A Gentle Introduction to Ranges v3

Say Goodbye to STL the Way You (Probably Don't) Know It

Hints

- You can download this talk **and** code samples in the form of unit tests
  - https://github.com/daixtrose/gentle-intro-to-ranges
- Published under MIT licence
  - Code for talk framework based on remark.js was stolen from Kirk Shoop's Intro to RxCpp
  - Please contribute fixes or file an issue if you find errors or know about improvements.

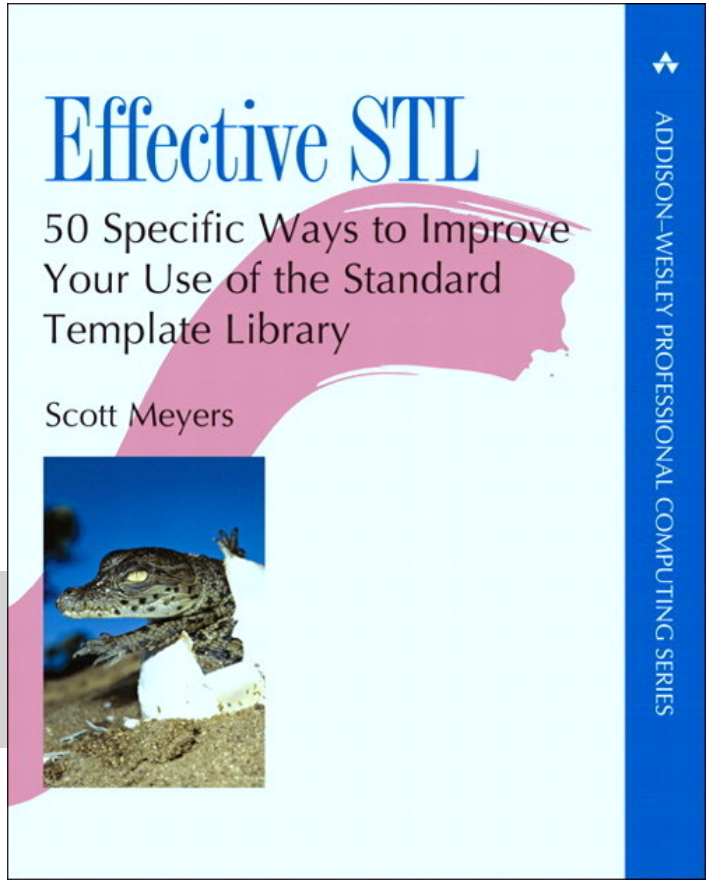
Disclaimer

- This is not "The Complete Guide", but
  - just a gentle **introduction**
  - a **user's view** (sic!) on ranges
  - focus is more on getting the basic ideas (tutorial)

Our Starting Point: STL

- STL is the recommended **C++** way to deal with sets and operations on sets
  - Provides Containers
  - Provides Algorithms
  - Provides Utilities
- STL is generic
- STL makes no compromise: performance first
- STl requires educated users

> Hint: I am a great fan of the STL and encourage(d) its usage wherever possible.



STL: The Crisis

Andrei Alexandrescu: "Iterators Must Go" (2009), Video here

TL;DR:

- STL was built using a rigorous fundamental, academic, generic approach, but
  - leads to counter-intuitive rules in some places
  - is hard to learn (and hard to remember)
- Working with data streams is a mess.
- Algorithms are NOT **composable**.
- Iterators
  - are so complex, one needs 10 years of education and/or
  - library help like Boost.Iterator in order to get things right.

Ranges

- Many attempts/proposals, e.g.
  - Boost.Range Version 1 and 2
  - Eric Niebler: 3 (!) versions
- Rest of the talk deliberately ignores all other attempts,
  - shows application of Eric Niebler's ranges-v3 for C++11/14/17

Eric Niebler's Range-v3

- GitHub Repository at https://github.com/ericniebler/range-v3
- ISO Standard Proposal
  - https://ericniebler.github.io/std/wg21/D4128.html
  - TS: https://www.iso.org/obp/ui/#iso:std:iso-iec:ts:21425:ed-1:v1:en
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4651.pdf
- Explanations about iterators and why we need a sentinel
  - http://ericniebler.com/2015/02/03/iterators-plus-plus-part-1/

October 2014

December 2017

What are Ranges?

