From STL to Ranges

Jeff Garland

Created: 2019-09-19 Thu 14:07

Intro

the beginning of the end – for begin and end

hello ranges

the old way: `sort`

```
std::array<int, 6> a { 6, 2, 3, 4, 5, 1 };
std::sort ( a.begin(), a.end() );
```

the ranges way: `sort`

```cpp
namespace rng = std::ranges;

std::array<int, 6> a { 6, 2, 3, 4, 5, 1 };
rng::sort ( a ); //clear, obvious meaning, -13 characters
```

the old way: `find_if`

```
std::array<int, 6> a { 6, 2, 3, 4, 5, 1 };
...

auto is_six = [](int i) -> bool { return i == 6; };

// so many beginings and endings
auto i = std::find_if( v.begin(), v.end(), is_six );
if (i != std::end( v ) ) {
    cout << "i: " << *i << endl;
}
```

the ranges way: `find_if`

```
namespace rng = std::ranges;

std::array<int, 6> a { 6, 2, 3, 4, 5, 1 };

auto is_six = [](int i) -> bool { return i == 6; };

auto i = rng::find_if( a, is_six );  //no begin/end
if (i != rng::end( a ) ) {
    cout << "i: " << *i << endl;
}
```

the ranges way: `filter_view`

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

auto is_six = [](int i) -> bool { return i == 6; };

for (int i : rng::filter_view( a, is_six ))
{
  std::cout << i << " ";
}
cout << "\n";
```

talk goals general

- firehose of ranges code
- bootstrap std::ranges use
- how to apply std::ranges
- projections on algorithms
- view versus range versus adaptor

talk outline

- Intro
- Range Basics
- Range Algorithms Details and Survey
- Views and View Adaptor Details
- View types `string_view` & `span`
- Survey of C++20 Views - by example
- Performance
- Resources & Observations

things I will and won’t do in talk

- will: refer to the standard
- **will:** take questions as we go - until we get behind
- will: defer questions I can't answer immediately
- will: no doubt get something wrong
- will: show you lots of code
- will: shorten namespaces (`std::ranges`) and leave out `#includes`
- wont: show you an example that hasn't compiled

the environment

- Ubuntu Linux g++ 8.2, 8.3, 9.1
- typically with -fconcepts and -stdc++20
- range v3 - ranges::cpp20
- cmcstl2 - std::experimental::ranges
- NanoRange - only c++20 ranges

status of ranges implementations

> *“Patience you must have my young padawan”*
>
> - *Yoda*

status of c++20 ranges

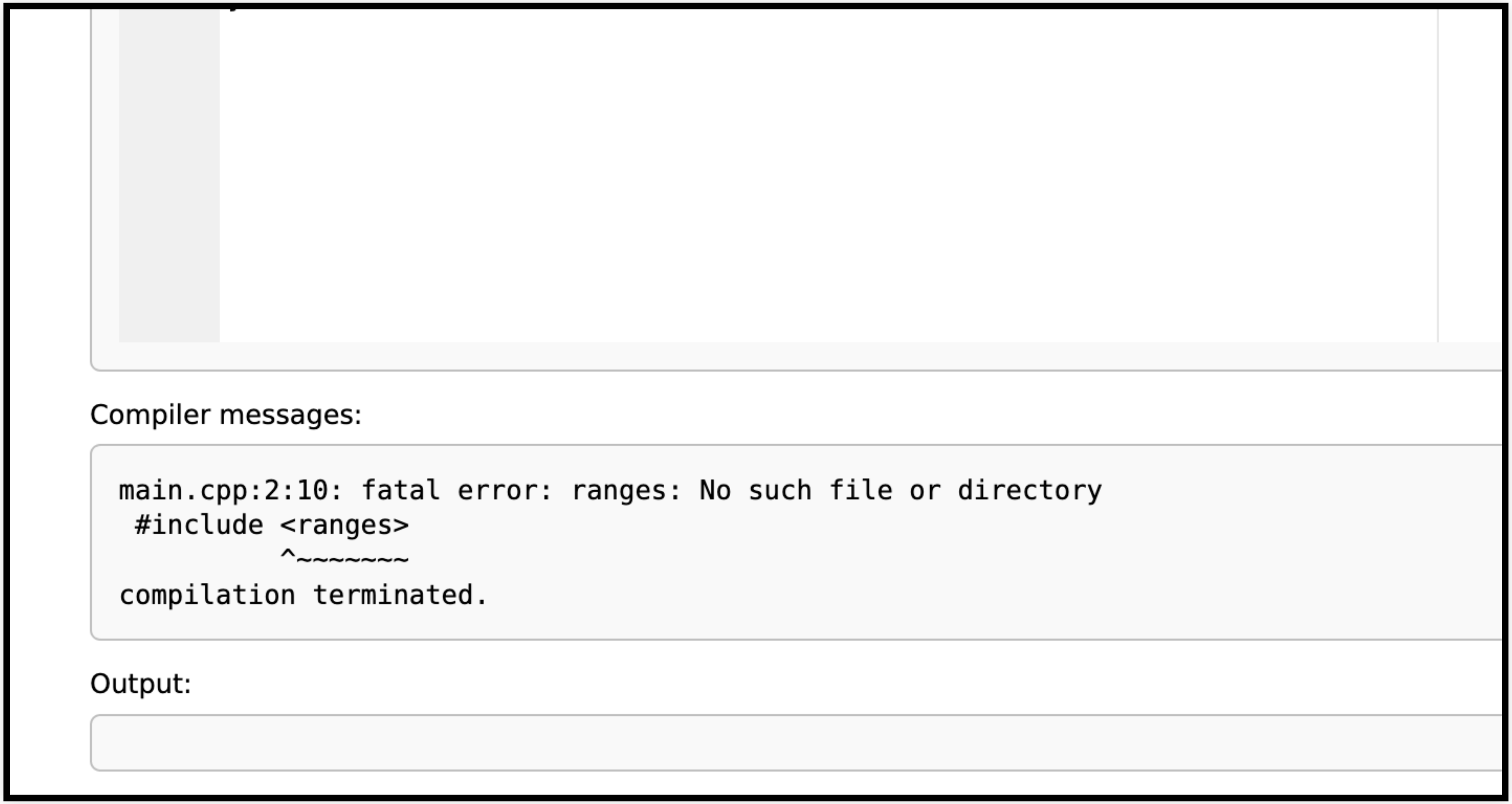
- one ranges proposal P0896
    - 4+ years in the making (see N4130)
    - Added in San Diego 2018
    - not everything - series of follow up papers
- 'design complete' for c++20
- only bug fixes till it ships
- expect vendors to implement quickly

using Ranges now - getting close

- there are still bugs
- How about Godbolt!
  - supports cmcstl2 and v3
  - can pull in nanorange via include

How about cppreference?

- https://en.cppreference.com/w/cpp/ranges



Compiler messages:

