https://www.wikiwand.com/en/Open_range#Media/File:Open_Range_Sign_Interstate_10_Frontage_Road_Arizona_2014.jpg

Introduction to ranges

C++ User Group Aachen

2024-06-05

- Ever got tired of this?

```
std::vector<std::string> vec;
std::sort(vec.begin(), vec.end());
```

- Or ended up with this bug?

```
std::sort(vec1.begin(), vec2.end());
```

- Or *actually* wanted to use algorithms in a clean, readable and composable way?

Where to get it

- https://github.com/ericniebler/range-v3
  - For C++14 and above
- Built into C++20/23
  - https://en.cppreference.com/w/cpp/ranges
- Some nice things from range-v3 are *not* in the standard or only in C++23 :-(

- **Algorithms**
  - Like STL, but with range overloads
- **Views**
  - Composable, lazy adaptations or ranges
- **Actions**
  - Eager application of an algorithm to a container

- Pipe operator - program like it’s ~~1972~~ bash!

Algorithms

```
std::vector<int> values{4, 3, 2, 1};
auto it1 = ranges::find(values, 2);
auto it2 = ranges::find(values, 17);

if (ranges::contains(values, 3)) { /* ... */ }


auto print = [](const auto& n) { std::cout << ' ' << n; };
ranges::for_each(values, print); // " 4 3 2 1"


ranges::sort(values);   // 1, 2, 3, 4
```

Algorithms & projections

- Projection: applied to every element

```
struct Endpoint
{
    std::string host;
    unsigned short port;
};

std::vector<Endpoint> addresses = /* ... */;
auto res = ranges::find(addresses, 443, &Endpoint::port);

// old-style:
auto res = std::find_if(addresses.begin(), addresses.end(),
          [] (const Endpoint& ep) { return ep.port == 443; });
```

Algorithms & projections

```
ranges::sort(addresses, {}, &Endpoint::getPort);

auto to_lower = [] (const auto& s) { return boost::to_lower_copy(s); };
std::vector<std::string> names;
auto entry = ranges::find(names, "me"s, to_lower);
```

Views

- Light-weight “wrapper” around a sequence of elements
  - Composable
  - Lazy evaluation
- Functional programming style
- “Smart iterators” in good

Views: composition

- Chain function calls or use the pipe operator
  - D(C) —> C | D

```
namespace rv = ranges::views;
auto notLocalhost = [](const Endpoint& ep) {
  return !ranges::contains(std::array{"localhost", "127.0.0.1"}, ep.host);
};

auto view = rv::transform(rv::filter(addresses, notLocalhost), toUri);

// same as:
auto view2 = addresses | rv::filter(notLocalhost) | rv::transform(toUri);
```

Views: lazy evaluation

- Elements are computed *during iteration*

```
auto toUri = [] (const Endpoint& ep) {
    return fmt::format("http://{}:{}", ep.host, ep.port); };

auto urisViews = addresses | ranges::views::transform(toUri);
for (const std::string& uri: urisViews) { /* ... */ }
```

```cpp
std::vector<int> v{2, 3, 4, 1, 4, 7, 3, 5};
auto is_even = [](int i) { return i % 2 == 0; };
auto square = [](int i) { return i * i; };

auto view = v | rv::filter(is_even)
              | rv::transform(square)
              | rv::take(2);
// no operation performed yet – only declared a view

// iterate over the view or create a container:
std::vector<int> v2 = view | ranges::to_vector;
auto v_set = view | ranges::to<std::set>;

// Because of take(2), square is only called twice.
```

Problem: dangling iterators

What about owning_view?!

