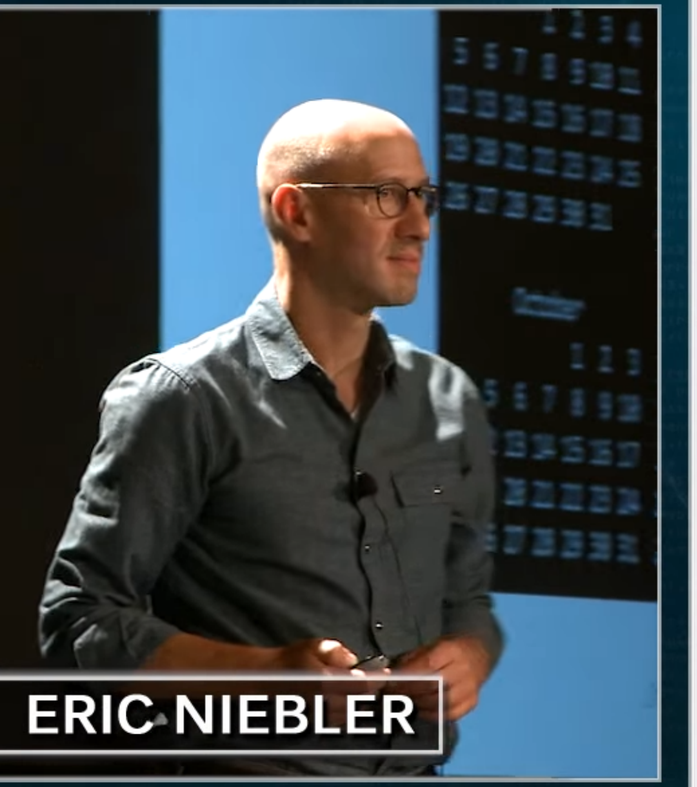
accu 2023

C++23 RANGES

CONCEPTUAL CHANGES AND USEFUL PRACTICALITIES

DVIR YITZCHAKI

https://dvirtz.github.io/slides/ranges_23/ranges.html



Ranges for the Standard Library

Ta-da!

```
$ ./example/calendar.exe
      January                 February                  March
                1  2  3   1  2  3  4  5  6  7   1  2  3  4  5  6  7
 4  5  6  7  8  9 10   8  9 10 11 12 13 14   8  9 10 11 12 13 14
11 12 13 14 15 16 17  15 16 17 18 19 20 21  15 16 17 18 19 20 21
18 19 20 21 22 23 24  22 23 24 25 26 27 28  22 23 24 25 26 27 28
25 26 27 28 29 30 31                       29 30 31

       April                   May                     June
          1  2  3  4                      1  2      1  2  3  4  5  6
 5  6  7  8  9 10 11   3  4  5  6  7  8  9   7  8  9 10 11 12 13
12 13 14 15 16 17 18  10 11 12 13 14 15 16  14 15 16 17 18 19 20
19 20 21 22 23 24 25  17 18 19 20 21 22 23  21 22 23 24 25 26 27
26 27 28 29 30        24 25 26 27 28 29 30  28 29 30
                      31
       July                    August                  September
          1  2  3  4                         1            1  2  3  4  5
 5  6  7  8  9 10 11   2  3  4  5  6  7  8   6  7  8  9 10 11 12
12 13 14 15 16 17 18   9 10 11 12 13 14 15  13 14 15 16 17 18 19
19 20 21 22 23 24 25  16 17 18 19 20 21 22  20 21 22 23 24 25 26
26 27 28 29 30 31     23 24 25 26 27 28 29  27 28 29 30
                      30 31
      October                 November                December
                1  2  3   1  2  3  4  5  6  7            1  2  3  4  5
 4  5  6  7  8  9 10   8  9 10 11 12 13 14   6  7  8  9 10 11 12
11 12 13 14 15 16 17  15 16 17 18 19 20 21  13 14 15 16 17 18 19
18 19 20 21 22 23 24  22 23 24 25 26 27 28  20 21 22 23 24 25 26
25 26 27 28 29 30 31  29 30                 27 28 29 30 31
```

RECAP

Source: https://discovernorthernireland.com/

WHAT IS A RANGE

- A sequence of elements between two locations `i`, `k`.
- Often denoted by `[i, k)`.

