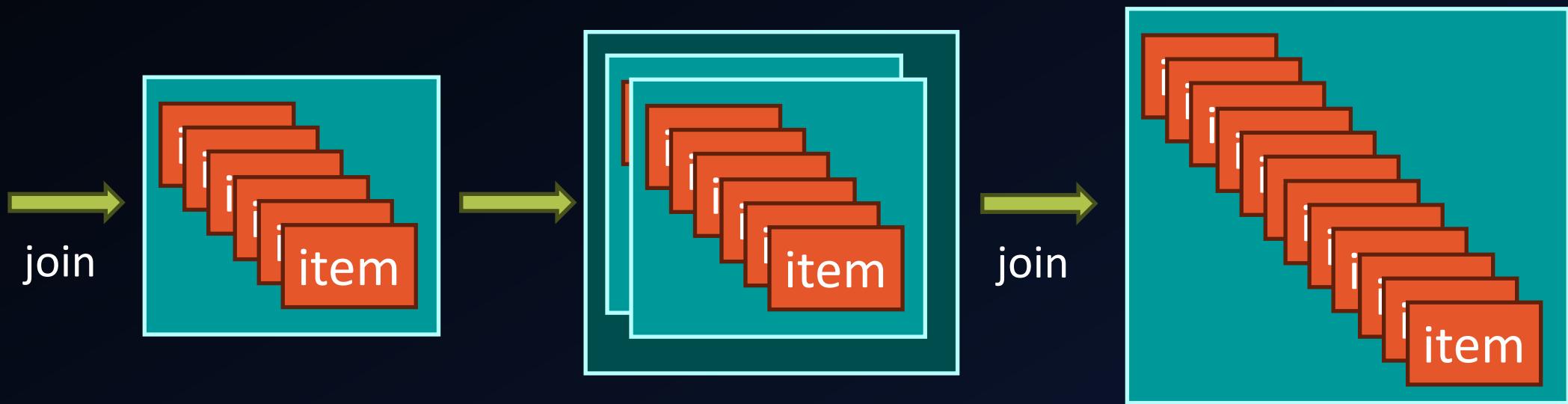
ranges::for_each(diagnostics, printDiagnostic);
```



Printing Diagnostics: Ranges/View Monad

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

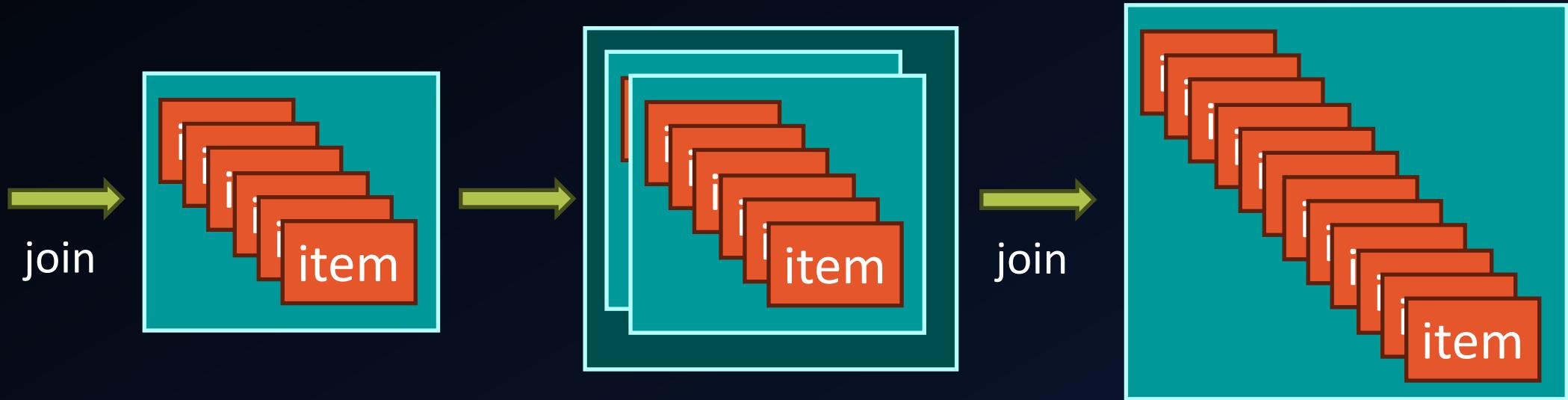
ranges::for_each(diagnostics, printDiagnostic);
```



Printing Diagnostics: Ranges/View Monad

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

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



Printing Diagnostics: Code Comparison

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

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

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

Pure Functions

AVOIDING TRAPS

A View of Multiplied Numbers

```
inline auto getMultipliedView(auto&& input, int multiplier)
{
    auto fMultiply = [&](const int number)
    {
        return number * multiplier;
    };

    return views::transform(input, fMultiply);
}

void test()
{
    const auto input = vector{2,4,6};
    auto multipliedView = getMultipliedView(input, 2);
    // Print output
}
```

A View of Multiplied Numbers

```
inline auto getMultipliedView(auto&& input, int multiplier)
{
    auto fMultiply = [&](const int number)
    {
        return number * multiplier;
    };

    return views::transform(input, fMultiply);
}

void test()
{
    const auto input = vector{2,4,6};
    auto multipliedView = getMultipliedView(input, 2);
    // Print output
}
```

A View of Multiplied Numbers

```
inline auto getMultipliedView(auto&& input, int multiplier)
{
    auto fMultiply = [&](const int number)
    {
        return number * multiplier;
    };

    return views::transform(input, fMultiply);
}

void test()
{
    const auto input = vector{2,4,6};
    auto multipliedView = getMultipliedView(input, 2);
    // Print output
}
```

A View of Multiplied Numbers

```
inline auto getMultipliedView(auto&& input, int multiplier)
{
    auto fMultiply = [&](const int number)
    {
        return number * multiplier;
    };

    return views::transform(input, fMultiply);
}

void test()
{
    const auto input = vector{2,4,6};
    auto multipliedView = getMultipliedView(input, 2);
    // Print output
}
```

A View of Multiplied Numbers

```
inline auto getMultipliedView(auto&& input, int multiplier)
{
    auto fMultiply = [&](const int number)
    {
        return number * multiplier;
    };

    return views::transform(input, fMultiply);
}

void test()
{
    const auto input = vector{2,4,6};
    auto multipliedView = getMultipliedView(input, 2);
    // Print output
}
```

A View of Multiplied Numbers

```
inline auto getMultipliedView(auto&& input, int multiplier)
{
    auto fMultiply = [&](const int number)
    {
        return number * multiplier;
    };

    return views::transform(input, fMultiply);
}

void test()
{
    const auto input = vector{2,4,6};
    auto multipliedView = getMultipliedView(input, 2);
    // Print output
}
```

```
Starting project...
1587538992
-1119889312
467649680
```

An Unexpected Result

```
inline auto getMultipliedView(auto&& input, int multiplier)
{
    auto fMultiply = [&](const int number)
    {
        return number * multiplier;
    };

    return views::transform(input, fMultiply);
}

void test()
{
    const auto input = vector{2,4,6};
    auto multipliedView = getMultipliedView(input, 2);
    // Print output
}
```

The Functor (or Monad) Controls...

...when
your functions
get called

...how often
your functions
get called

...in what context
your functions
get called

How to Avoid Misuse of Functions?

READ THE FINE PRINT



USE PURE FUNCTIONS



How to Avoid Misuse of Functions?

READ THE FINE PRINT



USE PURE FUNCTIONS



What is a Pure Function?

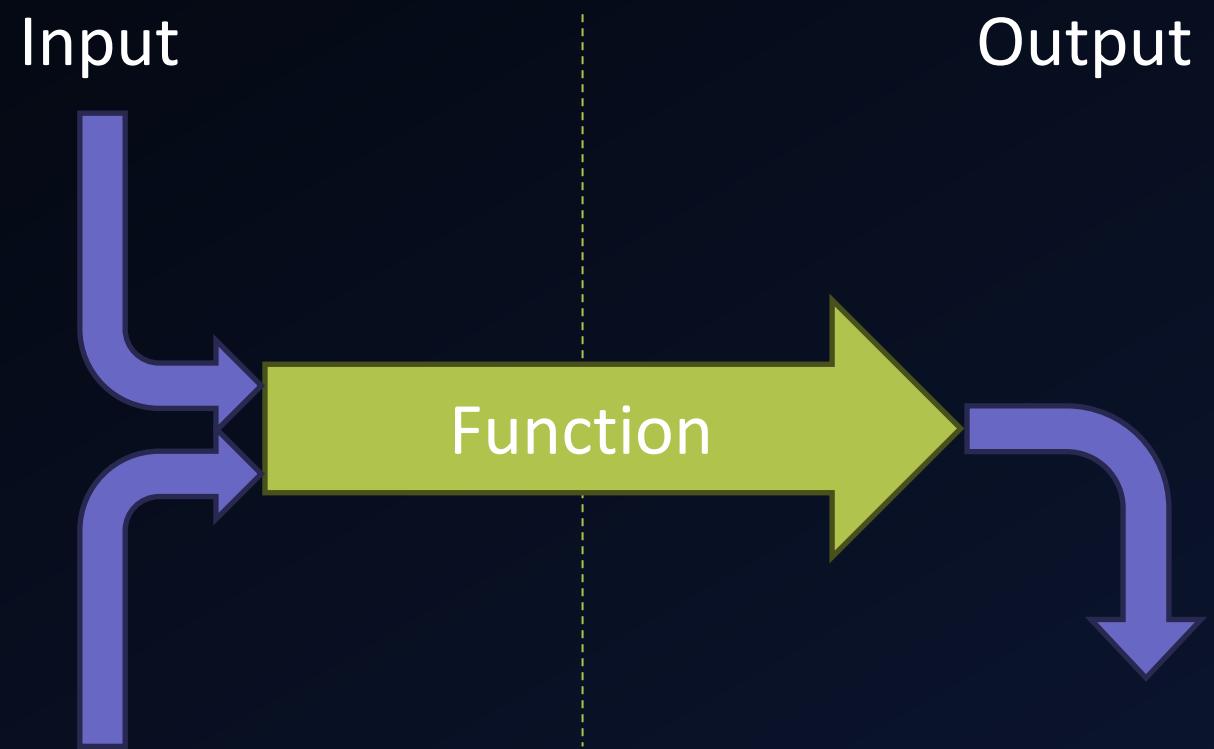
Same Result
for
Same Input

No Side Effects

What is a Pure Function?

Same Result
for
Same Input

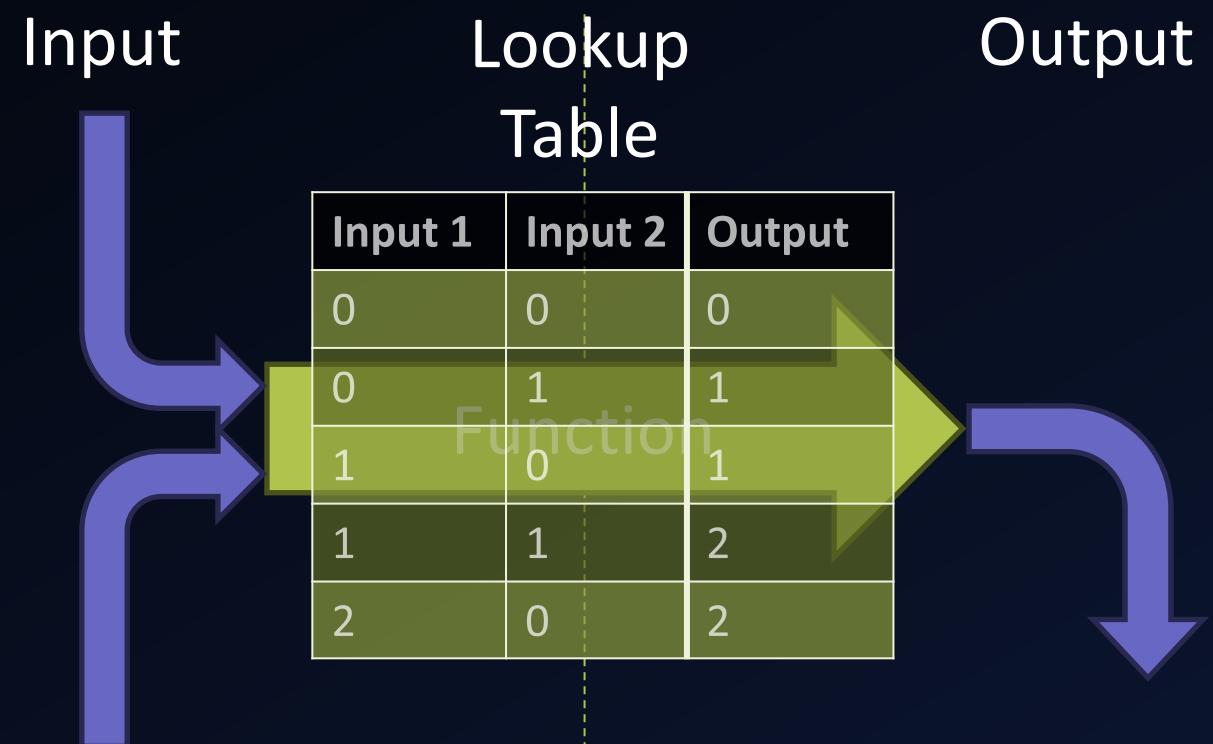
No Side Effects



What is a Pure Function?