```
main.cpp:2:10: fatal error: ranges: No such file or directory
 #include <ranges>
          ^~~~~~~~
compilation terminated.
```

Output:

Range Basics

range, range algorithms, views, adaptors

- range: something that can be iterated over
- range algo: algorithm that takes range
- view: lazy range thats cheap (to copy)
- range adaptor: make a range into a view

mechanics - headers, namespaces

```
#include <ranges> // new header for ranges and views

namespace std {
  namespace ranges {...};           //improved algorithms and views
  namespace ranges::views {...};    //range adaptors
  namespace views = ranges::views; //shortcut to adapters
}
//naming pattern view -> adaptor
// std::ranges::take_view  -> std::views::take

namespace rng = std::ranges; //some examples in this talk
```

why put this in std::ranges instead of just std

- the behavior and guarantees of some algorithms are changed
- it's **not** just a new overload
- would be very confusing if code started behaving differently
- also removing parameters from signatures is difficult

what's a range?

- iterator pair ex: `{ rbegin, rend }`
- technically an iterator and a sentinel
- sentinel and iterator can be different types
- Ranges can now be 'logically infinite'

collections, spans, strings, `string_view`, oh my

- collections 'are' ranges
- strings are ranges
- many items in std library that model the range concept
    - collections: `array`, `vector`, `map`, `set`, `list`
    - !container adapters: `queue`, `stack`, `priority_queue`
    - `fs::directory_iterator`, stream iterators, regex iterators
    - `string_view`, `span`

boost::flat_map example