Eric Niebler @ericniebler · 7. Mai

Antwort an @SeanParent @tvaneerd und 10 weitere

Containers are Ranges. A span is a Range and a View. But yeah, span is weird, and I'm generally not a fan. At least it's better than it was....

The Committee is a tough place for strict adherents of logical consistency.

Tweet übersetzen

3 2

Sean Parent @SeanParent · 7. Mai

By "span is a Range and a View" you of course mean it is neither, since something cannot be both. I think this is an instance were we need user defined reference types. i.e. `class span : public &`. It seems to be range&. (My use of range was the "classic" term, now view).

Tweet übersetzen

2 2

What are Ranges?

Eric Niebler @ericniebler · 7. Mai

"Range" is just something that you can pass to begin and end. So yes, I mean span is a Range and a View, because all Views are Ranges. But Views are not Containers.

(The terminology changed. "Range" used to be spelled "Iterable".)

Tweet übersetzen

1 3

Eric Niebler @ericniebler · 7. Mai

The algorithms are constrained with "Range". They only care that they can call begin and end and pass the results to an algorithm that accepts iterators.

Tweet übersetzen

1 1

What are Ranges?

What are Ranges?

¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯

¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯

¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯

¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯

¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯ ¯_(ツ)_/¯

What are Ranges?

> C++ Ranges are **C#**'s **LINQ** for C++ - Markus Werle

C# is a language I struggled with may times because of its unnecessary limitations due to bad language design (esp. generics are multiple inheritance) and its limited expressiveness, but

C# got some things right

- Lambdas
- Concurrency esp. `async/await` (see Steven Toub's Concurrency in C# Cookbook) and `CancellationToken`
- LINQ (`*this` talk)
- Reactive Extensions (`*this->next()` talk)

C++ gotta catch up

- C++17's **generic** lambdas ✔
- `std::async` flawed, but concurrency will evolve (I see a bright `<future>` here, with `coroutines` and more confusion)
- Ranges converge soon into ISO C++ ✔
- RxCpp will converge during the next years

What are Ranges?

> C++ Ranges are **Pure Monadic Goodness** - Bartosz Milewski

- **Functional Programming** curiously recurring all the time
- Milewski: A **monad** is an applicative functor with an additional ability, which can be expressed either as a way of flattening a doubly encapsulated object, or as a way of applying a functor factory to an encapsulated object.
- Simply said, it's all about
  - functions/algorithms (operations on values or sets of values)
  - composition of functions
  - programming without side effects (helps a lot when dealing with concurrency)

What are Ranges?

> Ranges are a contribution to making C++ easier to write and reason about

STL vs. Ranges: Transform (LINQ: Select, J*: Map)

STL Approach

```
typedef /* ... ? ... */ result_t;

std::vector<result_t> bar;

std::transform (
    std::begin(foo),
    std::end(foo),
    std::back_inserter(bar),
    [](auto const v) {
        return 2 * v;
    });
```

Ranges Approach

```
// bar is lazy and composable
auto bar =

    foo | view::transform(


        [](auto const v) {
            return 2 * v;
        });
```

STL vs. Ranges: Composition

```
auto timesTwo = [](auto const v) { return 2 * v; };
auto plusTen = [](auto const v) { return v + 10; };
```

STL Approach

```
auto composition = [](auto v) {
    return plusTen(timesTwo(v));
});

std::transform (
    std::begin(foo),
    std::end(foo),
    std::back_inserter(bar),
    composition);
```

Ranges Approach

```
// Remember: lazy,
// i.e. optimizer looks through
auto bar =
    foo
    | view::transform(timesTwo)
    | view::transform(plusTen)
    ;
```

STL vs. Ranges: Composition

```
auto timesTwo = [](auto const v) { return 2 * v; });
auto plusTen = [](auto const v) { return v + 10.0; });
```