Same Result
for
Same Input

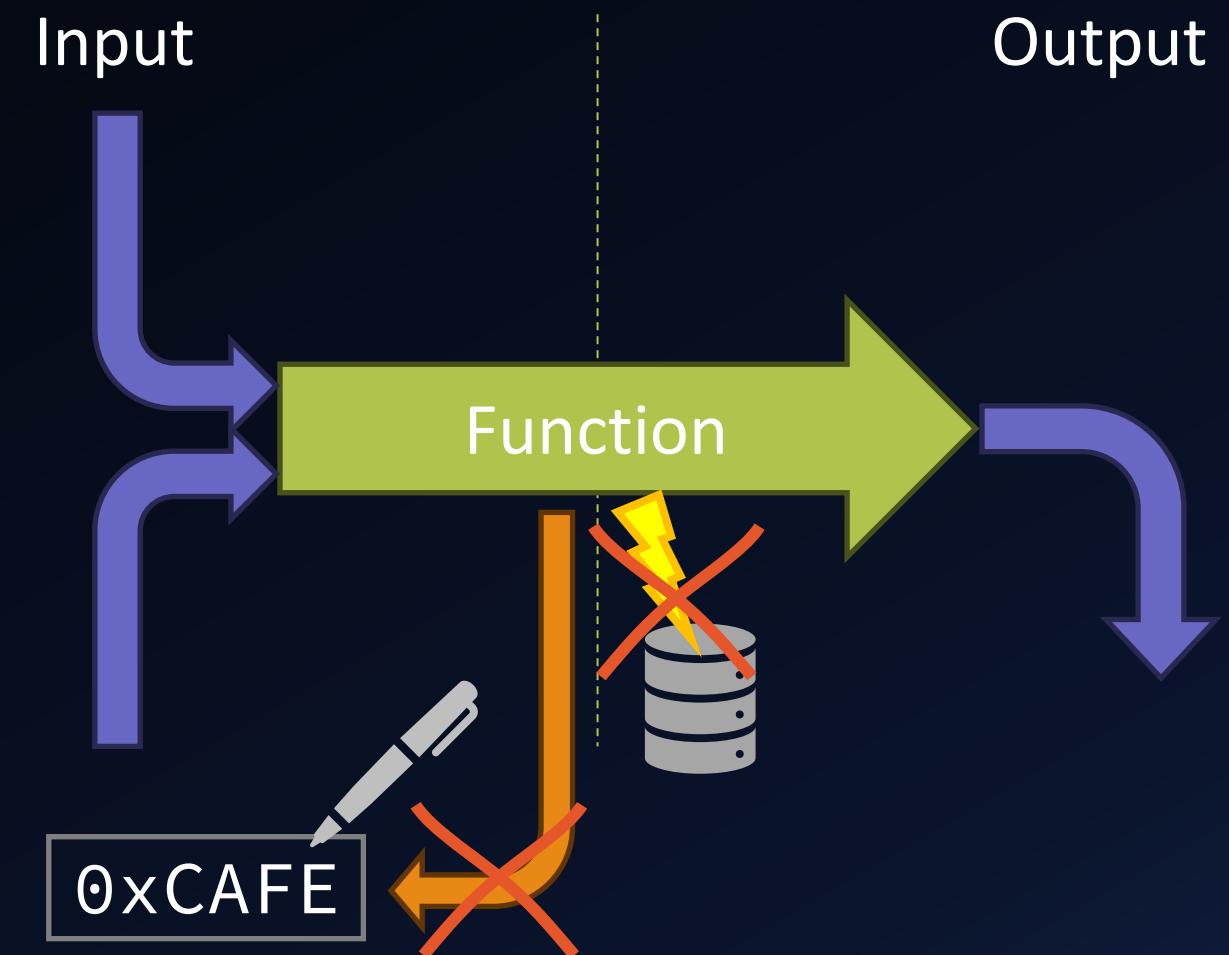
No Side Effects



What is a Pure Function?

Same Result
for
Same Input

No Side Effects



Advantages of Pure Functions

Easy to
reason about

Easy to unit test

Thread-safe

Quiz Time! – Which Function is Pure?

Function	Pure?	Notes
max(int,int)		

Quiz Time! – Which Function is Pure?

Function	Pure?	Notes
max(int,int)	Yes	
fill		

Quiz Time! – Which Function is Pure?

Function	Pure?	Notes
max(int,int)	Yes	
fill	No	Side effects: Operates on passed iterators
chrono::system_clock::now		

Quiz Time! – Which Function is Pure?

Function	Pure?	Notes
max(int,int)	Yes	
fill	No	Side effects: Operates on passed iterators
chrono::system_clock::now	No	Different results: External input
std::sin(double)		

Quiz Time! – Which Function is Pure?

Function	Pure?	Notes
<code>max(int,int)</code>	Yes	
<code>fill</code>	No	Side effects: Operates on passed iterators
<code>chrono::system_clock::now</code>	No	Different results: External input
<code>std::sin(double)</code>	No ^{*)}	Different results: Rounding mode may change
<code>std::abs(int)</code>		

Quiz Time! – Which Function is Pure?

Function	Pure?	Notes
<code>max(int,int)</code>	Yes	
<code>fill</code>	No	Side effects: Operates on passed iterators
<code>chrono::system_clock::now</code>	No	Different results: External input
<code>std::sin(double)</code>	No ^{*)}	Different results: Rounding mode may change
<code>std::abs(int)</code>	?	UB on -INT_MIN (may depend on platform)
<code>std::any_of</code>		

Quiz Time! – Which Function is Pure?

Function	Pure?	Notes
max(int,int)	Yes	
fill	No	Side effects: Operates on passed iterators
chrono::system_clock::now	No	Different results: External input
std::sin(double)	No ^{*)}	Different results: Rounding mode may change
std::abs(int)	?	UB on -INT_MIN (may depend on platform)
std::any_of	?	Depends on predicate

Pure Functions and Reality

Pure functions
help us write
safer programs

Many functions
are somewhat
impure

Find the
level of
purity that
makes sense

An (Almost) Pure Fix

```
inline auto getMultipliedView(auto&& input, int multiplier)
{
    auto fMultiply = [&](const int number)
    {
        return number * multiplier;
    };

    return views::transform(input, fMultiply);
}

void test()
{
    const auto input = vector{2,4,6};
    auto multipliedView = getMultipliedView(input, 2);
    // Print output
}
```

Not pure
Depends on
outside reference

An (Almost) Pure Fix

```
inline auto getMultipliedView(auto&& input, int multiplier)
{
    auto fMultiply = [multiplier](const int number)
    {
        return number * multiplier;
    };

    return views::transform(input, fMultiply);
}

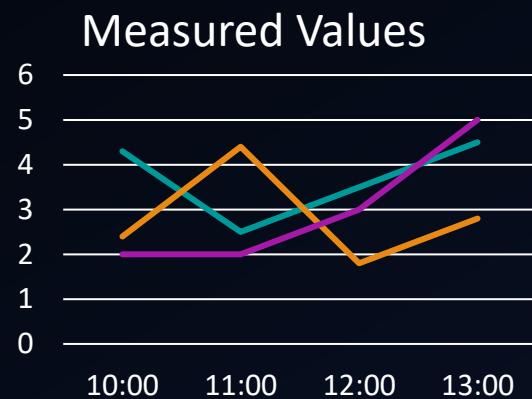
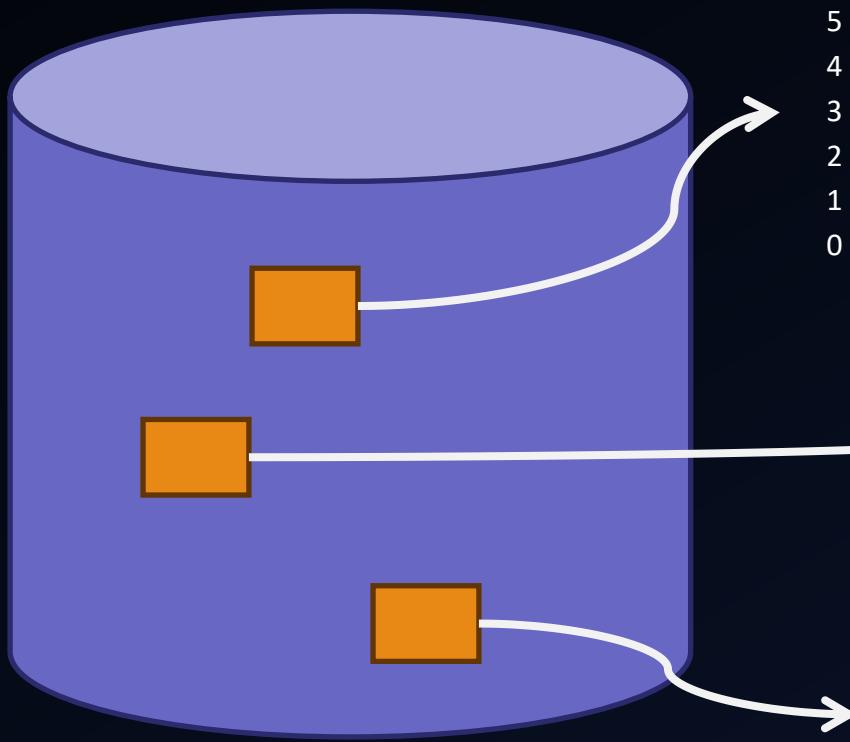
void test()
{
    const auto input = vector{2,4,6};
    auto multipliedView = getMultipliedView(input, 2);
    // Print output
}
```

Maybe not really pure,
but **pure enough**

Handling Failure

THE OPTIONAL AND EXPECTED MONADS

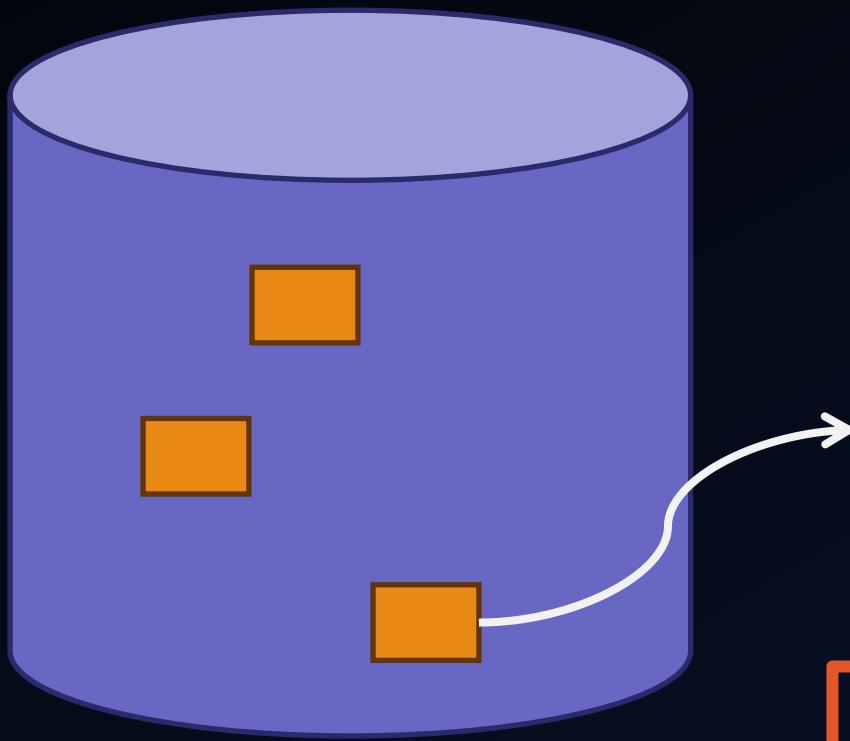
Handling Failure



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

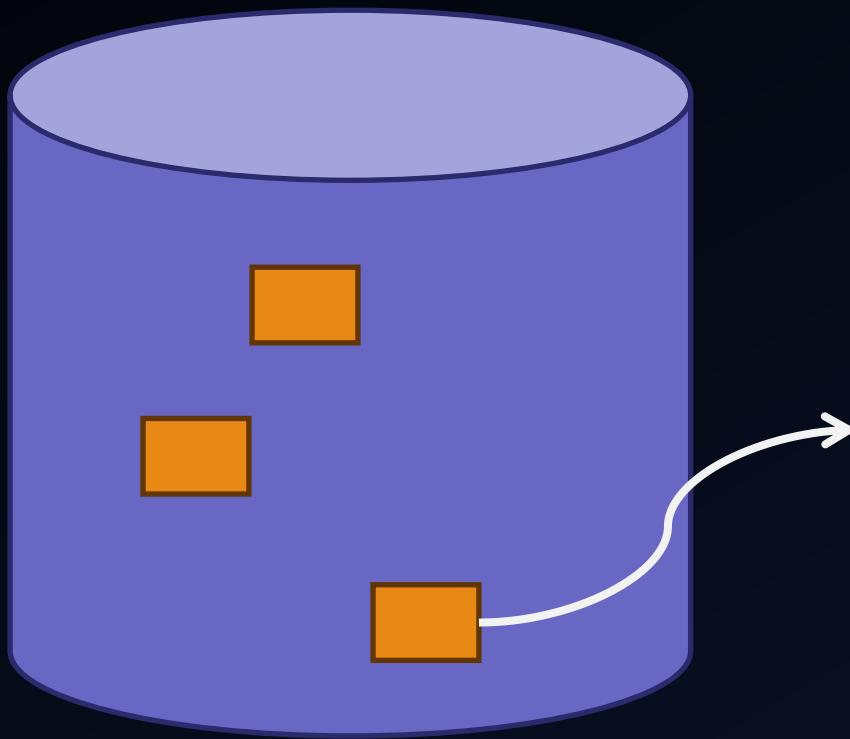


Handling Failure



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

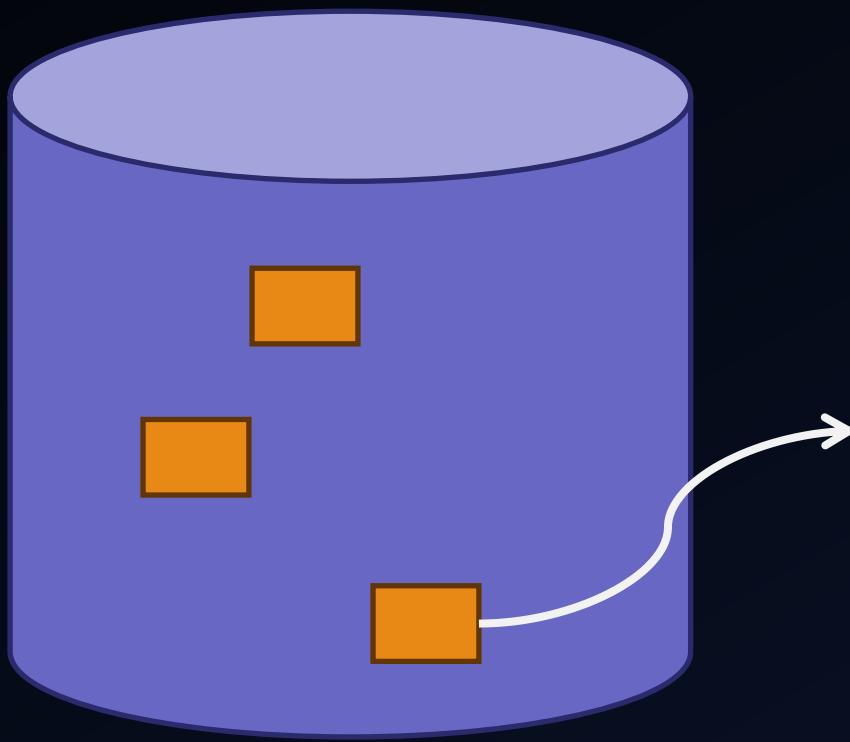
Handling Failure



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

A thought bubble containing the text "Number?" is positioned above the Statuette row. A red rectangle highlights the "-8" value in the Profit column of the Statuette row. Three small circles are positioned near the bottom right corner of the table area.