```cpp
#include <string>
#include <iostream>
#include <range/v3/all.hpp>
#include <boost/container/flat_map.hpp>
namespace rng = ranges::cpp20;
using std::cout, std::vector, std::string,
      boost::container::flat_map;
int
main()
{
  flat_map<string, int> fm;
  fm["world"] = 2;
  fm["hello"] = 1;

  for ( auto [k, v]  : rng::reverse_view{fm} ) {
    cout << k << ":" << v << "\n" ;
  }
  //world:2
  //hello:1
```

range algorithms

- are like STL algorithms
- execute immediately
- iteration is interally controlled by algorithm
- added projection arguments
- return values improved
- not always a drop in replacement

views are ranges with ’lazy evaluation’

- non-owning of elements
- all methods `O(1)` - copy and assignment
- allows for composition of several processing steps
- allows for 'infinite ranges'
- because: iteration is externally driven
- functional or declarative style versus imperative

range adaptors -> views from ranges

- utilities to transform a Range into a View with custom behaviors
- create pipelines of tranformations lazily evaluated
- pipe syntax allows for 'unix pipe/powershell' type composition
- range adaptors are declared in namespace std::ranges::views.
- Example of view composition…later

constructed `filter_view` example

```
#include <experimental/ranges/ranges> //cmcstl2
#include <vector>
#include <iostream>

namespace rng = std::experimental::ranges;

int main() {

  std::vector<int> vi{ 0, 1, 2, 3, 4, 5, 6 };

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

  ranges::filter_view evens{vi, is_even}; //no computation

  for (int i : evens)
  {
     std::cout << i << " "; //0 2 4 6
  }
}
```

`filter_view` example - predicate in loop

```cpp
std::vector<int> vi{ 0, 1, 2, 3, 4, 5, 6 };

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

// put the predicate right in the range for loop
for (int i : ranges::filter_view( vi, is_even ))
{
  std::cout << i << " "; //0 2 4 6
}
```

`views::filter` adpator example

```
std::vector<int> vi{ 0, 1, 2, 3, 4, 5, 6 };

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

// view on stack with adaptor
auto evens = vi | rng::views::filter( is_even );

for (int i : evens)
{
  std::cout << i << " "; //0 2 4 6
}
cout << "\n";
```

filter with algorithm - indirect

```cpp
std::vector<int> vi{ 0, 1, 2, 3, 4, 5, 6 };

auto is_even_print =
  [](int i) { if (0 == i % 2) { cout << i << " "; } };

rng::for_each( vi, is_even_print );
```

collections != views, but are ranges

```
//n4810 range.view
-- examples of Views are:
  — A Range type that wraps a pair of iterators.
  — A Range type that holds its elements by shared_ptr
           and shares ownership with all its copies.
  — A Range type that generates its elements on demand

Most containers (Clause 22) are not views since copying the
container copies the elements, which cannot be done in constant time.
```

no more `for` loops - `for` real?

- Coding guideline: never write a loop
- but: `range for` was created as part of ranges effort
- and yet: c++20 improves for loops again!