STL Approach

```
auto composition = [](auto v) {
    return plusTen(timesTwo(v));
});

std::transform (
    std::begin(foo),
    std::end(foo),
    std::back_inserter(bar),
    composition);
```

Ranges Approach (Composition to the Max)

```
// Remember: lazy,
// i.e. optimizer looks through
auto bar =
    foo | view::transform(timesTwo);

auto baz =
    bar | view::transform(plusTen);
```

Switching back to STL

Explicit

```
auto bar =
    foo | view::transform(timesTwo)
    | to_vector
    ;

auto bar =
    /* ... * /
    | to_<std::list>();

auto bar =
    /* ... * /
    | to_<std::vector<long>>();
```

Implicit

```
std::vector<result_t> bar =
    foo | view::transform(timesTwo)
    ;

std::list<result_t> bar =
    /* ... * /
    ;

std::vector<long> bar =
    /* ... * /
    ;
```

Switching back to STL with Optimal Runtime Performance

Citation from Casey Carter's Answer on Stack Overflow:

```
auto b = a | ranges::view::transform(complexFun) | ranges::to_vector;
```

If you already have a destination vector whose capacity you want to reuse:

```
b.clear(); // Assuming b already contains junk
b |= ranges::action::push_back(a | ranges::view::transform(complexFun));
```

> In both cases, **range-v3** is smart enough to reserve capacity in the destination vector for **ranges::size(a | ranges::view::transform(complexFun))** elements to avoid copies due to reallocation.

Dual Interface: Chainable Pipe Operator vs. Regular Function Call

For most algorithms there exist two **ranges-v3** interfaces:

```
auto bar = foo
    | view::transform(fn);
```

```
auto bar =
    transform(foo, fn);
```

Some algorithms are not available in the pipe (|) form:

```
// std::copy(std::begin(v), std::end(v),
//     std::ostream_iterator<int>(std::cout, ' '));
ranges::copy(v, ranges::ostream_iterator<int>(std::cout, ' '));
// or (assuming using namespace ranges;)
std::cout << view::transform(vs, view::all) << std::endl;
// or TODO: testen!
std::cout << vs | view::transform(view::all) << std::endl;
```

Group_By (LINQ: GroupBy) and Join (LINQ: SelectMany)

```
struct Person {
    std::string firstname;
    std::string surname;
    int year;
};

std::ostream & operator<<(std::ostream & os, Person const & person) {
    os << person.surname << ", " << person.firstname
       << " was born in " << person.year;
    return os;
}

std::vector<Person> people{
    { "Jared", "Kushner", 1981 }, { "Melania", "Trump", 1970 },
    { "Donald", "Trump", 1946 }, { "Ivana", "Trump", 1949 },
};
```

Group_By (LINQ: GroupBy)

```
auto surname_is_equal = [](auto const & p1, auto const & p2)
    { return p1.surname == p2.surname; };

auto groups = people | ranges::view::group_by(surname_is_equal);

for (auto const & group : groups) {
    std::cout << "-------\n";
    copy(group, ostream_iterator<Person>(std::cout, "\n"));
}
```

```
-------
Kushner, Jared was born in 1981
-------
Trump, Melania was born in 1970
Trump, Donald was born in 1946
Trump, Ivana was born in 1949
```

Join (LINQ: SelectMany)

```
auto is_younger = [](auto const & p1, auto const & p2)
    { return p2.year < p1.year; };

auto each_group_sorted_by_age = groups
    | transform([=](auto g) {
        sort(g, is_younger); // range-v3/issues/266: no view::sorted
        return /* expensive copy of */ g; });

copy(join(each_group_sorted_by_age),
    ostream_iterator<Person>(std::cout, "\n"));

--> sorted by age, then joined:
Kushner, Jared was born in 1981
Trump, Melania was born in 1970
Trump, Ivana was born in 1949
Trump, Donald was born in 1946
```