Although Concepts are constraints on types, you don't find them by looking at the types in your system.

You find them by studying the algorithms.

Eric Neibler

std::for_each

```
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```

std::for_each

```
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```

ITERATOR CONCEPTS

`input_or_output_iterator` can be dereferenced (`*it`) and incremented (`++it`)

ITERATOR CONCEPTS

`input_or_output_iterator`	can be dereferenced (`*it`) and incremented (`++it`)
`input_iterator`	referenced values can be read (`auto v = *it`)
`output_iterator`	referenced values can be written to (`*it = v`)
`forward_iterator`	input_iterator + comparable and multi-pass
`bidirectional_iterator`	forward_iterator + decrementable (`--it`)
`random_access_iterator`	bidirectional_iterator + random access (`it += n`)
`contiguous_iterator`	random_access_iterator + contiguous in memory

std::for_each

```
template<class InputIt, class UnaryFunction>
constexpr UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```

SENTINELS

- a semi-regular `S` is a `sentinel_for` an iterator `I` if for a range `[i, s)`
  - `i == s` is well defined.
  - If `i != s` then `i` is dereferenceable and `[++i, s)` denotes a range.
- Such an `s` is called a sentinel

RANGE

a type we can feed to

- `ranges::begin` - to get an iterator
- `ranges::end` - to get a sentinel

A range `[i, s)` refers to the elements

`*i, *(i++), *((i++)++), …, j`

such that j == s.

RANGE CONCEPTS

`input_range`	e.g. a range over a `std::istream_iterator`
`output_range`	e.g. a range over a `std::back_insert_iterator`
`forward_range`	e.g. `std::forward_list`
`bidirectional_range`	e.g. `std::list`
`random_access_range`	e.g. `std::deque`
`contiguous_range`	e.g. `std::vector`
`common_range`	sentinel is same type as iterator

ALGORITHMS

now constrained with concepts:

```
namespace std::ranges {
template<input_iterator I, sentinel_for<I> S,
         class Proj = identity,
         indirectly_unary_invocable<projected<I, Proj>> Fun>
constexpr for_each_result<I, Fun>
for_each(I first, S last,
         Fun f,
         Proj proj = {});

template<input_range R,
         class Proj = identity,
         indirectly_unary_invocable<projected<iterator_t<R>, Proj>> Fun>
constexpr for_each_result<borrowed_iterator_t<R>, Fun>
for_each(R&& r,
         Fun f,
         Proj proj = {});
}
```

ALGORITHMS

now constrained with concepts:

```
namespace std::ranges {
template<input_iterator I, sentinel_for<I> S,
         class Proj = identity,
         indirectly_unary_invocable<projected<I, Proj>> Fun>
constexpr for_each_result<I, Fun>
for_each(I first, S last,
         Fun f,
         Proj proj = {});

template<input_range R,
         class Proj = identity,
         indirectly_unary_invocable<projected<iterator_t<R>, Proj>> Fun>
constexpr for_each_result<borrowed_iterator_t<R>, Fun>
for_each(R&& r,
         Fun f,
         Proj proj = {});
}
```

ALGORITHMS

now constrained with concepts:

```
namespace std::ranges {
template<input_iterator I, sentinel_for<I> S,
         class Proj = identity,
         indirectly_unary_invocable<projected<I, Proj>> Fun>
constexpr for_each_result<I, Fun>
for_each(I first, S last,
         Fun f,
         Proj proj = {});

template<input_range R,
         class Proj = identity,
         indirectly_unary_invocable<projected<iterator_t<R>, Proj>> Fun>
constexpr for_each_result<borrowed_iterator_t<R>, Fun>
for_each(R&& r,
         Fun f,
         Proj proj = {});
}
```

The law of useful return:

A procedure should return all the potentially useful information it computed.

Alexander Stepanov

RANGE

a type we can feed to

- `ranges::begin` - to get an iterator
- `ranges::end` - to get a sentinel

A range `[i, s)` refers to the elements

`*i, *(i++), *((i++)++), …, j`

such that j == s.

RETURN VALUE

```
namespace std::ranges {
template<class I, class F>
struct for_each_result {
  [[no_unique_address]] I in;
  [[no_unique_address]] F fun;

  template<class I2, class F2>
  requires convertible_to<const I&, I2> && convertible_to<const F&, F2>
  operator for_each_result<I2, F2>() const & {
    return {in, fun};
  }

  template<class I2, class F2>
  requires convertible_to<I, I2> && convertible_to<F, F2>
  operator for_each_result<I2, F2>() && {
    return {std::move(in), std::move(fun)};
  }
};
}
```

VIEWS

A lightweight™ handle to a range.

- A range type that wraps a pair of iterators.
- A range type that holds its elements by `std::shared_ptr` and shares ownership with all its copies.
- A range type that generates its elements on demand.

EXAMPLE

```
namespace std::ranges {
template<class T>
requires is_object_v<T>
class empty_view : public view_interface<empty_view<T>> {
public:
  static constexpr T* begin() noexcept { return nullptr; }
  static constexpr T* end() noexcept { return nullptr; }
  static constexpr T* data() noexcept { return nullptr; }
  static constexpr size_t size() noexcept { return 0; }
  static constexpr bool empty() noexcept { return true; }

  friend constexpr T* begin(empty_view) noexcept { return nullptr; }
  friend constexpr T* end(empty_view) noexcept { return nullptr; }
};
}
```

FACTORIES & ADAPTORS

- Utility objects for creating views
  - adaptors (if transform an existing range) or
  - factories (otherwise)

```
1 namespace std::ranges::views {
2 template<class T>
3 inline constexpr empty_view<T> empty{};
4 }
5
6 static_assert(std::ranges::empty(std::views::empty<int>));
```

CLASSIC STL

```
int sum_of_squares(int count) {
  using namespace std;

  vector<int> numbers(static_cast<size_t>(count));
  iota(numbers.begin(), numbers.end(), 1);
  transform(numbers.begin(), numbers.end(), numbers.begin(),
                [](int x) { return x * x; });
  return accumulate(numbers.begin(), numbers.end(), 0);
}
```

FUNCTION CALL SYNTAX

```
1  int sum_of_squares(int count) {
2    using namespace ranges;
3
4    return accumulate(
5      views::transform(
6        views::iota(1, count),
7        [](int x) { return x * x; }
8      ), 0
9    );
10 }
```

PIPED SYNTAX

```
1 int sum_of_squares(int count) {
2   using namespace ranges;
3
4   auto squares = views::iota(1, count)
5                | views::transform([](int x) { return x * x; });
6   return accumulate(squares, 0);
7 }
```

LISTING THE DAYS

```
1  using date = std::chrono::year_month_day;
2
3  auto first_day(std::chrono::year year) {
4    using namespace std::literals;
5    using std::chrono::January;
6    return std::chrono::sys_days{year / January / 1d};
7  }
8
9  auto dates(std::chrono::year start, std::chrono::year stop) {
10   return views::iota(first_day(start), first_day(stop)) |
11          views::transform([](const auto day) { return date{day}; });
12 }
```

LISTING THE DAYS

```
1  using date = std::chrono::year_month_day;
2
3  auto first_day(std::chrono::year year) {
4    using namespace std::literals;
5    using std::chrono::January;
6    return std::chrono::sys_days{year / January / 1d};
7  }
8
9  auto dates(std::chrono::year start, std::chrono::year stop) {
10   return views::iota(first_day(start), first_day(stop)) |
11          views::transform([](const auto day) { return date{day}; });
12 }
```

LISTING THE DAYS

