Effective Ranges: A tutorial for using C++2x ranges

Jeff Garland

Created: 2023-10-03 Tue 08:50

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() );

for ( auto v : a )
   cout << v << " " << endl;
```

the ranges way: `sort`

```
namespace rng = std::ranges;

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

rng::sort ( a );          //clear, obvious meaning, -13 characters

std::print( "{}", a );  //[1, 2, 3, 4, 5, 6]
```

- https://godbolt.org/z/o478PEMff

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 ) )
{
    std::print( "i: {}\n", *i ); //i: 6
}
```

- https://godbolt.org/z/4KEM3W939

the ranges way: `filter_view`

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

using std::views::filter;
auto divisible_by3 = filter([](int i) -> bool { return i%3 == 0; } );

std::print("{}", a | divisible_by3 ); //[6, 3]
```

- https://godbolt.org/z/bnchn9jaf

talk outline

- range basics & concepts
- range algo details and survey
- range views details and survey
- adjuncts
- observations & futures

Range Basics

ranges, 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 adaptors
}

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

what's a range?

- iterator pair ex: `{ rbegin, rend }`
- an object with `begin()`, `end()` that returns iterators
- technically an iterator and a sentinel
- sentinel and iterator can be different types

Examples of ranges

- standard library things
  - containers: `array`, `vector`, `map`, `set`, `list`
  - container adaptors: `queue`, `stack`, `priority_queue`
  - strings: `string` and `string_view`
  - `fs::directory_iterator`, stream iterators, regex iterators
  - `span`, `mdspan`
- many libraries outside std

`boost::flat_map` example

```
flat_map<string, int> fm;
fm["world"] = 2;
fm["hello"] = 1;

std::print("{}\n", fm); // {"hello": 1, "world": 2}

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

- https://godbolt.org/z/Mdqh58zrM

range algorithms

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

range concepts: sized, etc

- `range` - provides a begin iterator and an end sentinel
- `input_range` - specifies that a range has InputIterator
- `sized_range` - specifies that a range knows its size in constant time
- `random_access_range` - specifies a range whose iterator type satisfies `random_access_iterator`
- `contiguous_range` - specifies a range whose iterator type satisfies `contiguous_iterator`

Range Algorithms Details and Survey

range algorithms are familiar

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

cheatsheet

```
* c++2x Range Algorithms
** queries     : ~find~, ~find_if~, ~find_if_not~
** queries     : ~find_last~, ~find_last_if~, ~find_last_if_not~ (23)
** queries     : ~adjacent_find~
** queries     : ~any_of~, ~all_of~, ~none_of~
** queries     : ~contains~, ~contains_subrange~ (23)
** queries     : ~is_partioned~
** queries     : ~is_sorted~, ~is_sorted_until~
** queries     : ~lower_bound~, ~upper_bound~, ~partion_point~
** queries     : ~clamp~
** queries     : ~mismatch~
** queries     : ~starts_with~, ~ends_with~ (23)
** sampling    : ~copy~, ~copy_if~, ~copy_n~
** sampling    : ~rotate_copy~, ~unique_copy~
** sampling    : ~stride~, ~sample~, ~take~
** modifiers   : ~merge~, ~inplace_merge~
** 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~
```

cheatsheet

```
** modifiers   : ~shuffle~, ~sort~
** modifiers   : ~shift_left~, ~shift_right~ (23)
** modifiers   : ~transform~, ~for_each~, ~for_each_n~
** modifiers   : ~unique~
** modifiers   : ~unitialized_value_construct~, ~unitialized_copy~
** calculation : ~count~, ~count_if~
** calculation : ~fold_left~, ~fold_left_first~, ~fold_right~, ~fold_right_last~ (23)
** calculation : ~min~, ~max~, ~minmax~
** generation  : ~iota~ (23)
** generation  : ~generate~
** generation  : ~next_permutation~, ~prev_permutation~
```

projection parameters

- filtering predicate independent of function being invoked
- comes for the Adobe Source Libraries (ASL)

for example - sort with projection

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

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

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

//short form
ranges::sort( vstuff, std::less<>{}, &stuff::idx );
```

- https://godbolt.org/z/8cvoEvd11

improved return values

- principle of not discarding calculation
- many ranges algorithm return a struct with iterator+info
- `find_last` example
  - find with reverse iterators is suprisingly difficult
  - boost.algorithm has one already call `find_backward`
  - what should the return value be?

`find_last` example

```
std::string path = "/foo/bar/baz";

auto leaf = rng::find_last(path, '/')
                | std::views::drop(1);
std::print("{}\n", std::string_view(leaf)); // baz

path = "none";
leaf = rng::find_last(path, '/')
                | std::views::drop(1); //empty range
std::print("empty: {}", std::string_view(leaf)); //empty:
```

- returns a range
- https://godbolt.org/z/8KPjqYrG1

Views and Adaptor Details

views are ranges with ’lazy evaluation’

- non-owning of elements (mostly)
- 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

constructed `filter_view` example

```
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
  }
}
```

range adaptors -> views from ranges

- transform Range to View - with custom behaviors
- adaptors support the piping operator
- pipe syntax allows for 'unix pipe/powershell' type composition
- range adaptors are declared in namespace std::ranges::views.

`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 << " ";
}
cout << "\n";
```

range adaptors execution trace

```
  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?
  }
}
```