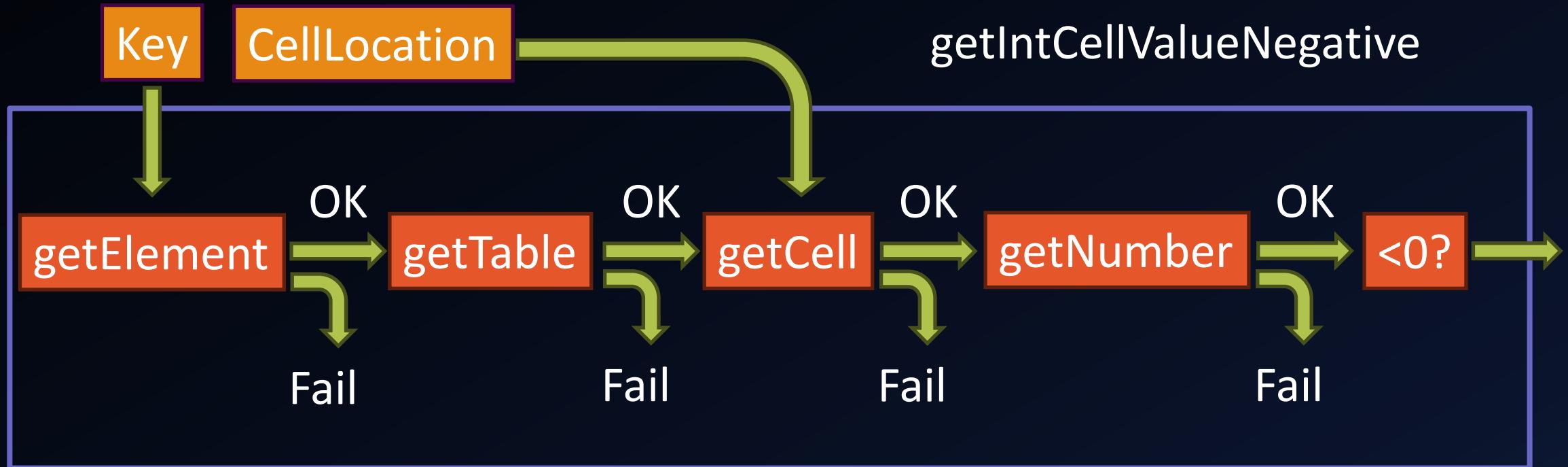
Handling Failure



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

Negative?

Handling Failure



Handling Failure: Classic Way

```
bool getElement(CDb db, CElementKey key, CElement& out);  
bool getTable(CElement element, CTable& out);  
bool getCell(CTable table, CLocation location CCell& out);  
bool getNumericCellValue(CCell cell, int& out);
```

Handling Failure: Classic Way

```
bool getElement(CDb db, CElementKey key, CElement& out);  
bool getTable(CElement element, CTable& out);  
bool getCell(CTable table, CLocation location CCell& out);  
bool getNumericCellValue(CCell cell, int& out);
```

Handling Failure: Classic Way

```
bool getElement(CDb db, CElementKey key, CElement& out);  
bool getTable(CElement element, CTable& out);  
bool getCell(CTable table, CLocation location CCell& out);  
bool getNumericCellValue(CCell cell, int& out);
```

Handling Failure: Classic Way

```
bool getElement(CDb db, CElementKey key, CElement& out);  
bool getTable(CElement element, CTable& out);  
bool getCell(CTable table, CLocation location CCell& out);  
bool getNumericCellValue(CCell cell, int& out);
```

Handling Failure: Classic Way

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

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

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

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

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

Handling Failure: Classic Way

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

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

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

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

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

Handling Failure: Classic Way

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

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

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

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

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

Handling Failure: Classic Way

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

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

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

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

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

Handling Failure: Classic Way

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

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

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

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

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

Handling Failure: The Optional Monad

```
bool           getElement(CDb db, CElementKey key, CElement& out);  
bool           getTable(CElement element, CTable& out);  
bool           getCell(CTable table, CLocation location CCell& out);  
bool           getNumericCellValue(CCell cell, int& out);
```

Handling Failure: The Optional Monad

```
optional<CElement> getElement(CDb db, CElementKey key);  
optional<CTable>    getTable(CElement element);  
optional<CCell>     getCell(CTable tableData, CLocation location);  
optional<int>       getNumericCellValue(CCCell cell);
```

Handling Failure: The Optional Monad

```
optional<CElement> getElement(CDb db, CElementKey key);  
optional<CTable>    getTable(CElement element);  
optional<CCell>     getCell(CTable tableData, CLocation location);  
optional<int>       getNumericCellValue(CCCell cell);  
  
{  
    //...  
    if /* Key not found */  
    {  
        return{}; // or: return nullopt;  
    }  
    CElement elem = //...  
    return elem;  
}
```

Handling Failure: The Optional Monad

```
optional<CElement> getElement(CDb db, CElementKey key);  
optional<CTable>    getTable(CElement element);  
optional<CCell>     getCell(CTable tableData, CLocation location);  
optional<int>       getNumericCellValue(CCCell cell);  
  
{  
    //...  
    if /* Key not found */  
    {  
        return{}; // or: return nullopt;  
    }  
    CElement elem = //...  
    return elem;  
}
```

Handling Failure: The Optional Monad

```
optional<CElement> getElement(CDb db, CElementKey key);  
optional<CTable>   getTable(CElement element);  
optional<CCell>    getCell(CTable tableData, CLocation location);  
optional<int>      getNumericCellValue(CCCell cell);
```

Handling Failure: The Optional Monad

```
optional<CElement> getElement(CDb db, CElementKey key);  
optional<CTable>    getTable(CElement element);  
optional<CCell>     getCell(CTable tableData, CLocation location);  
optional<int>       getNumericCellValue(CCCell cell);
```

Handling Failure: The Optional Monad

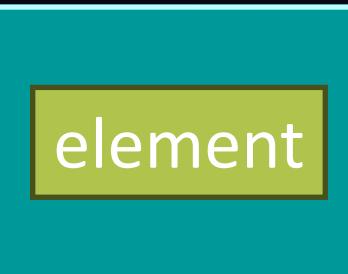
```
optional<CElement> getElement(CDb db, CElementKey key);  
optional<CTable>    getTable(CElement element);  
optional<CCell>     getCell(CTable tableData, CLocation location);  
optional<int>       getNumericCellValue(CCCell cell);
```

Handling Failure: The Optional Monad

```
optional<CElement> getElement(CDb db, CElementKey key);  
optional<CTable>    getTable(CElement element);  
optional<CCell>     getCell(CTable tableData, CLocation location);  
optional<int>        getNumericCellValue(CCCell cell);  
bool                 isNegative(int value);
```

Handling Failure: The Optional Monad

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative);
}
```



element

Handling Failure: The Optional Monad

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative);
}
```



`and_then`

Quiz Time! – What Makes this a Monad?

```
getElement(...) -> optional<CElement>  
getElement(db, key)  
    .and_then(getTable)  
    // ...  
getTable(optional<CElement>) -> optional<CTable>
```

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

```
optional<int> unwrapped =
```

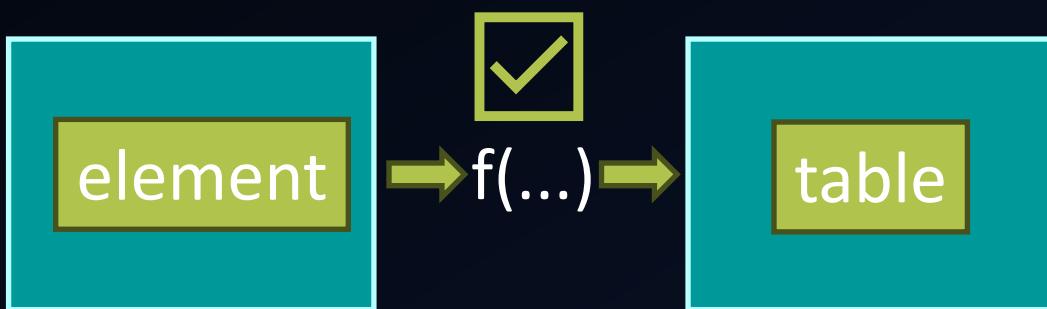
```
wrapped.and_then(identity{});
```

?

identity

Handling Failure: The Optional Monad

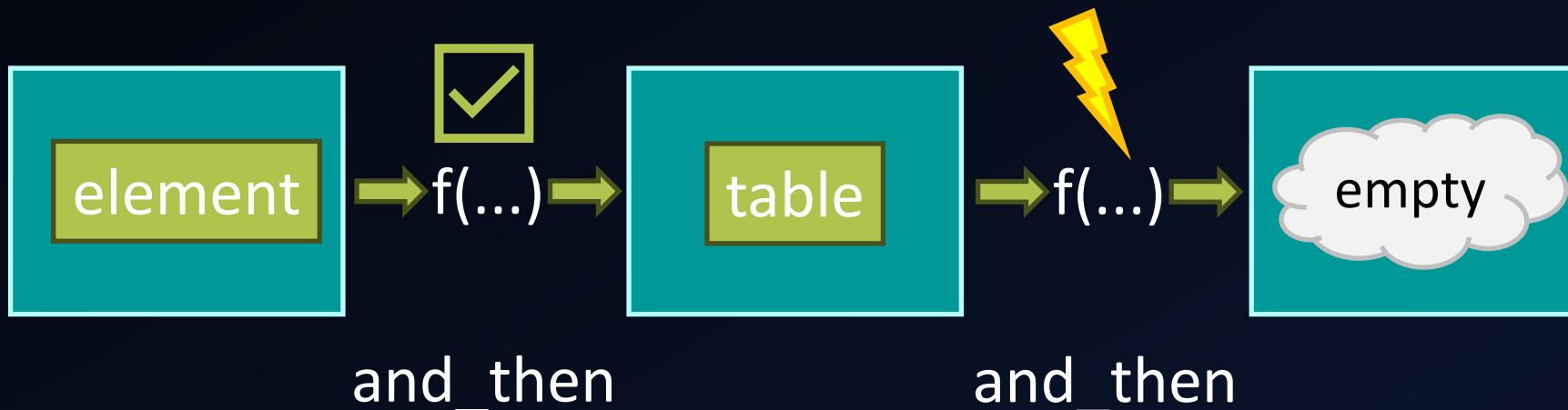
```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative);
}
```



`and_then`

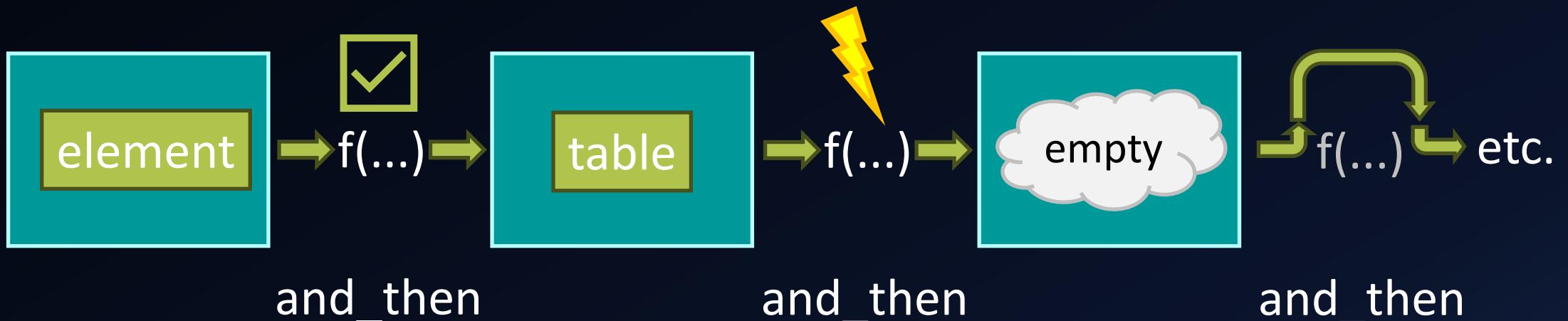
Handling Failure: The Optional Monad

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative);
}
```



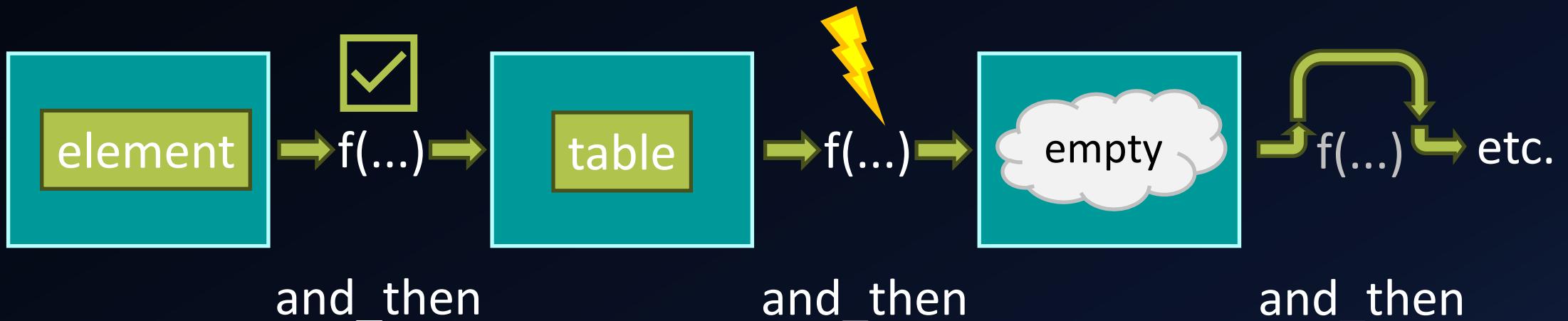
Handling Failure: The Optional Monad

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative);
}
```



Handling Failure: The Optional Monad

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative);
}
```



What can you do with std::optional?