```
for ( init-statement(optional) range_declaration : range_expression
     loop_statement
```

views and range algorithm combined

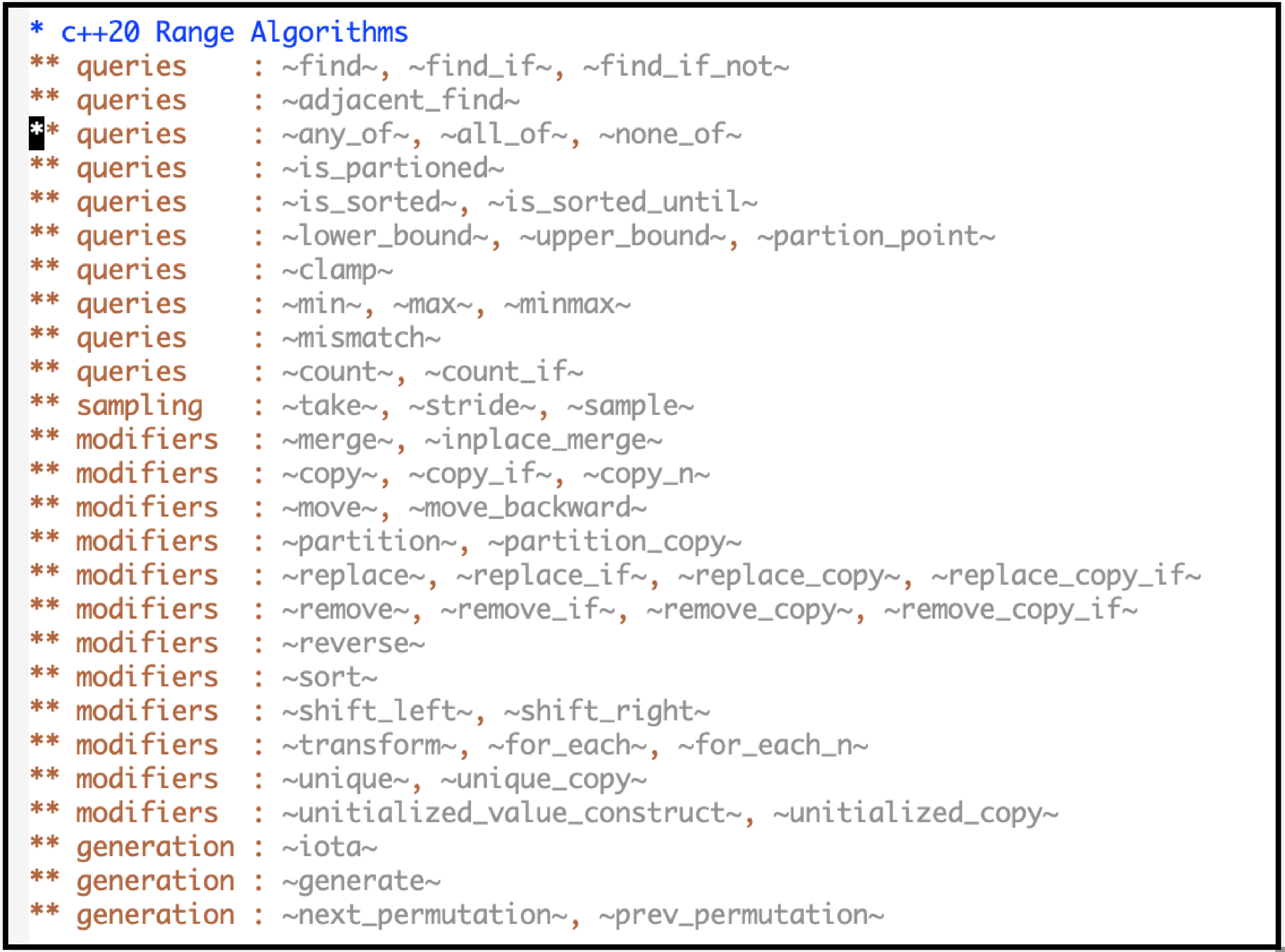
```cpp
//utility functions
auto print   = [](int i) { cout << i << " "; };
auto is_even = [](int i) { return i % 2 == 0; };

vector<int> v { 6, 2, 3, 4, 5, 6 };

//view is defined -- no calculation performed
auto after_leading_evens = rng::views::drop_while(v, is_even);

//Now drive the iteration/calculation
rng::for_each(after_leading_evens, print); //prints-> 3 4 5 6 7
cout << endl;
```

Range Algorithms Details and Survey

The range algorithms are familiar

- derived from the STL algorithms, but better
- improved return information compared to STL for some algos
- specified with concepts

algorithms cheatsheet



```
* c++20 Range Algorithms
** queries    : ~find~, ~find_if~, ~find_if_not~
** queries    : ~adjacent_find~
** queries    : ~any_of~, ~all_of~, ~none_of~
** queries    : ~is_partioned~
** queries    : ~is_sorted~, ~is_sorted_until~
** queries    : ~lower_bound~, ~upper_bound~, ~partion_point~
** queries    : ~clamp~
** queries    : ~min~, ~max~, ~minmax~
** queries    : ~mismatch~
** queries    : ~count~, ~count_if~
** sampling   : ~take~, ~stride~, ~sample~
** modifiers  : ~merge~, ~inplace_merge~
** modifiers  : ~copy~, ~copy_if~, ~copy_n~
** modifiers  : ~move~, ~move_backward~
** modifiers  : ~partition~, ~partition_copy~
** modifiers  : ~replace~, ~replace_if~, ~replace_copy~, ~replace_copy_if~
** modifiers  : ~remove~, ~remove_if~, ~remove_copy~, ~remove_copy_if~
** modifiers  : ~reverse~
** modifiers  : ~sort~
** modifiers  : ~shift_left~, ~shift_right~
** modifiers  : ~transform~, ~for_each~, ~for_each_n~
** modifiers  : ~unique~, ~unique_copy~
** modifiers  : ~unitialized_value_construct~, ~unitialized_copy~
** generation : ~iota~
** generation : ~generate~
** generation : ~next_permutation~, ~prev_permutation~
```

`for_each` string example