```
using date = std::chrono::year_month_day;

auto first_day(std::chrono::year year) {
  using namespace std::literals;
  using std::chrono::January;
  return std::chrono::sys_days{year / January / 1d};
}

auto dates(std::chrono::year start, std::chrono::year stop) {
  return views::iota(first_day(start), first_day(stop)) |
         views::transform([](const auto day) { return date{day}; });
}
```

LISTING THE DAYS

```
1  using date = std::chrono::year_month_day;
2
3  auto first_day(std::chrono::year year) {
4    using namespace std::literals;
5    using std::chrono::January;
6    return std::chrono::sys_days{year / January / 1d};
7  }
8
9  auto dates(std::chrono::year start, std::chrono::year stop) {
10   return views::iota(first_day(start), first_day(stop)) |
11          views::transform([](const auto day) { return date{day}; });
12 }
```

LISTING THE DAYS

```
using date = std::chrono::year_month_day;

auto first_day(std::chrono::year year) {
  using namespace std::literals;
  using std::chrono::January;
  return std::chrono::sys_days{year / January / 1d};
}

auto dates_from(std::chrono::year year) {
  return views::iota(first_day(year)) |
         views::transform([](const auto day) { return date{day}; });
}
```

dates()

```
[2023-01-01, 2023-01-02, 2023-01-03, 2023-01-04, 2023-01-05, 2023-01-06, 2023-01-07, 2023-01-08, 2023-01-09, 2023-01-10, 2023-01-11, 2023-01-12,
2023-01-13, 2023-01-14, 2023-01-15, 2023-01-16, 2023-01-17, 2023-01-18, 2023-01-19, 2023-01-20, 2023-01-21, 2023-01-22, 2023-01-23, 2023-01-24, 2023-
01-25, 2023-01-26, 2023-01-27, 2023-01-28, 2023-01-29, 2023-01-30, 2023-01-31, 2023-02-01, 2023-02-02, 2023-02-03, 2023-02-04, 2023-02-05, 2023-02-
06, 2023-02-07, 2023-02-08, 2023-02-09, 2023-02-10, 2023-02-11, 2023-02-12, 2023-02-13, 2023-02-14, 2023-02-15, 2023-02-16, 2023-02-17, 2023-02-18,
2023-02-19, 2023-02-20, 2023-02-21, 2023-02-22, 2023-02-23, 2023-02-24, 2023-02-25, 2023-02-26, 2023-02-27, 2023-02-28, 2023-03-01, 2023-03-02, 2023-
03-03, 2023-03-04, 2023-03-05, 2023-03-06, 2023-03-07, 2023-03-08, 2023-03-09, 2023-03-10, 2023-03-11, 2023-03-12, 2023-03-13, 2023-03-14, 2023-03-
15, 2023-03-16, 2023-03-17, 2023-03-18, 2023-03-19, 2023-03-20, 2023-03-21, 2023-03-22, 2023-03-23, 2023-03-24, 2023-03-25, 2023-03-26, 2023-03-27,
2023-03-28, 2023-03-29, 2023-03-30, 2023-03-31, 2023-04-01, 2023-04-02, 2023-04-03, 2023-04-04, 2023-04-05, 2023-04-06, 2023-04-07, 2023-04-08, 2023-
04-09, 2023-04-10, 2023-04-11, 2023-04-12, 2023-04-13, 2023-04-14, 2023-04-15, 2023-04-16, 2023-04-17, 2023-04-18, 2023-04-19, 2023-04-20, 2023-04-
21, 2023-04-22, 2023-04-23, 2023-04-24, 2023-04-25, 2023-04-26, 2023-04-27, 2023-04-28, 2023-04-29, 2023-04-30, 2023-05-01, 2023-05-02, 2023-05-03,
2023-05-04, 2023-05-05, 2023-05-06, 2023-05-07, 2023-05-08, 2023-05-09, 2023-05-10, 2023-05-11, 2023-05-12, 2023-05-13, 2023-05-14, 2023-05-15, 2023-
05-16, 2023-05-17, 2023-05-18, 2023-05-19, 2023-05-20, 2023-05-21, 2023-05-22, 2023-05-23, 2023-05-24, 2023-05-25, 2023-05-26, 2023-05-27, 2023-05-
28, 2023-05-29, 2023-05-30, 2023-05-31, 2023-06-01, 2023-06-02, 2023-06-03, 2023-06-04, 2023-06-05, 2023-06-06, 2023-06-07, 2023-06-08, 2023-06-09,
2023-06-10, 2023-06-11, 2023-06-12, 2023-06-13, 2023-06-14, 2023-06-15, 2023-06-16, 2023-06-17, 2023-06-18, 2023-06-19, 2023-06-20, 2023-06-21, 2023-
06-22, 2023-06-23, 2023-06-24, 2023-06-25, 2023-06-26, 2023-06-27, 2023-06-28, 2023-06-29, 2023-06-30, 2023-07-01, 2023-07-02, 2023-07-03, 2023-07-
04, 2023-07-05, 2023-07-06, 2023-07-07, 2023-07-08, 2023-07-09, 2023-07-10, 2023-07-11, 2023-07-12, 2023-07-13, 2023-07-14, 2023-07-15, 2023-07-16,
2023-07-17, 2023-07-18, 2023-07-19, 2023-07-20, 2023-07-21, 2023-07-22, 2023-07-23, 2023-07-24, 2023-07-25, 2023-07-26, 2023-07-27, 2023-07-28, 2023-
07-29, 2023-07-30, 2023-07-31, 2023-08-01, 2023-08-02, 2023-08-03, 2023-08-04, 2023-08-05, 2023-08-06, 2023-08-07, 2023-08-08, 2023-08-09, 2023-08-
10, 2023-08-11, 2023-08-12, 2023-08-13, 2023-08-14, 2023-08-15, 2023-08-16, 2023-08-17, 2023-08-18, 2023-08-19, 2023-08-20, 2023-08-21, 2023-08-22,
```

FORMATTING RANGES

Source: pennaspillo.it

STANDARD