```
optional<int> result = foo();

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

What can you do with std::optional?

```
optional<int> result = foo();

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

What can you do with std::optional?

```
optional<int> result = foo();  
  
if (auto result = foo())  
{  
    auto value = result.value();  
    //...  
}  
else  
{  
    //...  
}
```

What can you do with std::optional?

```
optional<TRet> result = foo();  
  
if (auto result = foo())  
{  
    auto value = foo().value_or(0);  
    //...  
}  
else  
{  
    //...  
}
```

Handling Failure: Code Comparison

```
CElement element;
if ( ! getElement(db, key, element))      { return false; }
CTable table;
if ( ! getTable(element, table))          { return false; }
CCell 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](CTable table)
        { return getCell(table, location); })
    .and_then(getNumericCellValue)
    .transform(isNegative);
```

Catching failure: or_else

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative)
        .or_else(log<bool>);
}
```

Catching failure: or_else

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative)
        .or_else(log<bool>);
}
```

```
template<class TRet>
optional<TRet> log();
```

Catching failure: Lack of Error Context

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative)
        .or_else(log<bool>);
}
```



No error context!

Returning Error State: The Expected Monad

```
optional<CElement> getElement(CDb db, CElementKey key);  
optional<CTable> getTable(CElement element);  
optional<CCell> getCell(CTable tableData, CLocation location);  
optional<int> getNumericCellValue(CCcell cell);  
bool isNegative(int value);
```

Returning Error State: The Expected Monad

```
expected<CElement,CErr> getElement(CDb db, CElementKey key);  
expected<CTable,CErr> getTable(CElement element);  
expected<CCell,CErr> getCell(CTable tableData, CLocation location);  
expected<int,CErr> getNumericCellValue(CCCell cell);  
bool isNegative(int value);
```

Returning Error State: The Expected Monad

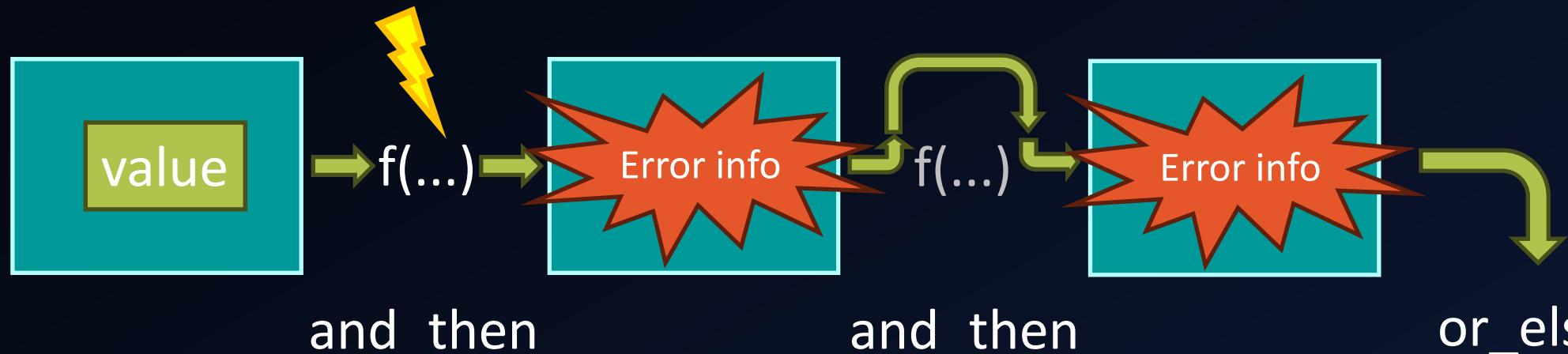
```
expected<CElement, CErr> getElement(CDb db, CElementKey key);  
expected<CTable, CErr> getTable(CElement element);  
expected<CCell, CErr> getCell(CTable tableData, CLocation location);  
expected<int, CErr> getNumericCellValue(CCCell cell);  
bool isNegative(int value);  
  
{  
    //...  
    if /* Key not found */  
    {  
        return unexpected{CErr("Key not found")};  
    }  
    CElement elem = //...  
    return elem;  
}
```

Returning Error State: The Expected Monad

```
expected<CElement, CErr> getElement(CDb db, CElementKey key);  
expected<CTable, CErr> getTable(CElement element);  
expected<CCell, CErr> getCell(CTable tableData, CLocation location);  
expected<int, CErr> getNumericCellValue(CCCell cell);  
bool isNegative(int value);  
  
{  
    //...  
    if /* Key not found */  
    {  
        return unexpected{CErr("Key not found")};  
    }  
    CElement elem = //...  
    return elem;  
}
```

Returning Error State: The Expected Monad

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative);
}
```



Returning Error State: The Expected Monad

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative)
        .or_else(log<bool>);
}
```

CErr with context

```
template<class TRet>
expected<TRet, CErr> log(CErr errorInfo);
```

Returning Error State: The Expected Monad

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative)
        .or_else(log<bool>);
}
```

```
template<class TRet>
expected<TRet, CErr> log(CErr errorInfo);
```

What can do with std::expected?

```
expected<int, CErr> result = foo();

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

What can do with std::expected?

```
expected<int, CErr> result = foo();  
  
if (result.has_value())  
{  
    auto value = result.value();  
    //...  
}  
else  
{  
    auto err = result.error();  
    //...  
}
```

What can do with std::expected?

```
expected<int, CErr> result = foo();

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

What can do with std::expected?

```
optional<int> result = foo();  
  
if (auto result = foo())  
{  
    auto value = result.value();  
    //...  
}  
else  
{  
    // Cannot access result  
}
```

What can do with std::expected?

```
optional<TRet> result = foo();  
  
if (auto result = foo())  
{  
    auto value = foo().value_or(0);  
    //...  
}  
else  
{  
    //...  
}
```

Transforming the Error Type

```
MyError convertCErrToMyErr(CErr errorInfo);  
//...  
expected<int, CErr> result = foo();  
expected<int, MyError> = result.transform_error(convertCErrToMyErr);
```

Transforming the Error Type

```
MyError convertCErrToMyErr(CErr errorInfo);  
//...  
expected<int, CErr> result = foo();  
expected<int, MyError> = result.transform_error(convertCErrToMyErr);
```

Transforming the Error Type

```
MyError convertCErrToMyErr(CErr errorInfo);  
//...  
expected<int, CErr> result = foo();  
expected<int, MyError> = result.transform_error(convertCErrToMyErr);
```

Transforming the Error Type

```
MyError convertCErrToMyErr(CErr errorInfo);  
//...  
expected<int, CErr> result = foo();  
expected<int, MyError> = result.transform_error(convertCErrToMyErr);
```

The Default "Monad"

THE OTHER SIDE OF STD::OPTIONAL

Picking First Success: The Default "Monad"

```
optional<ELanguage> getLanguageFromCommandLine();  
optional<ELanguage> getLanguageFromRegistry();  
optional<ELanguage> getLanguageFromEnvironment();  
//Fallback: ELanguage::English
```

Picking First Success: The Default "Monad"

```
optional<ELanguage> getLanguageFromCommandLine();  
optional<ELanguage> getLanguageFromRegistry();  
optional<ELanguage> getLanguageFromEnvironment();  
//Fallback: ELanguage::English
```

Picking First Success: The Default "Monad"

```
optional<ELanguage> getLanguageFromCommandLine();  
optional<ELanguage> getLanguageFromRegistry();  
optional<ELanguage> getLanguageFromEnvironment();  
//Fallback: ELanguage::English
```

Picking First Success: The Default "Monad"

```
optional<ELanguage> getLanguageFromCommandLine();  
optional<ELanguage> getLanguageFromRegistry();  
optional<ELanguage> getLanguageFromEnvironment();  
//Fallback: ELanguage::English
```

Picking First Success: The Default "Monad"

```
optional<ELanguage> getLanguageFromCommandLine();  
optional<ELanguage> getLanguageFromRegistry();  
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);
}
```



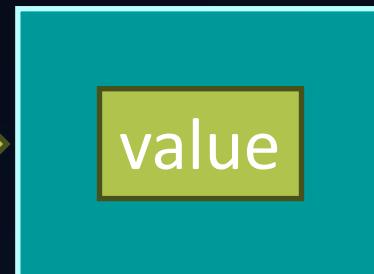
or_else

Picking First Success: The Default "Monad"

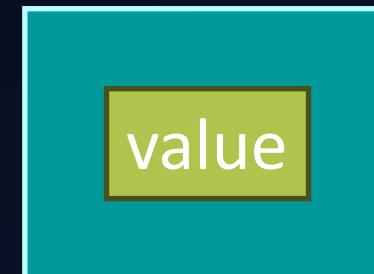
```
ELanguage getStartupLanguage()
{
    return getLanguageFromCommandLine()
        .or_else(getLanguageFromRegistry)
        .or_else(getLanguageFromEnvironment)
        .value_or(ELanguage::English);
}
```



or_else

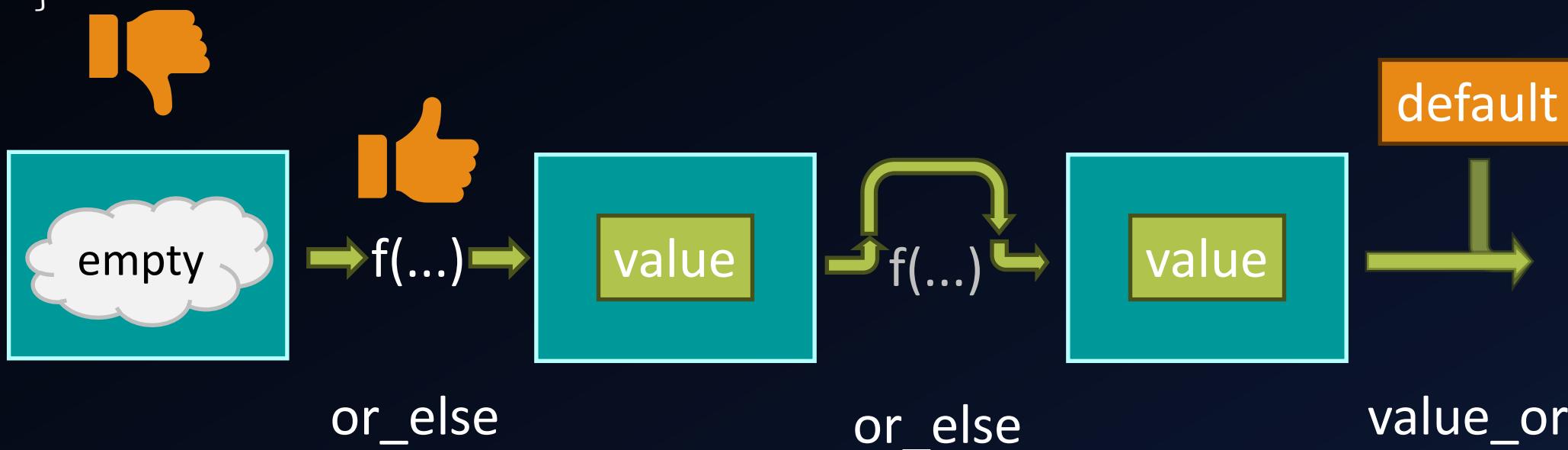


or_else



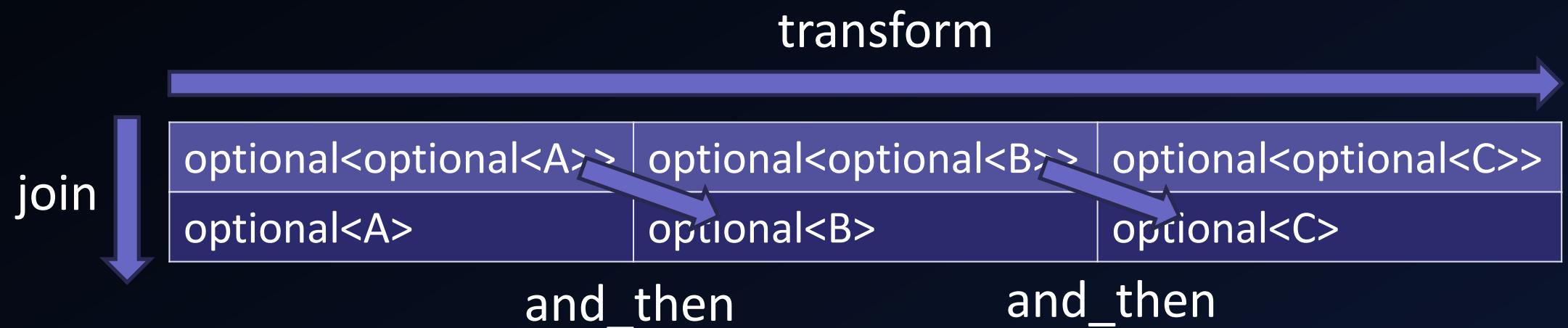
Picking First Success: The Default "Monad"

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



Quiz Time! Why "Monad" in Quotes?

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



Quiz Time! Why "Monad" in Quotes?

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



Compiler Errors

SOME HEURISTICS TO FIX THEM QUICKLY

It Compiles...Not

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

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