```
string s { "hello" };
//output: h e l l o
rng::for_each( s, [](char c){ cout << c << " "; } );

vector<int> v { 6, 2, 3, 4, 5, 6 };
int c = rng::count( v, 6 ); // c == 2

vector<int> v { 1, 2, 3, 4, 5, 6 };
if (rng::is_sorted( v )) { cout << "true" << endl; };
```

`minmax` element example

```
{
  //returns an iterator
  auto i = rng::min_element( v );
  if (i != v.end()) { //check it
    cout << "min element " << *i; //deref the iterator
  }
  cout << "\n";
}

{
  //structured bindings...c++17...these are iterators
  auto [min_value, max_value] = rng::minmax_element( v );
  cout << "min ele: " << *min_value << " max ele: " << *max_value << "\
}
```

`copy_if` example

```cpp
auto is_six = [](int i) -> bool { return i == 6; };
auto print = [] (int i) { cout << i << " "; };

int main() {

  vector<int> v { 1, 2, 3, 4, 5, 6};
  {
    // copy from one container to another
    vector<int> v_cpy;
    rng::copy_if(v, rng::back_inserter(v_cpy), is_six);
    rng::for_each( v_cpy, print ); //6
    cout << "\n";
  }
}
```

sort example

```
std::array<int, 6> arr { 6, 2, 3, 4, 5, 1 };
rng::sort ( arr );

deque<int> d { 6, 2, 3, 4, 5, 1 };
auto reverse_compare = [](int i, int j) { return i > j; };
rng::sort ( d, reverse_compare ); // 6 5 4 3 2 1

list<int> li { 6, 2, 3, 4, 5, 1 };
rng::sort ( li );
```

Projection parameters

- Projections provide first class filtering predicate independent of function to be invoked
- Comes for the Adobe Source Libraries (ASL)

for example - sort with projection

```cpp
struct stuff {
   int idx = 0;
   string s;
};

vector<stuff> vstuff = {{2, "foo"}, {1, "bar"}, {0, "baz"}};

ranges::sort(vstuff, std::less<>{},
             [](auto const& iii) { return iii.idx; });
```

Views and View Adaptor Details

range adaptor equivalence

- from `range.object.adaptor`

```
// range.adaptor.object
// range adaptor object with more than one argument,
// then the following are equivalent:

 adaptor (range, args...)
 adaptor (args...) (range)
 range | adaptor (args...)
```

Range adaptors examples -> make range a view

```
   //example from standard range.adaptors
  vector<int> vec_int{ 0, 1, 2, 3, 4, 5 };

  auto even   = [](int i) -> bool { return 0 == i % 2; };
  auto square = [](int i) -> int  { return i * i; };

  for (int i : vec_int
         | ranges::views::filter(even)
         | ranges::views::transform(square))
  {
    cout << i << " "; //0 4 16
  }
  if (ranges::equal(
        vec_int | ranges::views::filter(even), //piped collection
        ranges::views::filter(vec_int, even))  //param collection
      )
  {
    cout << "\n" << "equal!!\n"; //will execute
  }
}
```

range adaptors execution trace

```
int
main()
{
  vector<int> vec_int{ 0, 1, 2, 3, 4, 5 };

  //annotated
  auto even =   [](int i) -> bool { cout << "ev:" << i << "\n";
                                     return 0 == i % 2;
                                    };
  auto square = [](int i) -> int  { cout << "sq:" << i << "\n";
                                     return i * i; };

  for (int i : vec_int
         | ranges::views::filter(even)
         | ranges::views::transform(square)) //how many executions
  {
    cout << "for: "<< i << "\n"; //how many executions?
  }
}
```

range adaptors execution trace - output

