Kindergarten C++20 Ranges

Range-v3 library by Eric Niebler

Voted to standardize in C++20

https://github.com/ericniebler/range-v3

Steve MacCabe

2019/June

Warmup –
what made me want to give this talk?

- **Let’s make a vector of integers**
- **Then let’s**
  - Pull out the even values
  - Double them
  - Add 3
  - Print their squares

```
std::vector< int > vec( 6 );
std::iota( vec.begin(), vec.end(), 1 );
std::vector< int > vec_even;
vec_even.reserve( vec.size() / 2 + 1 );
std::copy_if( vec.begin(), vec.end(), std::back_inserter(
vec_even ),
        []( int n ) { return n % 2 == 0; });
std::transform( vec_even.begin(), vec_even.end(),
vec_even.begin(),
        []( int n ) { return 2 * n;      });
std::transform( vec_even.begin(), vec_even.end(),
vec_even.begin(),
        []( int n ) { return n + 3;      });
std::vector< int >::const_iterator it = vec.begin();
for( ; it != vec.end(); ++it )
    std::cout << ( *it ) * ( *it ) << "  ";
// 49  121  225
```

```
vector< int > vec = ranges::view::ints( 1, 7 );
auto vec_even = vec | filter( EvenFilter )
                    | transform( MultX2  )
                    | transform( Add_3   );

for( const auto& value : vec_even )
     cout << value * value << "  ";

// 49  121  225
```

```cpp
auto EvenFilter  = []( int n ) { return n % 2 == 0; };
auto MultX2      = []( int n ) { return n * 2;      };
auto Add_3       = []( int n ) { return n + 3;      };

vector< int > vec = ranges::view::ints( 1, 7 );
auto vec_even = vec | filter( EvenFilter )
                    | transform( MultX2  )
                    | transform( Add_3   );

for( const auto& value : vec_even )
     cout << value * value << "  ";

// 49  121  225
```

```
auto EvenFilter  = []( int n ) { return n % 2 == 0; };
auto MultX2      = []( int n ) { return n * 2;       };
auto Add_3       = []( int n ) { return n + 3;       };
vector< int > vec { 1, 2, 3, 4, 5, 6 };
vector< int > vec_even( vec.size());
std::copy_if( vec.begin(), vec.end(), vec_even.begin()),
        EvenFilter );
std::transform( vec_even.begin(), vec_even.end(), vec_even.begin(),
        MultX2 );
std::transform( vec_even.begin(), vec_even.end(), vec_even.begin(),
        Add_3 );
for( const auto& value : vec_even )
     cout << value * value << "  ";
// 49  121  225
```

Kindergarten C++20 Ranges

- Ranges and simplicity
- Views and range adaptors
- Actions

Kindergarten C++20 Ranges

- Ranges and simplicity
- Views and range adaptors
- Actions

Ranges – Fluent C++ Blog

- https://www.fluentcpp.com/2017/01/12/ranges-stl-to-the-next-level/
- https://www.fluentcpp.com/2018/02/09/introduction-ranges-library/

Ranges – Fluent C++ Blog

- Essentially a ***range*** is something that can be traversed

- More precisely, a range is something that has a begin() and an end() method,
  - that return objects (iterators) that
    - Let you iterate over the range
    - And can be dereferenced to access these elements

- All STL containers are ranges

What’s so very cool about ranges?

- Haven’t you sensed that C++ is different from many high-level languages in things like sorting, especially for the simplest cases?
  - Java – `Arrays.sort( myArray )`
  - Python – `numpy.sort( myArray)`
  - *C++ – `std::sort( myArray.begin(), myArray.end(), myComparator )`*
- And more (ref: warmup slides)

Ranges – Fluent C++ Blog

- Ranges hide iterators, which are an implementation detail of containers
- Ranges allow composable functions
- Adaptors – these are ranges and composable
  - view::transform and view::filter

Eric Niebler’s Blog

- “Range v3 is a generic library that augments the existing standard library with facilities for working with ***ranges***. A range can be loosely thought of a pair of iterators, although they need not be implemented that way.”
- https://ericniebler.github.io/range-v3/index.html#tutorial-quick-start

Eric Niebler’s Blog

- Range v3 contains a full implementation of all the standard algorithms with range-based overloads for convenience.

Why use Ranges? - Niebler

Convenience

- It's more convenient to pass a single range object to an algorithm than separate begin/end iterators.

Why use Ranges? - Niebler

Convenience

- It's more convenient to pass a single range object to an algorithm than separate begin/end iterators.

```
std::vector<int> v{ 1, 6, 3, 8, 9, 3, 8 };
std::sort( v.begin(), v.end());
```

Why use Ranges? - Niebler

Convenience

- It's more convenient to pass a single range object to an algorithm than separate begin/end iterators.

```
std::vector<int> v{ 1, 6, 3, 8, 9, 3, 8 };
ranges::sort( v )
```

Why use Ranges? - Niebler

Composability

- Single range object [as opposed to iterator pairs] permits pipelines of operations.

Why use Ranges? - Niebler

Composability

- Single range object (as opposed to iterator pairs) permits pipelines of operations.

In a pipeline, a range is lazily adapted or eagerly mutated in some way, with the result immediately available for further adaptation or mutation.

Lazy adaption is handled by ***views***, and eager mutation is handled by ***actions***.

Kindergarten C++20 Ranges

- Ranges and simplicity
- Views and range adaptors
- Actions

Niebler

View

- A lightweight wrapper that presents a view of an underlying sequence of elements in some custom way without mutating or copying it.
  - Views are cheap to create and copy, and have non-owning reference semantics.

Niebler

View example

```
std::vector<int> vecin{1,2,3,4,5,6,7,8,9,10};

auto rngview = vecin
    | view::remove_if([](int i){return i % 2 == 1;})
    | view::transform([](int i)
           {return std::to_string(i);});

// Now rngview == {"2","4","6","8","10"}
// Now vecin   == {1,2,3,4,5,6,7,8,9,10} // Unchanged
```

Kindergarten C++20 Ranges

- Ranges and simplicity
- Views and range adaptors
- Actions (not in C++20)

Niebler

Action

- Construct that allows you to mutate a container in-place, or forward it through a chain of mutating operations.

Niebler

Action example

```
// Sort a vector and make it unique,
// note:
//  end trimmed, not like std::unique which needs erase()

vi = { 6,2,9,6,5,7,1,3,1,5 };

auto vi2 = std::move(vi) | ranges::action::sort
                         | ranges::action::unique;
// Now vi2 == {1,2,3,5,6,7,9};
// Now vi is gone (moved from)

// Same thing in-place using the 'pipe equals' operator
vi = { 6,2,9,6,5,7,1,3,1,5 };

vi |= ranges::action::sort | ranges::action::unique;
// Now vi == {1,2,3,5,6,7,9};
```

Summary

- Java
  - Arrays.sort(myArray)
- Python
  - numpy.sort(myArray)
- C++
  - std::sort(myArray.begin(), myArray.end(), myComparator)