```
1 std::println("{}", std::vector{1, 2, 3});
2 std::println("{}", std::set{1, 2, 3});
3 std::println("{}", std::tuple{42, 16});
4 std::println("{}", std::map<int, int>{{1, 2}, {3, 4}});
```

STANDARD

```cpp
std::println("{}", std::vector{1, 2, 3});
std::println("{}", std::set{1, 2, 3});
std::println("{}", std::tuple{42, 16});
std::println("{}", std::map<int, int>{{1, 2}, {3, 4}});
```

```
[1, 2, 3]
{1, 2, 3}
(42, 16)
{1: 2, 3: 4}
```

STANDARD

```
1 std::println("{}", std::vector{1, 2, 3});
2 std::println("{}", std::set{1, 2, 3});
3 std::println("{}", std::tuple{42, 16});
4 std::println("{}", std::map<int, int>{{1, 2}, {3, 4}});
```

```
1 [1, 2, 3]
2 {1, 2, 3}
3 (42, 16)
4 {1: 2, 3: 4}
```

STANDARD

```
1 std::println("{}", std::vector{1, 2, 3});
2 std::println("{}", std::set{1, 2, 3});
3 std::println("{}", std::tuple{42, 16});
4 std::println("{}", std::map<int, int>{{1, 2}, {3, 4}});
```

```
1 [1, 2, 3]
2 {1, 2, 3}
3 (42, 16)
4 {1: 2, 3: 4}
```

STANDARD

```
1 std::println("{}", std::vector{1, 2, 3});
2 std::println("{}", std::set{1, 2, 3});
3 std::println("{}", std::tuple{42, 16});
4 std::println("{}", std::map<int, int>{{1, 2}, {3, 4}});
```