```
// ev:0
// sq:0
// for: 0
// ev:1
// ev:2
// sq:2
// for: 4
// ev:3
// ev:4
// sq:4
// for: 16
// ev:5

for:   <- 3
square <- 3
even   <- 6
```

Types in a view chain - under the hood

```
//typename magic
template <class T>
constexpr
std::string_view
type_name()
{
  //gcc only version see  https://stackoverflow.com/questions/81870/
  std::string_view p = __PRETTY_FUNCTION__;
  return std::string_view(p.data() + 49, p.find(';', 49) - 49);
}
```

Types in a view chain - under the hood

```
int
main()
{

  vector<string> vs{"hello", " ", "ranges", "!"};
  cout << type_name<decltype(vs)>() << "\n";
  //vector<basic_string<char> >

  auto jv = join_view{vs};
  cout << type_name<decltype(jv)>() << "\n";
  //ranges::join_view<ranges::ref_view<vector<basic_string<char> > > >

  take_view tv{jv, 2};
  cout << type_name<decltype(tv)>() << "\n";
  //ranges::take_view<ranges::join_view<ranges::ref_view<vector<basic_s
```

View types `string_view` & `span`

- not technically part of ranges
- designed to inter-operate
- satisfy `contiguous_range` concept

`string_view`

- c++17 view around strings
- view wrapper for string types
- support interface similar to `std::string`

`span` (C++20)

- span is a non-owning 'view' over contiguous sequence of object
- cheap to copy - implementation is a pointer and size
- constant time complexity for all member functions
- defined in header `<span>`
- unlike most 'view types' can mutate
- constexpr ready

span construction

```
vector<int> vi = { 1, 2, 3, 4, 5 };
span<int>   si ( vi );

array<int, 5> ai = { 1, 2, 3, 4, 5 };
span<int>     si2 ( ai );

int cai[] = { 1, 2, 3, 4, 5 };
span<int>     si3( cai );
```

span as a function parameter

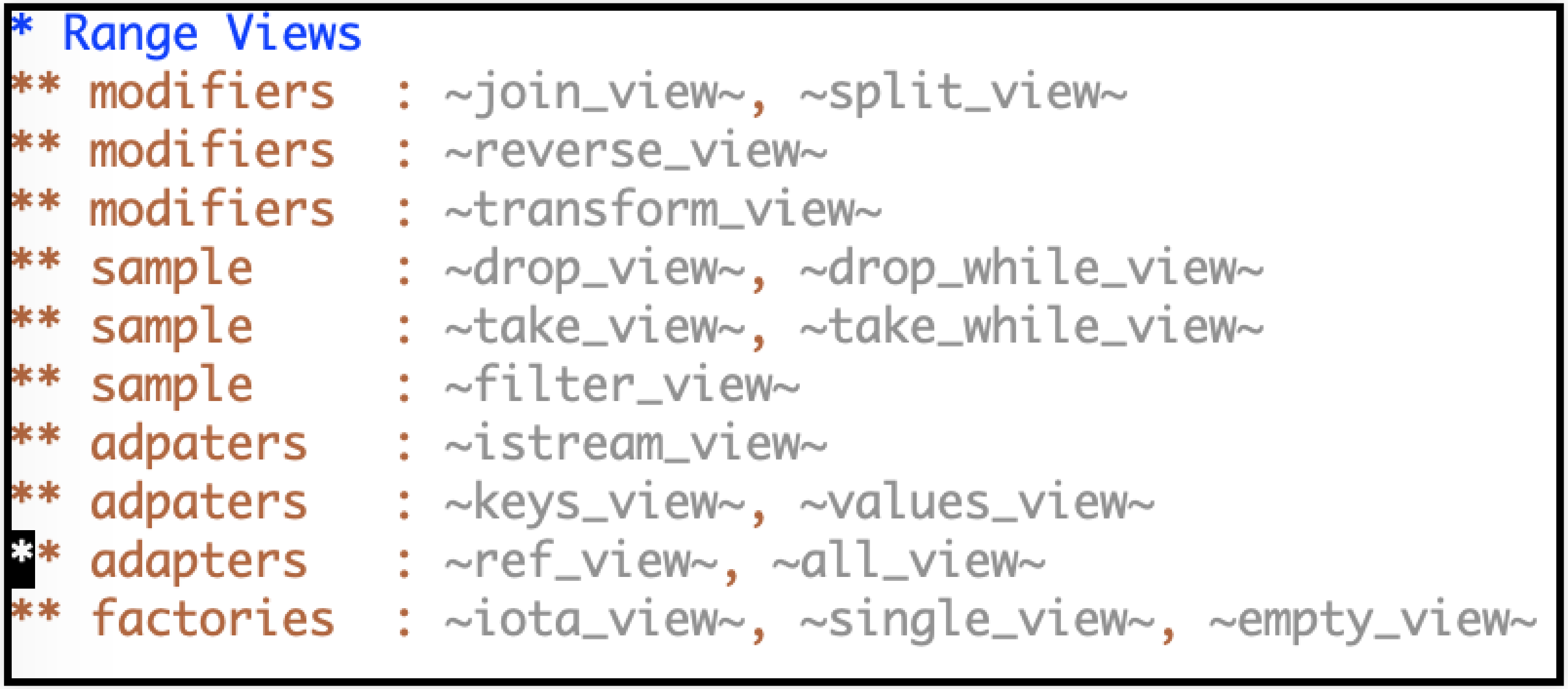
```
void print_reverse(span<int> si) { //by value
  for ( auto i  : ranges::reverse_view{si} ) {
    cout << i << " " ;
  }
  cout << "\n";
}