```
x.cpp(76): error C3889: call to object of class type
'std::ranges::views::_Transform_fn': no matching call operator found
note: could be 'auto std::ranges::views::_Transform_fn::operator ()(_Rng
&&, _Fn) noexcept(<expr>) const'
note: 'auto std::ranges::views::_Transform_fn::operator ()(_Rng &&, _Fn)
noexcept(<expr>) const': expects 2 arguments - 1 provided
note: or      'auto std::ranges::views::_Transform_fn::operator ()(_Fn &&)
noexcept(<expr>) const'
x.cpp(76): note: 'auto std::ranges::views::_Transform_fn::operator ()(_Fn
&&) noexcept(<expr>) const': could not deduce template argument for '_Fn'
```

It Compiles...Not

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

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

```
x.cpp(76): error C3889: call to object of class type
'std::ranges::views::_Transform_fn': no matching call operator found
note: could be 'auto std::ranges::views::_Transform_fn::operator ()(_Rng
&&,_Fn) noexcept(<expr>) const'
note: 'auto std::ranges::views::_Transform_fn::operator ()(_Rng &&,_Fn)
noexcept(<expr>) const': expects 2 arguments - 1 provided
note: or      'auto std::ranges::views::_Transform_fn::operator ()(_Fn &&)
noexcept(<expr>) const'
x.cpp(76): note: 'auto std::ranges::views::_Transform_fn::operator ()(_Fn
&&) noexcept(<expr>) const': could not deduce template argument for '_Fn'
```

Fixing Compiler Errors: Cheat Sheet

1. Which part of the pipeline causes the error?
2. Are functions overloaded?
3. Do functions accept the correct type?
4. Do functions return the correct type?
5. For ranges::views
 - a) Too many / too few calls to join?
 - b) const views?
6. For optional / expected
 - a) Mixed up and_then with transform?

```
// Split the pipeline
auto s1 = projects |
views::transform(getFilesInProject);
auto s2 = s1 | views::join;
auto s3 = s2 | views::transform(compile);
auto s4 = s3 | views::join;
```

Fixing Compiler Errors: Cheat Sheet

1. Which part of the pipeline causes the error?
2. Are functions overloaded?
3. Do functions accept the correct type?
4. Do functions return the correct type?
5. For ranges::views
 - a) Too many / too few calls to join?
 - b) const views?
6. For optional / expected
 - a) Mixed up and_then with transform?

Use wrapper lambda

Fixing Compiler Errors: Cheat Sheet

1. Which part of the pipeline causes the error?
2. Are functions overloaded?
3. Do functions accept the correct type?
4. Do functions return the correct type?
5. For ranges::views
 - a) Too many / too few calls to join?
 - b) const views?
6. For optional / expected
 - a) Mixed up and_then with transform?



```
// Make parameter types explicit
auto f1 = [](auto radius)
    { return calcArea(radius); };

auto f2 = [](double radius)
    { return calcArea(radius); };
```

Fixing Compiler Errors: Cheat Sheet

1. Which part of the pipeline causes the error?
2. Are functions overloaded?
3. Do functions accept the correct type?
4. Do functions return the correct type?
5. For ranges::views
 - a) Too many / too few calls to join?
 - b) const views?
6. For optional / expected
 - a) Mixed up and_then with transform?

```
// Make return types explicit
auto f1 = [](auto radius)
    { return calcArea(radius); };

auto f2 = [](double radius) -> double
    { return calcArea(radius); };
```

Fixing Compiler Errors: Cheat Sheet

1. Which part of the pipeline causes the error?
2. Are functions overloaded?
3. Do functions accept the correct type?
4. Do functions return the correct type?
5. For ranges::views
 - a) Too many / too few calls to join?
 - b) const views?
6. For optional / expected
 - a) Mixed up and_then with transform?

Function returns Thing
You expected vector<Thing>
(or vice versa)

Fixing Compiler Errors: Cheat Sheet

1. Which part of the pipeline causes the error?
2. Are functions overloaded?
3. Do functions accept the correct type?
4. Do functions return the correct type?
5. For ranges::views
 - a) Too many / too few calls to join?
 - b) const views?
6. For optional / expected
 - a) Mixed up and_then with transform?

Will lead to weird compiler errors.
Just Don't.

Fixing Compiler Errors: Cheat Sheet

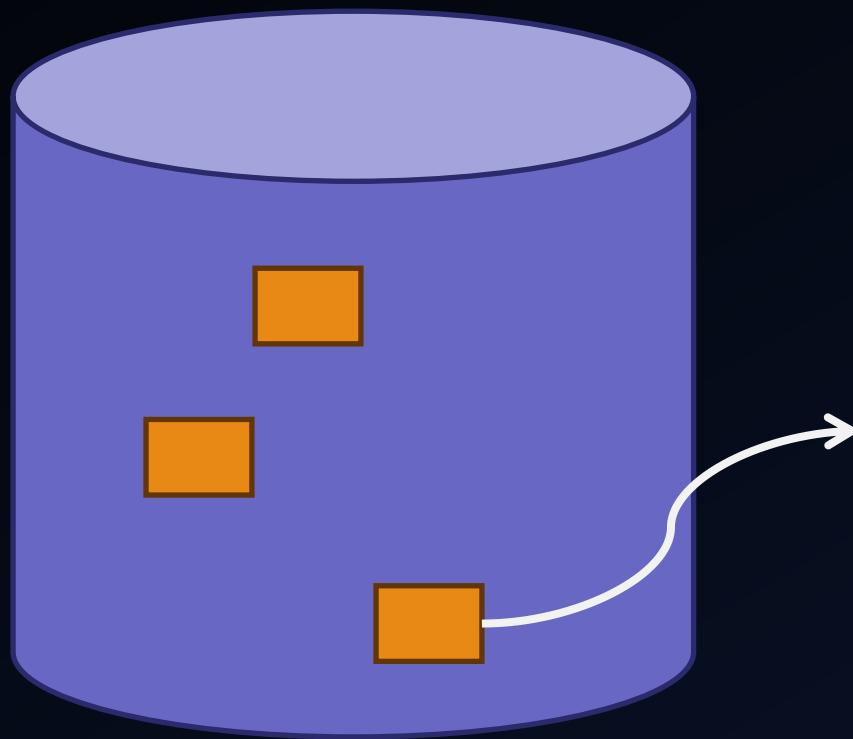
1. Which part of the pipeline causes the error?
2. Are functions overloaded?
3. Do functions accept the correct type?
4. Do functions return the correct type?
5. For ranges::views
 - a) Too many / too few calls to join?
 - b) const views?
6. For optional / expected
 - a) Mixed up and_then with transform?

Function returns Thing
You expected optional<Thing>
(or vice versa)

Writing Your Own Monad

POINTERS INSTEAD OF OPTIONALS

Remember that "getIntCellValueNegative"?



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

A thought bubble containing the text "Negative?" is positioned over the profit value for the Statuette row. A red rectangular box highlights the value "-8". Three green dots are positioned vertically next to the red box, indicating it is a negative value.

Handling Failure: Classic Way...

```
bool           getElement(const CDb& db, CElementKey key, CElement& out);  
bool           getTable(const CElement& element, CTable& out);  
bool           getCell(const CTable& table, CLocation loc, CCell& out);  
bool           getNumericCellValue(const CCell& cell, int& out);
```

...But With Pointers

```
const CElement* getElement(const CDb& db, CElementKey key, CElement& out);  
const CTable* getTable(const CElement& element, CTable& out);  
const CCell* getCell(const CTable& table, CLocation loc, CCell& out);  
optional<int> getNumericCellValue(const CCell& cell, int& out);
```

Handling Failure: Classic Way...

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

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

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

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

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

...But With Pointers

```
std::optional<bool> getIntCellValueNegative
    (const CDb& db, const Key& key, const CLocation& cellLocation)
{
    if (auto* pElement = getElement(db, key))
    {
        if (auto* pTableData = getTable(*pElement))
        {
            if (auto* pCell = getCell(*pTableData, cellLocation))
            {
                if (auto oValue = getNumericCellValue(*pCell))
                {
                    return oValue.value() < 0;
                }
            }
        }
    }
    return {};
}
```

Using the Pointer Monad

```
std::optional<bool> getIntCellValueNegative
    (const CDb& db, const Key& key, const CLocation& cellLocation)
{
    return CPtr(getElementPtr(db, key))
        .and_then(getTablePtr)
        .and_then([cellLocation](const auto& tableData)
            { return getCellPtr(tableData, cellLocation); })
        .and_then(getNumericCellValue)
        .transform([](int value){ return value < 0; });
}
```

Using the Pointer Monad

```
std::optional<bool> getIntCellValueNegative
    (const CDb& db, const Key& key, const CLocation& cellLocation)
{
    return CPtr(getElementPtr(db, key))
        .and_then(getTablePtr)
        .and_then([cellLocation](const auto& tableData)
            { return getCellPtr(tableData, cellLocation); })
        .and_then(getNumericCellValue)
        .transform([](int value){ return value < 0; });
}
```

Using the Pointer Monad

```
std::optional<bool> getIntCellValueNegative
    (const CDb& db, const Key& key, const CLocation& cellLocation)
{
    return CPtr(getElementPtr(db, key))
        .and_then(getTablePtr)
        .and_then([cellLocation](const auto& tableData)
            { return getCellPtr(tableData, cellLocation); })
        .and_then(getNumericCellValue)
        .transform([](int value){ return value < 0; });
}
```

Using the Pointer Monad

```
std::optional<bool> getIntCellValueNegative
    (const CDb& db, const Key& key, const CLocation& cellLocation)
{
    return CPtr(getElementPtr(db, key))
        .and_then(getTablePtr)
        .and_then([cellLocation](const auto& tableData)
            { return getCellPtr(tableData, cellLocation); })
        .and_then(getNumericCellValue)
        .transform([](int value){ return value < 0; });
}
```

Using the Pointer Monad

```
std::optional<bool> getIntCellValueNegative
    (const CDb& db, const Key& key, const CLocation& cellLocation)
{
    return CPtr(getElementPtr(db, key))
        .and_then(getTablePtr)
        .and_then([cellLocation](const auto& tableData)
            { return getCellPtr(tableData, cellLocation); })
        .and_then(getNumericCellValue)
        .transform([](int value){ return value < 0; });
}
```

Using the Pointer Monad

```
std::optional<bool> getIntCellValueNegative  
    (const CDb& db, const Key& key, const CLocation& cellLocation)  
{  
    return CPtr(getElementPtr(db, key))  
        .and_then(getTablePtr)  
        .and_then([cellLocation](const auto& tableData)  
            { return getCellPtr(tableData, cellLocation); })  
        .and_then(getNumericCellValue)  
        .transform([](int value){ return value < 0; });  
}
```

Pointers

optional

Implementing A Pointer Monad

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

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

    //...

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

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

Implementing A Pointer Monad

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

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

    //...

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

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



```
https://godbolt.org/z/hK1oxG8Tx
```

Implementing A Pointer Monad

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

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

    //...

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

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

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

    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 = add_lvalue_reference_t<remove_pointer_t<TPtr>>;
    using TRet = invoke_result_t<TCall, TRef>;

    if constexpr (cIsPointer<TRet>)
    {
        //...return type is a pointer
    }
    else if constexpr (cbIsOptional<TRet>)
    {
        //...return type is an std::optional<something>
    }
    else
    //...
}
```

Implementing A Pointer Monad

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

    if constexpr (cIsPointer<TRet>)
    {
        //...return type is a pointer
    }
    else if constexpr (cbIsOptional<TRet>)
    {
        //...return type is an std::optional<something>
    }
    else
    //...
}
```

Implementing A Pointer Monad

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



```
    if constexpr (cIsPointer<TRet>)
    {
        //...return type is a pointer
    }
    else if constexpr (cbIsOptional<TRet>)
    {
        //...return type is an std::optional<something>
    }
    else
    //...
}
```

Implementing A Pointer Monad

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

    if constexpr (cIsPointer<TRet>)
    {
        //...return type is a pointer
    }
    else if constexpr (cbIsOptional<TRet>)
    {
        //...return type is an std::optional<something>
    }
    else
    //...
}
```

Implementing A Pointer Monad

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

    if constexpr (cIsPointer<TRet>)
    {
        if (m_Ptr)
        {
            return CPtr<TRet>(invoke(forward<TCall>(fInvoke), *m_Ptr));
        }
        return CPtr<TRet>{};
    }
    else if constexpr (cbIsOptional<TRet>)
    {
        //...return type is an std::optional<something>
    }
    else
    //...
}
```

Implementing A Pointer Monad

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

    if constexpr (cIsPointer<TRet>)
    {
        if (m_Ptr)
        {
            return CPtr<TRet>(invoke(forward<TCall>(fInvoke), *m_Ptr));
        }
        return CPtr<TRet>{};
    }
    else if constexpr (cbIsOptional<TRet>)
    {
        //...return type is an std::optional<something>
    }
    else
    //...
}
```