Filtering - By Hand Using Yield_If (C#: Yield)

```cpp
std::vector<int> numbers{1, 2, 3, 4, 5, 6};

auto is_odd = [](int i) { return i % 2 != 0; };

using ranges::view::for_each;
using ranges::yield_if;

// odd_numbers is a *lazy* expression, i.e. is_odd
// will NOT be called in next statement
auto odd_numbers = for_each(numbers, [=](int i) {
    return yield_if(is_odd(i), i);
    });

// now that the expression gets evaluated, is_odd will be called
REQUIRE(ranges::equal(odd_numbers, std::vector<int>{1, 3, 5}));
```

Filtering with Filter and Remove_If (LINQ: Where)

```
std::vector<int> numbers{1, 2, 3, 4, 5, 6};

using ranges::view::filter; // ambiguous, but meant positive.
using ranges::view::remove_if;
using ranges::not_fn;

auto is_odd = [](int i) { return i % 2 != 0; };

auto odd_numbers = numbers | filter(is_odd);
auto odd_numbers_alt = numbers | remove_if(not_fn(is_odd));
auto even_numbers = numbers | remove_if(is_odd);
```

Set Operations

Unique (LINQ: Distinct)

Removes duplicate values from a collection. In most libraries, finding duplicates requires a sorted container/enumerable.

```
std::vector<int> numbers{3, 1, 2, 3, 3, 4, 3, 5, 6};

using ranges::action::unique;
using ranges::action::sort;

numbers |= sort | unique;
```

Set Operations

Set Difference (LINQ: Except)

Returns the set difference, which means the elements of one collection that do not appear in a second collection.

```
std::vector<int> v1{1, 2, 3, 4};
std::vector<int> v2{3, 4, 5};

using ranges::view::set_difference;

auto v = set_difference(v1, v2);

using ranges::equal;
REQUIRE(equal(v, std::vector<int>{1, 2}));
```

Set Operations

Symmetric Set Difference

The symmetric difference finds the elements that are found in either of the ranges, but not in both of them

```
std::vector<int> v1{1, 2, 3, 4};
std::vector<int> v2{3, 4, 5};

using ranges::view::set_symmetric_difference;

auto v = set_symmetric_difference(v1, v2);

using ranges::equal;
REQUIRE(equal(v, std::vector<int>{1, 2, 5}));
```

Set Operations

Set Intersection (LINQ: Intersect)

Returns the set intersection, which means elements that appear in each of two collections.

```
std::vector<int> v1{1, 2, 3, 4};
std::vector<int> v2{3, 4, 5};

using ranges::view::set_intersection;

auto v = set_intersection(v1, v2);

using ranges::equal;
REQUIRE(equal(v, std::vector<int>{3, 4}));
```

Set Operations

Set Union (LINQ: Union)

Returns the set union, which means unique elements that appear in either of two collections.

```
std::vector<int> v1{1, 2, 3, 4};
std::vector<int> v2{3, 4, 5};

using ranges::view::set_union;

auto v = set_union(v1, v2);

using ranges::equal;
REQUIRE(equal(v, std::vector<int>{1, 2, 3, 4, 5}));
```

Partitioning

- skip
- skip_while
- take
- take_while

Generating Sequences

- empty
- range of numbers
- repeat

Accessing elements

- at
- first
- first or default
- last
- last or default
- single

```
std::vector<PolarCoordinate>
selectNearestSegmentToAngleZero(
    std::vector<std::vector<PolarCoordinate>> const& segments)
{
    // ... using declaratives omitted ...
    auto average_angle = [](auto&& coordinates) {
        auto angles = coordinates
            | transform([](auto pc) { return pc.angle; });
        auto sumOfAngles = accumulate(angles, 0.0);
        auto result = sumOfAngles
            / static_cast<double>(size(coordinates));
        return result;
    };
    auto averageValuesAbs = segments
        | transform(average_angle) // yields a range of average values
        | transform([](auto v) { return std::abs(v); });
    auto index = distance(averageValuesAbs.begin(),
        min_element(averageValuesAbs));

    return segments[index];
```