int main () {

  vector<int> vi = { 1, 2, 3, 4, 5 };
  print_reverse( vi ); //5 4 3 2 1

  span<int> si ( vi );
  print_reverse( si.first(2) ); //2 1
  print_reverse( si.last(2) );  //5 4
```

basic public accessors

- supports collection-like interfaces (`begin`, `end`, `operator[]`)
- `data` pointer to start of sequence
- `size` number of elements
- `size_bytes` memory size
- `empty` true if no elements

Survey of Views - by example

cheatsheet



```
* Range Views
** modifiers  : ~join_view~, ~split_view~
** modifiers  : ~reverse_view~
** modifiers  : ~transform_view~
** sample     : ~drop_view~, ~drop_while_view~
** sample     : ~take_view~, ~take_while_view~
** sample     : ~filter_view~
** adpaters   : ~istream_view~
** adpaters   : ~keys_view~, ~values_view~
** adapters   : ~ref_view~, ~all_view~
** factories  : ~iota_view~, ~single_view~, ~empty_view~
```

iota_view example

```
// range.iota.overview iota_view generates a sequence of elements
// by repeatedly incrementing an initial value.
for (int i : ranges::iota_view{1, 5})
{
  cout << i << " "; //1 2 3 4
}
```

take_view example

```
vector<int> vi {0, 1, 2, 3, 4, 5};

// range.take.overview
// take_view produces a View of the first N elements from another View,
// or all the elements if the adapted View contains fewer than N.
{
  ranges::take_view tv{vi, 2};
  for (int i : tv) {
    cout << i << " "; //0 1
  }
}
{
  auto tv = ranges::take_view{vi, 2}; //construct via assignment
  for (int i : tv) {
    cout << i << " "; //0 1
  }
}
{
  ranges::take_view tv{vi, 10}; //10 is greater then vi.size()
  for (int i : tv) {
    cout << i << " "; //0 1 2 3 4 5
  }
}
```

`join_view` char string example

```
vector<string> vs{"hello", " ", "ranges", "!"};

// range.join.overview join_view flattens a View of
// ranges into a View
for ( char ch : ranges::join_view{vs} ) {
  cout << ch; // hello ranges!
}
```

transform_view example

```
//using cmcstl2 this is successful (compile errors with range v3)
vector<int> vi{ 0, 1, 2, 3, 4, 5 };

rng::transform_view times_ten{ vi, [](int i) { return i * 10; } };

for (int i : times_ten)
{
  cout << i << " ";  // 0 10 20 30 40 50
}
cout << "\n";
```

drop_while example

```
vector<int> v { 6, 2, 3, 4, 5, 6 , 7 };

auto is_even = [](int i) { return i % 2 == 0; };
auto print   = [](int i) { cout << i << " "; };

auto after_leading_evens =
         rng::views::drop_while(v, is_even) |
         rng::views::take(2);