```
1 [1, 2, 3]
2 {1, 2, 3}
3 (42, 16)
4 {1: 2, 3: 4}
```

FORMAT SPECIFIERS

```
"{ arg-id (optional) }"
"{ arg-id (optional) : format-spec }"
```

NESTED
FORMAT SPECIFIERS

```
"{ arg-id (optional) }"
"{ arg-id (optional) : format-spec }"
"{ arg-id (optional) : format-spec : format-spec }"
"{ arg-id (optional) : format-spec : format-spec : format-spec }"
```

EXAMPLES

```
1 std::println("{}",          std::vector{1, 2, 3});
2 std::println("{:}",         std::vector{1, 2, 3});
3 std::println("{:#>20}",     std::vector{1, 2, 3});
4 std::println("{:#>20:#x}", std::vector{1, 2, 3});
```

```
1 [1, 2, 3]
2 [1, 2, 3]
3 ##########[1, 2, 3]
4 ####[0x1, 0x2, 0x3]
```

EXAMPLES

```
1 std::println("{}",          std::vector{1, 2, 3});
2 std::println("{:}",         std::vector{1, 2, 3});
3 std::println("{:#>20}",     std::vector{1, 2, 3});
4 std::println("{:#>20:#x}", std::vector{1, 2, 3});
```

```
1 [1, 2, 3]
2 [1, 2, 3]
3 ##########[1, 2, 3]
4 ####[0x1, 0x2, 0x3]
```

EXAMPLES

```
1 std::println("{}",          std::vector{1, 2, 3});
2 std::println("{:}",         std::vector{1, 2, 3});
3 std::println("{:#>20}",     std::vector{1, 2, 3});
4 std::println("{:#>20:#x}", std::vector{1, 2, 3});
```

```
1 [1, 2, 3]
2 [1, 2, 3]
3 ##########[1, 2, 3]
4 ####[0x1, 0x2, 0x3]
```

EXAMPLES

```
1 std::println("{}",          std::vector{1, 2, 3});
2 std::println("{:}",         std::vector{1, 2, 3});
3 std::println("{:#>20}",     std::vector{1, 2, 3});
4 std::println("{:#>20:#x}", std::vector{1, 2, 3});
```

```
1 [1, 2, 3]
2 [1, 2, 3]
3 ##########[1, 2, 3]
4 ####[0x1, 0x2, 0x3]
```

CUSTOMIZATION

```
1 std::println("{:?}",  "A\tCCU"s);
2 std::println("{:m}",  std::pair{"C++", 23});
3 std::println("{:n}",  std::set{1, 2, 3});
```

```
1 "A\tCCU"
2 "C++": 23
3 1, 2, 3
```

CUSTOMIZATION

```
1 std::println("{:?}",  "A\tCCU"s);
2 std::println("{:m}",  std::pair{"C++", 23});
3 std::println("{:n}",  std::set{1, 2, 3});
```

```
1 "A\tCCU"
2 "C++": 23
3 1, 2, 3
```

CUSTOMIZATION

```
1 std::println("{:?}",  "A\tCCU"s);
2 std::println("{:m}",  std::pair{"C++", 23});
3 std::println("{:n}",  std::set{1, 2, 3});
```

```
1 "A\tCCU"
2 "C++": 23
3 1, 2, 3
```

WINDOWING VIEWS

Source: Mark Waugh

views::chunk

```
std::vector v = {1, 2, 3, 4, 5};
std::println("{}", v | std::views::chunk(2));
```

views::chunk

```
std::vector v = {1, 2, 3, 4, 5};
std::println("{}", v | std::views::chunk(2));
```

views::chunk

```
std::vector v = {1, 2, 3, 4, 5};
std::println("{}", v | std::views::chunk(2));
```