Implementing A Pointer Monad

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

    if constexpr (cIsPointer<TRet>)
    {
        if (m_Ptr)
        {
            return CPtr<TRet>(invoke(forward<TCall>(fInvoke), *m_Ptr));
        }
        return CPtr<TRet>{};
    }
    else if constexpr (cbIsOptional<TRet>)
    {
        //...return type is an std::optional<something>
    }
    else
    //...
}
```

Implementing A Pointer Monad

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

    if constexpr (cIsPointer<TRet>)
    {
        if (m_Ptr)
        {
            return CPtr<TRet>(invoke(forward<TCall>(fInvoke), *m_Ptr));
        }
        return CPtr<TRet>{};
    }
    else if constexpr (cbIsOptional<TRet>)
    {
        //...return type is an std::optional<something>
    }
    else
    //...
}
```

Implementing A Pointer Monad

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

    if constexpr (cIsPointer<TRet>)
    {
        if (m_Ptr)
        {
            return CPtr<TRet>(invoke(forward<TCall>(fInvoke), *m_Ptr));
        }
        return CPtr<TRet>{};
    }
    else if constexpr (cbIsOptional<TRet>)
    {
        //...return type is an std::optional<something>
    }
    else
    //...
}
```

Implementing A Pointer Monad

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

    if constexpr (cIsPointer<TRet>)
    {
        if (m_Ptr)
        {
            return CPtr<TRet>(invoke(forward<TCall>(fInvoke), *m_Ptr));
        }
        return CPtr<TRet>{};
    }
    else if constexpr (cbIsOptional<TRet>)
    {
        //...return type is an std::optional<something>
    }
    else
    //...
}
```

Implementing A Pointer Monad

```
if constexpr (cIsPointer<TRet>)
{
    if (m_Ptr)
    {
        return CPtr<TRet>(invoke(forward<TCall>(fInvoke), *m_Ptr));
    }
    return CPtr<TRet>{};
}
else if constexpr (cbIsOptional<TRet>)
{
    if (m_Ptr)
    {
        return invoke(forward<TCall>(fInvoke), *m_Ptr);
    }
    return remove_cvref_t<TRet>{};
}
else
//...
}
```

```
template<class>
constexpr bool cbIsOptional = false;
template<class T>
constexpr bool cbIsOptional<optional<T>> = true;
```

Implementing A Pointer Monad

```
if constexpr (cIsPointer<TRet>)
{
    if (m_Ptr)
    {
        return CPtr<TRet>(invoke(forward<TCall>(fInvoke), *m_Ptr));
    }
    return CPtr<TRet>{};
}
else if constexpr (cbIsOptional<TRet>)
{
    if (m_Ptr)
    {
        return invoke(forward<TCall>(fInvoke), *m_Ptr);
    }
    return remove_cvref_t<TRet>{};
}
else
//...
}
```

Implementing A Pointer Monad

```
if constexpr (cIsPointer<TRet>)
{
    if (m_Ptr)
    {
        return CPtr<TRet>(invoke(forward<TCall>(fInvoke), *m_Ptr));
    }
    return CPtr<TRet>{};
}
else if constexpr (cbIsOptional<TRet>)
{
    if (m_Ptr)
    {
        return invoke(forward<TCall>(fInvoke), *m_Ptr);
    }
    return remove_cvref_t<TRet>{};
}
else
//...
```

Implementing A Pointer Monad

```
if constexpr (cIsPointer<TRet>)
{
    if (m_Ptr)
    {
        return CPtr<TRet>(invoke(forward<TCall>(fInvoke), *m_Ptr));
    }
    return CPtr<TRet>{};
}
else if constexpr (cbIsOptional<TRet>)
{
    if (m_Ptr)
    {
        return invoke(forward<TCall>(fInvoke), *m_Ptr);
    }
    return remove_cvref_t<TRet>{};
}
else
//...
}
```

Implementing A Pointer Monad

```
if constexpr (cIsPointer<TRet>)
{
    if (m_Ptr)
    {
        return CPtr<TRet>(invoke(forward<TCall>(fInvoke), *m_Ptr));
    }
    return CPtr<TRet>{};
}
else if constexpr (cbIsOptional<TRet>)
{
    if (m_Ptr)
    {
        return invoke(forward<TCall>(fInvoke), *m_Ptr);
    }
    return remove_cvref_t<TRet>{};
}
else
//...
```

Implementing A Pointer Monad

```
else if constexpr (cbIsOptional<TRet>)
{
    if (m_Ptr)
    {
        return invoke(forward<TCall>(fInvoke), *m_Ptr);
    }
    return remove_cvref_t<TRet>{};
}
// else if constexpr (function already returns CPtr) ...
// else if constexpr (function returns std::expected) ...
else
{
    static_assert(false, "returns neither pointer nor optional");
}
```

Using the Pointer Monad

```
std::optional<bool> getIntCellValueNegative  
    (const CDb& db, const Key& key, const CLocation& cellLocation)  
{  
    return CPtr(getElementPtr(db, key))  
        .and_then(getTablePtr)  
        .and_then([cellLocation](const auto& tableData)  
            { return getCellPtr(tableData, cellLocation); })  
        .and_then(getNumericCellValue)  
        .transform([](int value){ return value < 0; });  
}
```

Pointers

optional

Pointer Monad – Summary

Writing monads
is not magic!



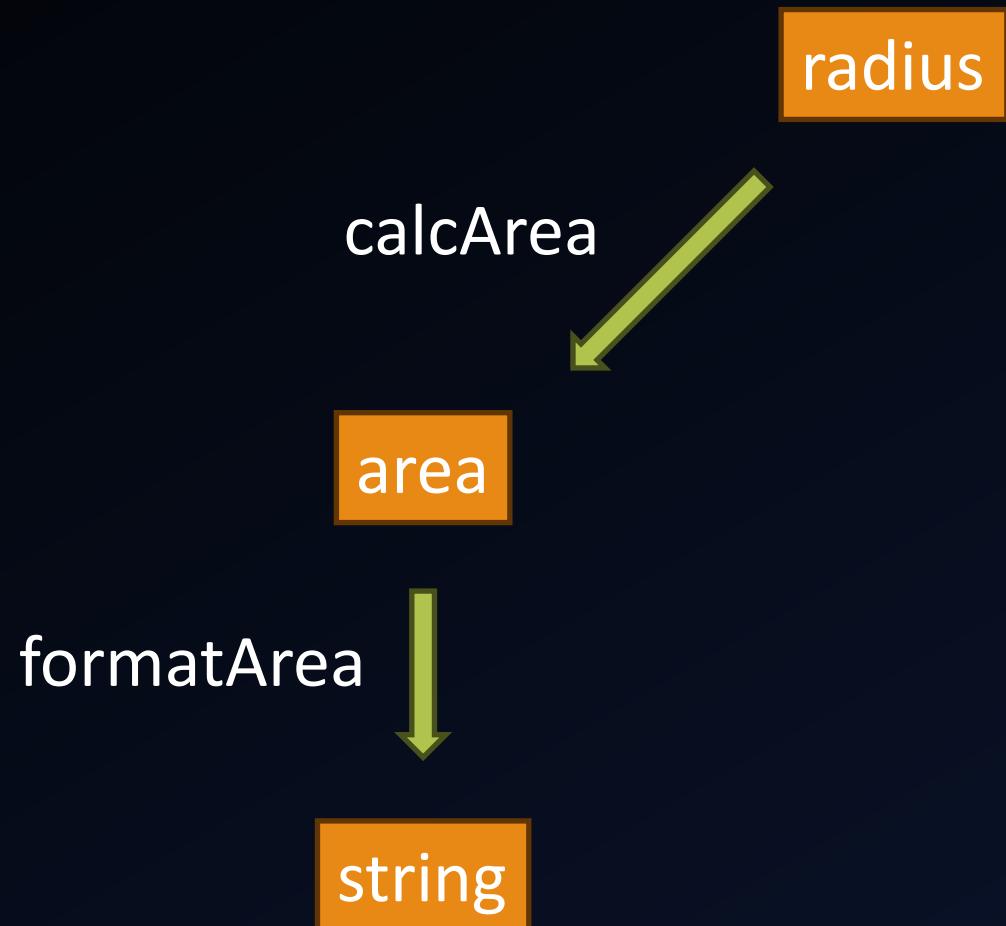
Brush up your
metaprogramming

- Start specific, then generalize
- Consider effects of value categories
(`const`, `&`, ...)
- Use concepts and `static_assert` to prevent misuse at compile time

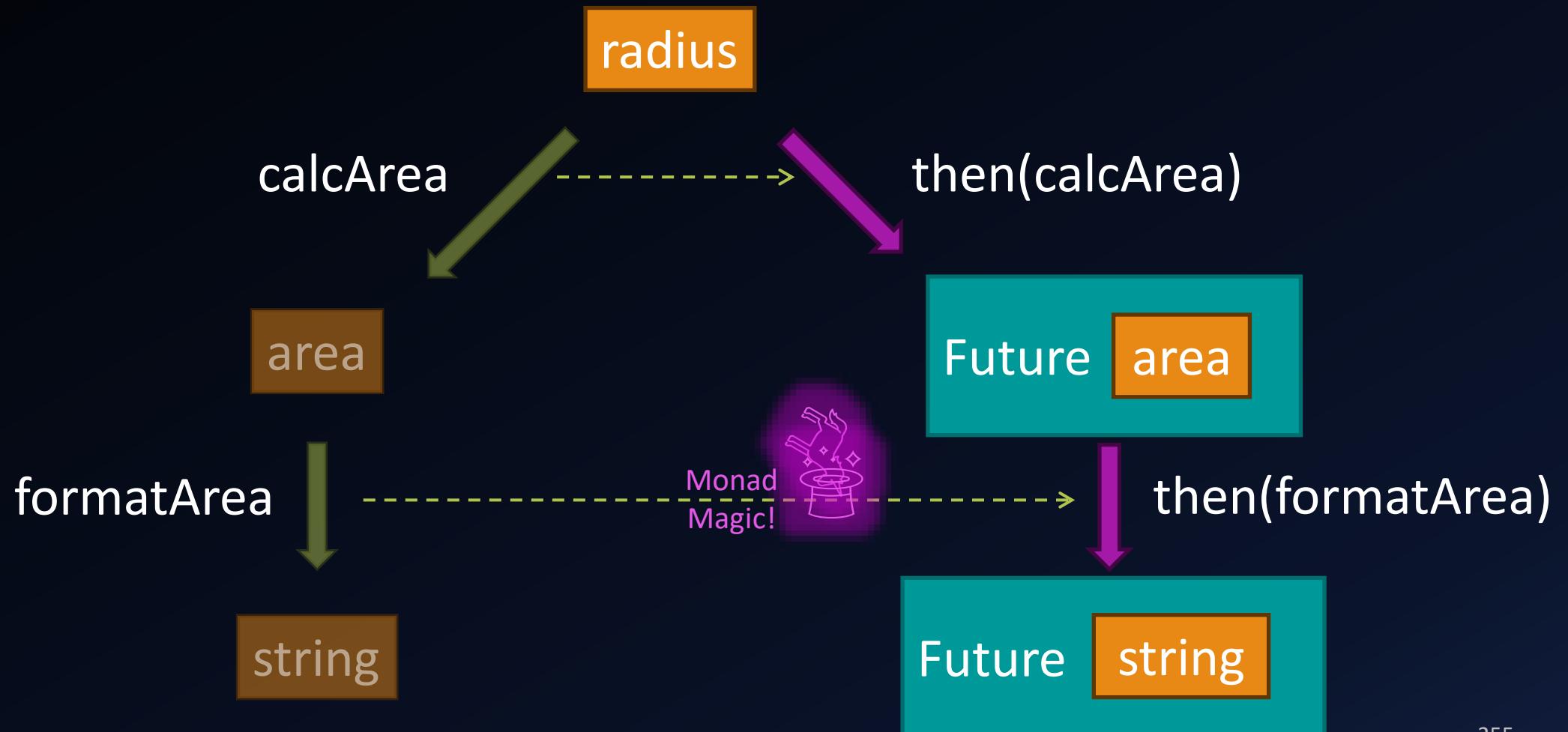
Combining Monads

CONTINUATION & WRITER

A Pipeline of Slow, Blocking Tasks



Continuation Monad



Using QtConcurrent With Continuation

```
double calcArea(double radius);
string formatArea(double area);

auto future = QtConcurrent::run(calcArea, initialValue)
    .then(formatArea);

//...
auto result = future.result(); // blocks until result is ready
print("Result: {}", result);
```

Using QtConcurrent With Continuation

```
double calcArea(double radius);
string formatArea(double area);

auto future = QtConcurrent::run(calcArea, initialValue)
    .then(formatArea);

//...
auto result = future.result(); // blocks until result is ready
print("Result: {}", result);
```

Using QtConcurrent With Continuation

```
double calcArea(double radius);
string formatArea(double area);

auto future = QtConcurrent::run(calcArea, initialValue)
    .then(formatArea);

//...
auto result = future.result(); // blocks until result is ready
print("Result: {}", result);
```

Using QtConcurrent With Continuation

```
double calcArea(double radius);
string formatArea(double area);

auto future = QtConcurrent::run(calcArea, initialValue)
    .then(formatArea);

//...
auto result = future.result(); // blocks until result is ready
print("Result: {}", result);
```

Using QtConcurrent With Continuation

```
double calcArea(double radius);
string formatArea(double area);

auto future = QtConcurrent::run(calcArea, initialValue)
    .then(formatArea);

//...
auto result = future.result(); // blocks until result is ready
print("Result: {}", result);
```