- https://godbolt.org/z/1863ce6sW

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

```
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_string<char> > > > >
```

- https://godbolt.org/z/G5PE614ff

views and const iteration

- perhaps suprisingly shouldn't pass by `const&`
- why? views like `filter_view` cache
- prefer to pass by value or forwarding reference
- https://quuxplusone.github.io/blog/2023/08/13/non-const-iterable-ranges/

```
template<std::ranges::range R>
void doit( const R& range); // nope

template<std::ranges::range R>
void doit( R&& range); // yes!
```

composition: views and range algorithm example

```
//utility functions
auto print_int = [](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);

rng::for_each(after_leading_evens, print_int); //now drive the iteratio
cout << endl;
```

- https://godbolt.org/z/c137bM8f4

Survey of Views - by example

cheatsheet

```
* Range Views
** modifiers  : ~join_view~, ~join_with~ (23)
** modifiers  : ~split_view~, ~lazy_split_view~
** modifiers  : ~reverse_view~
** modifiers  : ~transform_view~
** modifiers  : ~cartesian_product~ (23)
** modifiers  : ~zip~, ~zip_transform~ (23)
** sample     : ~drop_view~, ~drop_while_view~
** sample     : ~take_view~, ~take_while_view~
** sample     : ~filter_view~, ~stride~ (23)
** sample     : ~chunk~, ~chunk_by~ (23)
** adpaters   : ~istream_view~
** adpaters   : ~keys_view~, ~values_view~
** adapters   : ~ref_view~, ~all_view~
** adapters   : ~enumerate~ (23)
** factories  : ~iota_view~, ~single_view~, ~empty_view~
** factories  : ~repeat~
** convert    : ~as_const~ (23), ~as_rvalue~ (23)
```

`cartesian_product` example (23)

```
/* output
0 2
0 3
1 2
1 3
*/
namespace rv = ranges::views;

    std::vector<int> v1 { 0, 1 };
    std::vector<int> v2 { 2, 3 };
    for (auto&& [a,b] : rv::cartesian_product(v1, v2))
    {
        std::print("{} {}\n",  a, b);
    }
```

- https://godbolt.org/z/rYehsTMxY

`chunk_by` (23) example

```
std::vector v = {1, 2, 2, 3, 0, 4, 5, 2};
 // [[1, 2, 2, 3], [0, 4, 5], [2]]
std::print("{}\n", v | std::views::chunk_by( ranges::less_equal{} ));
```

`zip` and `zip_transform` (23)

- bind multiple ranges with pairwise matching
- especially important for dealing `tuple` and `pair`

`zip` example

```
std::vector v1 = {1, 2};
std::vector v2 = {'a', 'b', 'c'};
std::vector v3 = {3, 4, 5};

// {(1, 'a'), (2, 'b')}
std::print("{}\n", std::views::zip(v1, v2));
// {3, 8}
std::print("{}\n", std::views::zip_transform(std::multiplies(), v1, v3)
// {('a', 'b'), ('b', 'c')}
std::print("{}\n", v2 | std::views::pairwise);
// {7, 9}
std::print("{}\n", v3 | std::views::pairwise_transform(std::plus()));
```

Adjuncts

`span`

- 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
```

ranges::to

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

auto squared = numbers
    | ranges::views::transform( [](int n) { return n * n; } )
    | ranges::to<std::deque>();  // Convert to a deque<int>

std::print("{}", squared); //[1, 4, 9, 16, 25]
```

- https://godbolt.org/z/c137bM8f4

`ranges::to` string example

```
namespace rv = ranges::views;

vector<string> vs = {"foo", "bar", "baz", ""};

string s = vs | rv::join(string_view(" -- "))
              | ranges::to<string>();
cout << s << "\n"; //foo -- bar -- baz --

auto is_not_empty = [](const string& s) { return !s.empty(); };

s = vs | rv::filter(is_not_empty)
       | rv::join('-')
       | ranges::to<string>();
cout << s << "\n"; //foo-bar-baz
```

- https://godbolt.org/z/bd558h47j

range constructors and other operations

- collections and string gain range constructors
- `insert_range`, `assign_range`, `append_range`, `prepend_range`
- https://wg21.link/P1206

```
template<container-compatible-range<T> R>
constexpr vector(from_range_t, R&& rg);
```

writing a view

- write your own
- make it pipeable
- takes a lot of boiler plate
- https://www.boost.org/doc/libs/1_83_0/doc/html/stl_i

Observations and Future

sum of squares

```
auto square = [](int x) {return x*x;};
auto add =    [](int x, int y) {return x+y;};

auto square_view = rv::iota(0)  //infinite range
                 | rv::transform(square)
                 | rv::take(5);

int total = rng::fold_left( square_view, 0, add );

print("{}\n", total); //30

// [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16)]
print("{}\n", square_view | rv::enumerate );
```

- https://godbolt.org/z/qPh1hjdzo

C++26

- algos: `reduce`, `sum`, `product`
- views: `concat`, `cycle`, `getlines`
- views: `drop_last`, `take_last`, `drop_exactly`, `take_exactly`
- views: `split_when`
- views: `remove`, `remove_if`, `replace`, `replace_if`
- views: `transform_join`
- and a lot more…
- https://wg21.link/P2214

advice

- always use range algorithms first
  - constrained, better errors
  - projections, superior results
- view adapters first
- don't be evil and write inscrutable pipelines

final thoughts

- 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;
```