- rvalue range:

```
std::vector<int> getVector();
auto it = ranges::find(getVector(), 3); // it??
```

- Iterator cannot be used: it has type `ranges::dangling`, which results in compilation errors when trying to use it:

```
auto res = *it;
// Error: no match for operator*(operand type is
'ranges::borrowed_iterator_t<std::vector<int>> aka 'ranges::dangling')
```

- Some use cases support rvalues:

```
int min = ranges::min(getVector());
bool hasThree = ranges::contains(getVector(), 3);
```

OK: borrowed ranges

- rvalue ranges are “ok” for “borrowed ranges”
  (i.e. `std::string_view`, `std::span` and some internal types)

```cpp
std::string_view getStringView();
auto it = ranges::find(getStringView(), 'r'); // ok

std::span<int> getSpan();
auto min = ranges::min_element(getSpan());
```

Range concepts

- `range`
  - Has begin and end iterator
- `sized_range`
  - Size is known in constant time
- Matching iterator types
  - Forward range, bidirectional range, random access range, …
- …

Range (view) factories

- `views::empty`
  - Creates an empty view
- `views::single`
  - Creates a view that contains a single element
- `views::iota`
  - Creates a view over a monotonic counter
- `views::repeat`
  - Creates a sequence that repeats a single value N times

Range adaptors (selection)

`filter`	Only include elements satisfying the predicate
`transform`	Transform/convert values on the fly
`take`	Take only the N first elements
`take_while`	Take the first elements satisfying the predicate
`drop`	Skip the first N elements
`drop_while`	Skip the first elements satisfying the predicate
`join`	Flatten a view of ranges
`split`	Split into subranges using a delimiter
`reverse`	Iterates a bidirectional view in reverse order
`keys`	Iterates over the first member of a pair-like sequence
`values`	Iterates over the second member of a pair-like sequence

Range adaptors (selection)

`enumerate`	Creates index + value tuples
`zip`	Combine multiple ranges, iterating over tuples
`adjacent`	Iterate over adjacent elements in a sequence
`join_with`	Flatten a range or ranges with a delimiter
`slide`	Iterates over all “sliding windows” of size N
`chunk`	Iterates over blocks of size N
`chunk_by`	Split into subranges/blocks by predicate
`stride`	Iterate over every Nth element
`concat`	Merge multiple ranges into one
`as_const`	Convert to a constant_range
`as_rvalue`	Cast every element to an rvalue

Enumeration

```cpp
std::vector<std::string> v;

// by index
for (size_t i = 0; i != v.size(); ++i)
    fmt::print("entry[{}] = {}\n", i, v[i]);

// with ranges
for (const auto& [idx, value]: v | rv::enumerate)
    fmt::print("entry[{}] = {}\n", idx, value);
```

Keys & values

```
struct Student {
    std::string getName() const;
};

std::map<int, Student> studentsById;

auto studentNames = studentsById | rv::values
                                 | rv::transform(&Student::getName)
                                 | ranges::to_vector;
```

Actions

- Part of ranges-v3
- Not standardized yet?! :-(

- *Eager* application of an algorithm, modifying a container in-place
- Like algorithms, but supporting the *correct by construction* idea

Actions: example

```
struct Student {
    std::string getName() const;
};

std::map<int, Student> studentsById;

auto sortedNames = studentsById | rv::values
                                | rv::transform(&Student::getName)
                                | ranges::to_vector
                                | ranges::actions::sort;
```

Resources

- Standard library: https://en.cppreference.com/w/cpp/ranges
- Range-v3: https://ericniebler.github.io/range-v3/
- Quick reference and examples: https://www.walletfox.com/qttopics.php
- Presentations
  - Eric Niebler CppCon 2015: "Ranges for the Standard Library" (A calendar example): https://www.youtube.com/watch?v=mFUXNMfaciE
  - Tristan Brindle CppCon 2019: "An Overview of Standard Ranges”: https://www.youtube.com/watch?v=SYLgG7Q5Zws
  - Tristan Brindle CppCon 2020: "C++20 Ranges in Practice" (Three examples): https://www.youtube.com/watch?v=d_E-VLyUnzc

Resources

- Belle Views on C++ Ranges, their Details and the Devil - Nico Josuttis - Keynote Meeting C++ 2022
  - https://www.youtube.com/watch?v=O8HndvYNvQ4

- Shows (design) problems with ranges :-(
  - Const-correctness
  - Caching/thread-safety