Using QtConcurrent With Continuation

```
double calcArea(double radius);
string formatArea(double area);

auto future = QtConcurrent::run(calcArea, initialValue)
    .then(formatArea);

//...
auto result = future.result(); // blocks until result is ready
print("Result: {}", result);
```



How to trace such a pipeline
Lock-free and preserving order

Using a Writer Monad

```
double calcArea(double radius);
string formatArea(double area);

auto result = CWriter(inputValue)
    .and_then(calcArea, CTraceEntry{"Some trace"})
    .and_then(formatArea, CTraceEntry{"Other trace"});

print("Result: {} ", result.m_Value);
for(const auto& item : result.m_Trace)
{
    print("{}:", CTraceEntry::toString);
}
```

Using a Writer Monad

```
double calcArea(double radius);
string formatArea(double area);

auto result = CWriter(inputValue)
    .and_then(calcArea, CTraceEntry{"Some trace"})
    .and_then(formatArea, CTraceEntry{"Other trace"});

print("Result: {} ", result.m_Value);
for(const auto& item : result.m_Trace)
{
    print("{}:", CTraceEntry::toString);
}
```

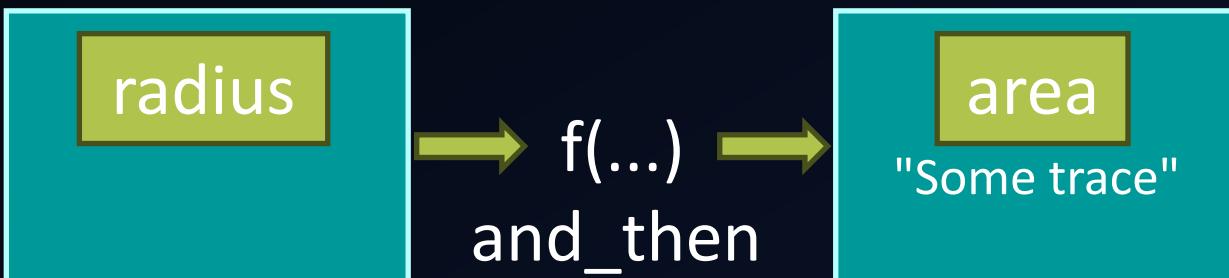
radius

Using a Writer Monad

```
double calcArea(double radius);
string formatArea(double area);

auto result = CWriter(inputValue)
    .and_then(calcArea, CTraceEntry{"Some trace"})
    .and_then(formatArea, CTraceEntry{"Other trace"});

print("Result: {} ", result.m_Value);
for(const auto& item : result.m_Trace)
{
    print("{}:", CTraceEntry::toString);
}
```

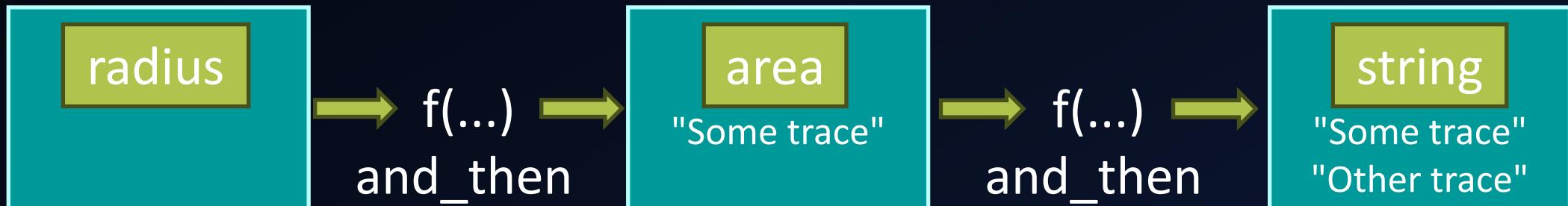


Using a Writer Monad

```
double calcArea(double radius);
string formatArea(double area);

auto result = CWriter(inputValue)
    .and_then(calcArea, CTraceEntry{"Some trace"})
    .and_then(formatArea, CTraceEntry{"Other trace"});
```

```
print("Result: {} ", result.m_Value);
for(const auto& item : result.m_Trace)
{
    print("{}:", CTraceEntry::toString());
}
```

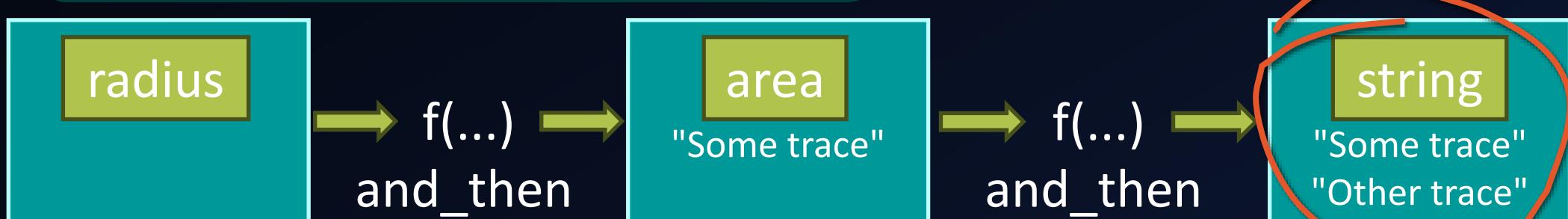


Using a Writer Monad

```
double calcArea(double radius);
string formatArea(double area);

auto result = CWriter(inputValue)
    .and_then(calcArea, CTraceEntry{"Some trace"})
    .and_then(formatArea, CTraceEntry{"Other trace"});
```

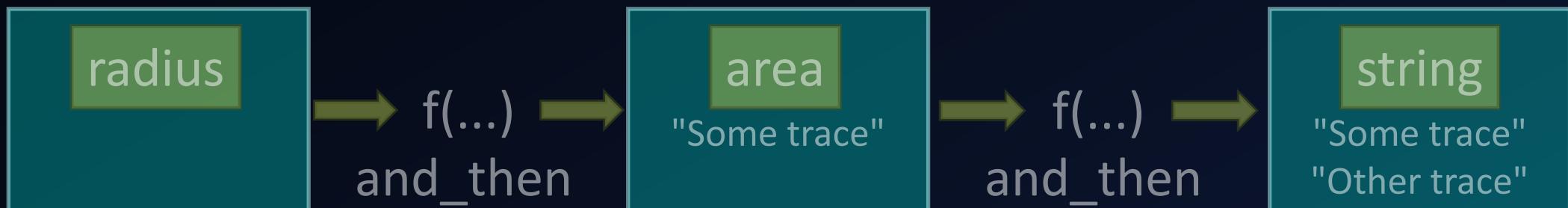
```
print("Result: {} ", result.m_Value);
for(const auto& item : result.m_Trace)
{
    print("{}:", CTraceEntry::toString);
}
```



Functions Returning Trace: Total Flexibility

```
CWriterValueAndTrace<double> calcArea(double radius);  
CWriterValueAndTrace<string> formatArea(double area);
```

```
auto result = CWriter(inputValue)  
    .and_then(calcArea)  
    .and_then(formatArea);  
  
print("Result: {} ", result.m_Value);  
for(const auto& item : result.m_Trace)  
{  
    print("{}:", CTraceEntry::toString);  
}
```

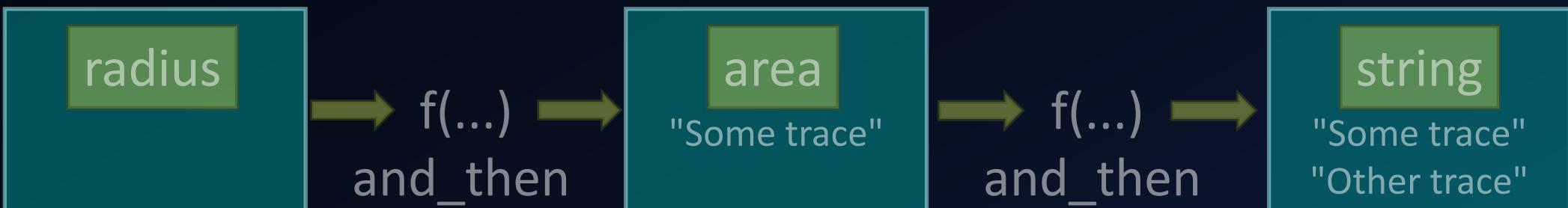


Functions Returning Trace: Total Flexibility

```
CWriterValueAndTrace<double> calcArea(double radius);  
CWriterValueAndTrace<string> formatArea(double area);
```

```
auto result = CWriter(inputValue)  
    .and_then(calcArea)  
    .and_then(formatArea);
```

```
print("Result: {} ", result.m_Value);  
for(const auto& item : result.m_Trace)  
{  
    print("{}:", CTraceEntry::toString);  
}
```



Combining Continuation and Writer

```
CWriterValueAndTrace<double> calcArea(double radius);
CWriterValueAndTrace<string> formatArea(double area);

auto future = QtConcurrent::run(makeWriter, inputValue)
    .then(addWriterApi(calcArea))
    .then(addWriterApi(formatArea));

CWriter<string> result = future.result(); // blocks
print("Result: {}", result.m_Value);

for(const auto& item : result.m_Trace)
{
    print("{}:{}", CTraceEntry::toString());
}
```

Combining Continuation and Writer

```
CWriterValueAndTrace<double> calcArea(double radius);
CWriterValueAndTrace<string> formatArea(double area);

auto future = QtConcurrent::run(makeWriter, inputValue)
    .then(addWriterApi(calcArea))
    .then(addWriterApi(formatArea));

CWriter<string> result = future.result(); // blocks
print("Result: {}", result.m_Value);

for(const auto& item : result.m_Trace)
{
    print("{}:", CTraceEntry::toString());
}
```

Combining Continuation and Writer

```
CWriterValueAndTrace<double> calcArea(double radius);
CWriterValueAndTrace<string> formatArea(double area);

auto future = QtConcurrent::run(makeWriter, inputValue)
    .then(addWriterApi(calcArea))
    .then(addWriterApi(formatArea));

CWriter<string> result = future.result(); // blocks
print("Result: {}", result.m_Value);

for(const auto& item : result.m_Trace)
{
    print("{}:", CTraceEntry::toString());
}
```

Combining Continuation and Writer

```
CWriterValueAndTrace<double> calcArea(double radius);
CWriterValueAndTrace<string> formatArea(double area);

auto future = QtConcurrent::run(makeWriter, inputValue)
    .then(addWriterApi(calcArea))
    .then(addWriterApi(formatArea));

CWriter<string> result = future.result(); // blocks
print("Result: {}", result.m_Value);

for(const auto& item : result.m_Trace)
{
    print("{}:", CTraceEntry::toString());
}
```

Combining Continuation and Writer

```
CWriterValueAndTrace<double> calcArea(double radius);
CWriterValueAndTrace<string> formatArea(double area);

auto future = QtConcurrent::run(makeWriter, inputValue)
    .then(addWriterApi(calcArea))
    .then(addWriterApi(formatArea));

CWriter<string> result = future.result(); // blocks
print("Result: {}", result.m_Value);

for(const auto& item : result.m_Trace)
{
    print("{}:{}", CTraceEntry::toString);
}
```

Combining Continuation and Writer

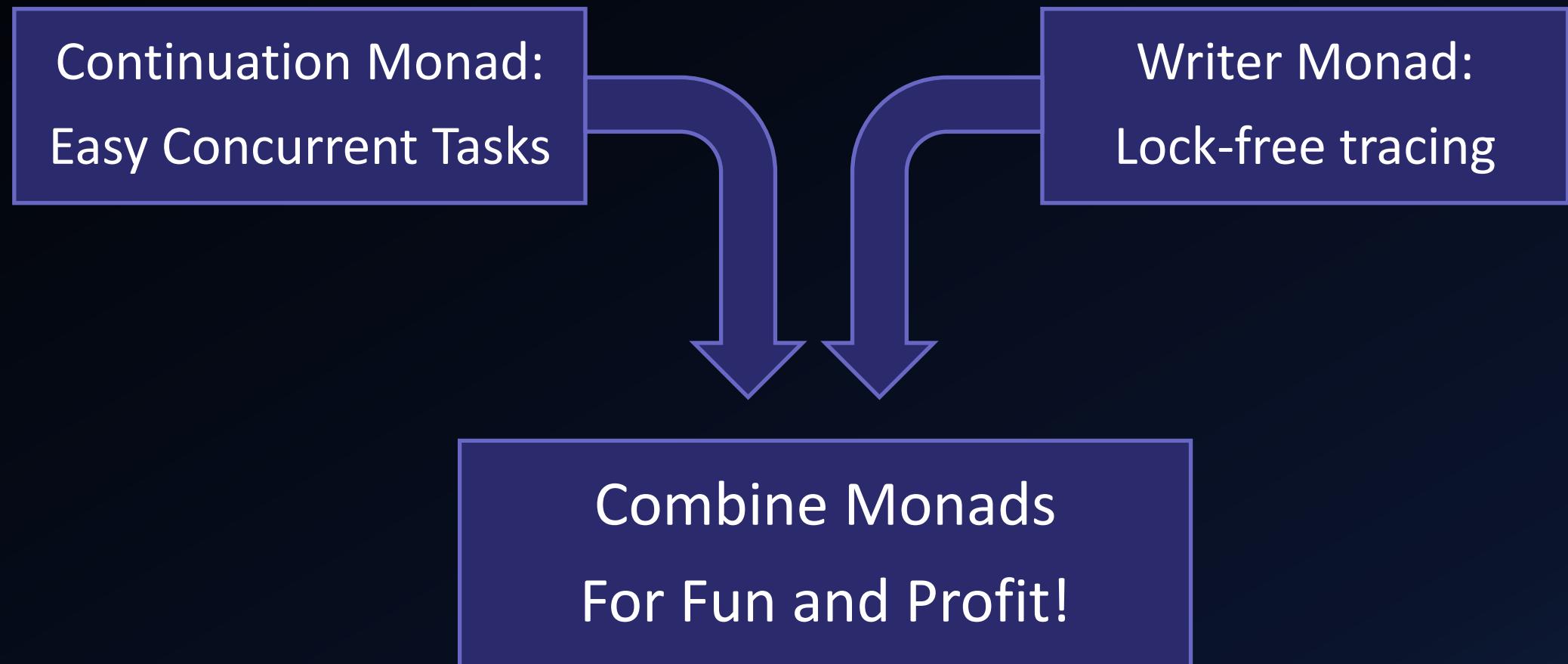
```
CWriterValueAndTrace<double> calcArea(double radius);
CWriterValueAndTrace<string> formatArea(double area);

auto future = QtConcurrent::run(makeWriter, inputValue)
    .then(addWriterApi(calcArea))
    .then(addWriterApi(formatArea));

CWriter<string> result = future.result(); // blocks
print("Result: {}", result.m_Value);

for(const auto& item : result.m_Trace)
{
    print("{}:{}", CTraceEntry::toString);
}
```