//execute
rng::for_each(after_leading_evens, print);  //prints-> 3 4

cout << "\n";
```

`istream_view` example

```
std::istringstream ints{"0 1 2 3 4"};

for (int i : ranges::istream_view<int>(ints)) {
    cout << i << "-"; // prints 0-1-2-3-4-
}
```

empty_view example

```
// range.empty.overview empty_view produces a View of no elements
// of a particular type.
ranges::empty_view<float> ev;
static_assert(ranges::empty(ev));
static_assert(0 == ev.size());

for (float f : ev)
{
  cout << "unreached:" << f << endl;
}
```

single_view example

```
//range.single.overview single_view produces a View that
//contains exactly one element of a specified value.
ranges::single_view svi{42};
for (int i : svi) cout << i; // 42

ranges::single_view ssv{string{" the answer to everything"} };
for (string s : ssv) cout << s; // the answer to everything
```

split_view char string example

```
string answer{"42 the answer to everything"};

// range.split.overview split_view takes a View and a
// delimiter, and splits the View into subranges on the
// delimiter. The delimiter can be a single element or
// a View of elements.
ranges::split_view words{answer, ' '};

//return is a ref_view<string>
for (auto word : words)
{
  for (char ch : word) //need to break down to elements
  {
    cout << ch;
  }
  cout << "-";
}
//42-the-answer-to-everything-
```

Performance

How do ranges perform?

- compile time
  - using range v3 in production for 2 years
  - mostly using range algorithms
  - no formal measurement - no noticeable difference
- runtime
  - have not seen so far, but application limited
  - A theory: range algorithms will be same as STL
  - why? just algorithms
  - `boost.range` – for a decade+ (`string_algo` particularly)
  - views allow for writing efficient code - avoid naive copies, for example

Resources

reference implementations

- Eric Range V3
  - https://github.com/ericniebler/range-v3
  - `git clone -b v1.0-beta https://github.com/ericniebler/range-v3.git`
- Casey cmcstl2
  - https://github.com/CaseyCarter/cmcstl2
  - `git clone https://github.com/CaseyCarter/cmcstl2`
- Tristan Brindle NanoRange
  - https://github.com/tcbrindle/NanoRange
  - `git clone https://github.com/tcbrindle/NanoRange`

range examples

- range v3 help https://ericniebler.github.io/range-v3/
- github (needs update) https://github.com/JeffGarland/range_by_example

C++ working draft and papers

- Latest draft standard http://wg21.link/n4810
- Unless otherwise noted most algorithms were added with the 'one ranges' proposal.
- One ranges paper (226 pages) http://wg21.link/p0896
- Standard Library Concepts 2018 Niebler, Carter http://wg21.link/p0898
- Ranges proposal V1 http://wg21.link/N4128

C++ working draft and papers - most recent

- input range adaptors (post Kona) http://wg21.link/p1035
- Kona Move Only Views http://wg21.link/p1456
- Post Kona new algorithms http://wg21.link/p1243
- ranges:to (not in c++20) http://wg21.link/p1206

Projections on algorithms

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers take-invokable-projections
- Reddit conversation https://www.reddit.com/r/cpp/comments/7jx1z7/stl_a
- From the Adobe Source Libraries (ASL) http://stlab.adc
- From S. Parent C++ Seasoning Talk https://github.com/ parent.github.com/wiki/presentations/2013-09-11-cp

Videos and Blogs

- Eric c++now Library Design Talk 2014 https://www.youtube.com/watch?v=zgOF4NrQllo
- Eric cppcon 2015 https://www.youtube.com/watch?v=mFUXNMfaciE
- Chris Debella Blog post https://www.cjdb.com.au/a-prime-opportunity-for-ranges

span resources

- in the working paper
- http://wg21.link/P0122
- implementation https://github.com/tcbrindle/span
- should span be regular http://wg21.link/p1085

Observations

- Ranges is key building blocks for the future
- overall: nicer cleaner more capable code
- begin ~~ing~~ not an end

Finish

```
ranges::single_view ssv{"  Thank You!!  \n"s};
for (string s : ssv) cout << s;
```