How I lost my reputation

```
std::vector<std::vector<PolarCoordinate>> segments;
auto segmentsReversedViaNestedTransform = segments
  | transform([](auto s) {
      return s | transform(polarToCartesian) | reverse;
    });

std::vector<PolarCoordinate> firstSegmentReversed =
    segments[0] | transform(polarToCartesian) | reverse;

for (auto const& point : firstSegmentReversed) {
    std::cerr << point.first << ", " << point.second << std::endl;
}

for (auto const& point : *begin(segmentsReversedViaNestedTransform)) {
    std::cerr << point.first << ", " << point.second << std::endl;
}
```

How I lost my reputation

```
std::vector<std::vector<PolarCoordinate>> segments;
std::vector<PolarCoordinate> firstSegmentReversed =
        segments[0] | transform(polarToCartesian) | reverse;

for (auto const& point : firstSegmentReversed) {
    std::cerr << point.first << ", " << point.second << std::endl;
}
```

yields

```
2.0944, 29.96
2.08567, 29.96
2.07694, 29.96
2.06821, 29.96
2.05949, 29.96
```

How I lost my reputation

```
auto segmentsReversedViaNestedTransform = segments
    | transform([](auto s) {
        return s | transform(polarToCartesian) | reverse; });

for (auto const& point : *begin(segmentsReversedViaNestedTransform)) {
    std::cerr << point.first << ", " << point.second << std::endl;
}
```

yields

```
2.0944, 29.96
2.08567, 29.96
2.07694, 29.96
2.06821, 29.96
0, 29.96   <------ HERE IS SOMETHING WRONG
```

How I lost my reputation

```
std::vector<std::vector<PolarCoordinate>> segments;
auto segmentsInCartesianCoordinates = segments
  | transform([](auto s) {
      return s | transform(polarToCartesian) | reverse;
    });
```

Explanation by @gnzlbg:

> `[](auto s) { return s | reverse; }` creates a dangling view. The inner vector will be copied into the `s` argument (the lambda stack frame), and `| reverse` creates a view (that is, a pair of pointers) into this local vector. The problem is that this vector will be destroyed on scope exit (freeing the memory), so when the view is returned its pair of pointers into the vector will point into freed memory. When you try to iterate the view, you are basically dereferencing these pointers, and that is undefined behavior: read after free.

How I lost my reputation

```
std::vector<std::vector<PolarCoordinate>> segments;
auto segmentsInCartesianCoordinates = segments
  | transform([](auto&& s) {
      return s | transform(polarToCartesian) | reverse;
    });
```

Dangling References due to Views

```
auto doub = [](int i) -> int { return 2 * i; };

int main() {
  auto e = find_if( view::ints(1) | view::transform(doub), is_six );
  cout << "find-six: " << *e.get_unsafe() << endl;
}
```

Prefer this version:

```
auto doub = [](int i) -> int { return 2 * i; };

int main() {
  auto candidates = view::ints(1) | view::transform(doub);
  auto e = find_if( candidates, is_six );
  cout << "find-six: " << *e << endl;
}
```

Maps

Extract the values from a dictionary

```
std::map<int, std::wstring> ss =
    { {1,L"1"},
      {2,L"2"},
      {3,L"3"} };
auto rng = ss | ranges::view::reverse | ranges::view::values;
```

Some More Tips and Tricks

Compare ranges (see this code live on Wandbox):

```
int main() {
    ranges::view::iota;
    ranges::view::reverse;

    std::vector<int> vec{5,4,3,2,1,0};
    assert(ranges::equal(
        vec,
        ints(0, 6) | reverse));
}
```

Thank You for Your Attention

(and welcome to modern C++)

Gimme that Old Time Religion (2)

Scott Meyer's perfect version of a map search and update

```
typename MapType::iterator lb =
    m.lower_bound(k);

if (lb != m.end() &&
    !(m.key_comp()(k, lb->first))) {
    lb->second = v;
    return lb;
}
else {
    typedef typename MapType::value_type MVT;
    return m.insert(lb, MVT(k, v));
}
```