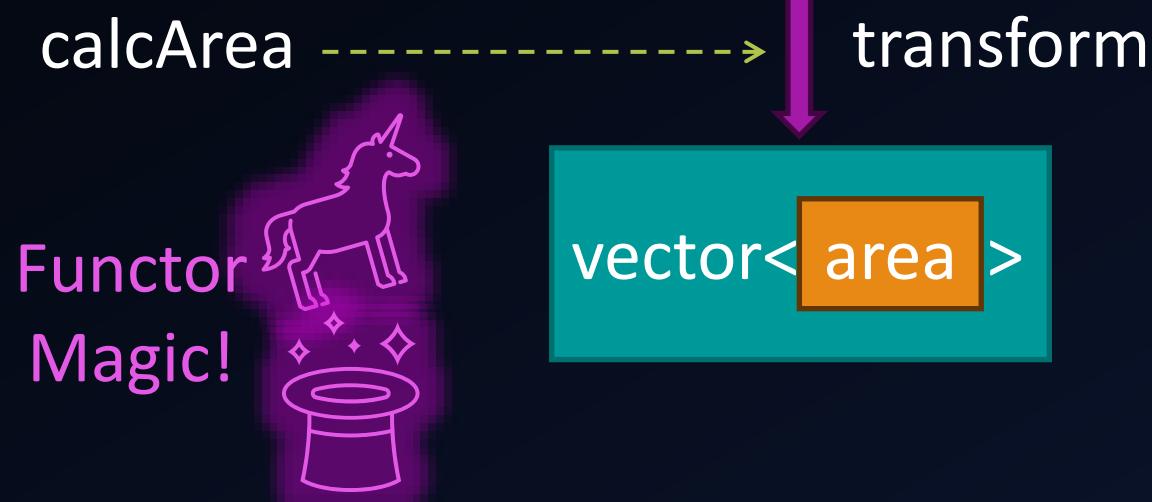
Combining Monads: Summary



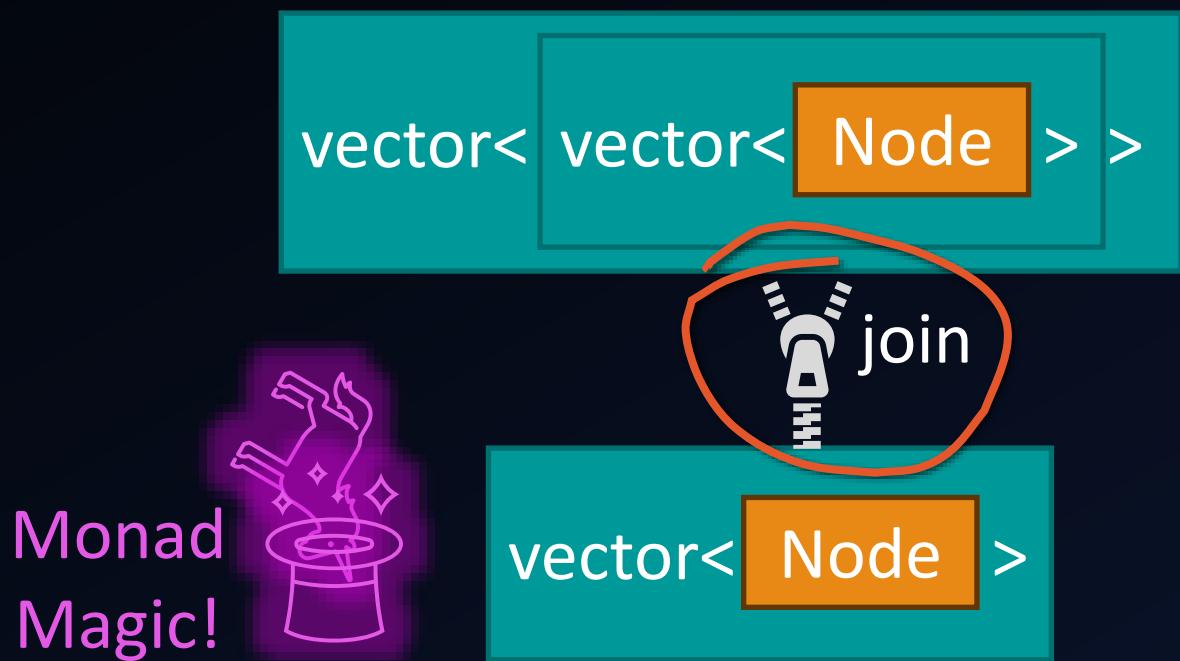
Wrap Up

FUNCTORs, MONADS AND C++23

Functor Magic!



Monad Magic!



Advantages of Pure Functions

Easy to
reason about

Easy to unit test

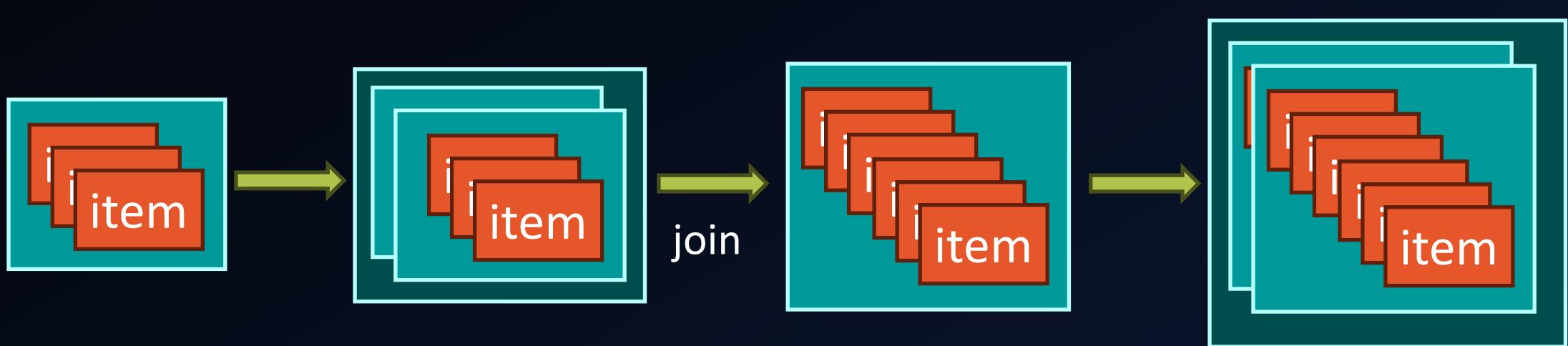
Thread-safe

Functors, Monads, Pure Functions



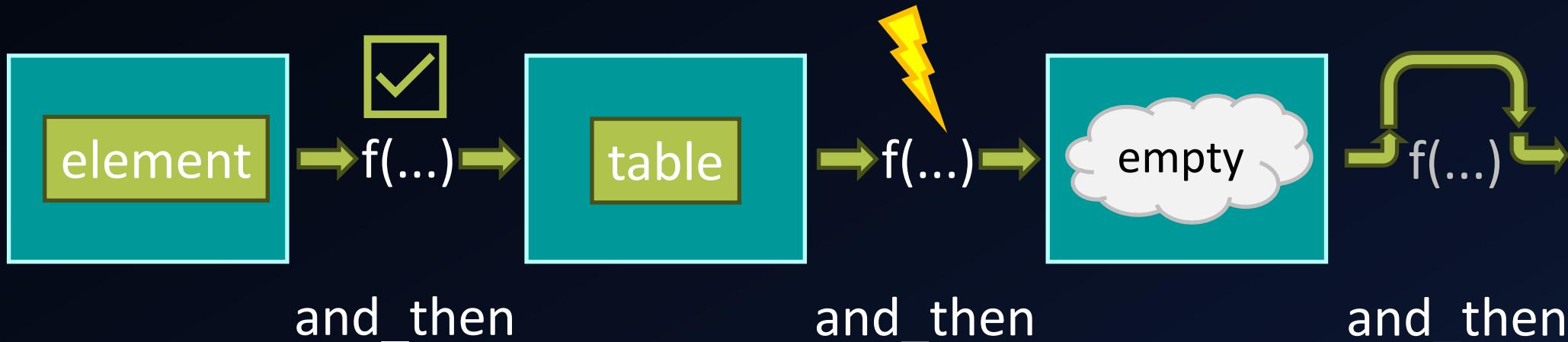
Ranges / Views Monad

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



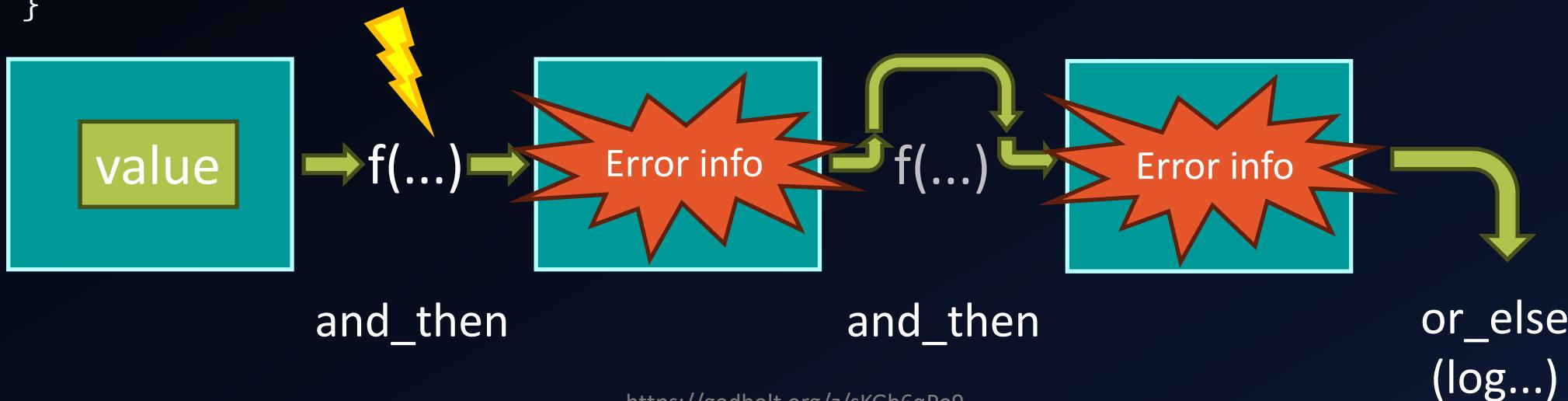
Optional (Maybe) Monad

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative);
}
```



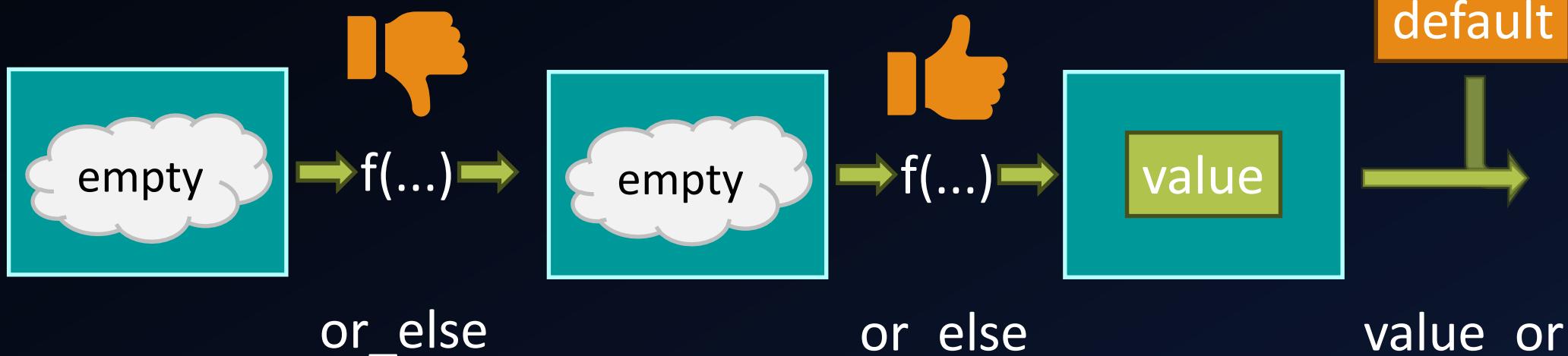
Expected Monad

```
optional<bool> isIntCellValueNegative(CDb db, Key key, CLocation location)
{
    return getElement(db, key)
        .and_then(getTable)
        .and_then([location](CTable table)
            { return getCell(table, location); })
        .and_then(getNumericCellValue)
        .transform(isNegative)
        .or_else(log<bool>);
}
```



Default “Monad”

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



Safe and Readable Code: 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

Thanks go to:

MUC++

BinaryCore

Ivan Čukić

C++ User Group Vienna

#include community

My family