```
[[1, 2], [3, 4], [5]]
```

views::slide

```
std::vector v = {1, 2, 3, 4, 5};
std::println("{}", v | std::views::slide(2));
```

views::slide

```
std::vector v = {1, 2, 3, 4, 5};
std::println("{}", v | std::views::slide(2));
```

views::slide

```
std::vector v = {1, 2, 3, 4, 5};
std::println("{}", v | std::views::slide(2));
```

views::slide

```
std::vector v = {1, 2, 3, 4, 5};
std::println("{}", v | std::views::slide(2));
```

```
[[1, 2], [2, 3], [3, 4], [4, 5]]
```

views::stride

```
std::vector v = {1, 2, 3, 4, 5};
std::println("{}", v | std::views::stride(2));
```

views::stride

```
std::vector v = {1, 2, 3, 4, 5};
std::println("{}", v | std::views::stride(2));
```

views::stride

```
std::vector v = {1, 2, 3, 4, 5};
std::println("{}", v | std::views::stride(2));
```

```
[1, 3, 5]
```

GENERIC?

Algorithm	Step	Size	Partial
generic	n	k	b
chunk	k	k	true
slide	1	k	false
stride	k	1	N/A

WINDOWED

```
namespace args {
struct windowed {
    std::size_t size;
    std::size_t stride;
};
}  // namespace args
constexpr auto windowed(args::windowed args) {
    using namespace std::views;
    return slide(args.size) | stride(args.stride);
}

std::println("{}",
             std::views::iota(0, 20)
             | windowed({.size = 5, .stride = 3}));
```

```
[[0, 1, 2, 3, 4], [3, 4, 5, 6, 7], [6, 7, 8, 9, 10], ...
```

views::chunk_by

```
using namespace std::ranges;
std::vector v = {1, 2, 2, 3, 1, 2, 0, 4, 5, 2};
std::println("{}", v | views::chunk_by(less_equal{}));
```

views::chunk_by

```
using namespace std::ranges;
std::vector v = {1, 2, 2, 3, 1, 2, 0, 4, 5, 2};
std::println("{}", v | views::chunk_by(less_equal{}));
```

```
[[1, 2, 2, 3], [1, 2], [0, 4, 5], [2]]
```

UNLIKE range-v3'S group_by

```
using namespace ranges;
std::vector v = {1, 2, 2, 3, 1, 2, 0, 4, 5, 2};
std::println("{}", v | views::group_by(less_equal{}));
```

```
[[1, 2, 2, 3, 1, 2], [0, 4, 5, 2]]
```

SPLIT MONTHS

```
auto by_month() {
  return views::chunk_by([](date a, date b) {
    return a.month() == b.month();
  });
}
```

dates()

```
[2023-01-01, 2023-01-02, 2023-01-03, 2023-01-04, 2023-01-05, 2023-01-06, 2023-01-07, 2023-01-08, 2023-01-09, 2023-01-10, 2023-01-11, 2023-01-12,
2023-01-13, 2023-01-14, 2023-01-15, 2023-01-16, 2023-01-17, 2023-01-18, 2023-01-19, 2023-01-20, 2023-01-21, 2023-01-22, 2023-01-23, 2023-01-24, 2023-
01-25, 2023-01-26, 2023-01-27, 2023-01-28, 2023-01-29, 2023-01-30, 2023-01-31, 2023-02-01, 2023-02-02, 2023-02-03, 2023-02-04, 2023-02-05, 2023-02-
06, 2023-02-07, 2023-02-08, 2023-02-09, 2023-02-10, 2023-02-11, 2023-02-12, 2023-02-13, 2023-02-14, 2023-02-15, 2023-02-16, 2023-02-17, 2023-02-18,
2023-02-19, 2023-02-20, 2023-02-21, 2023-02-22, 2023-02-23, 2023-02-24, 2023-02-25, 2023-02-26, 2023-02-27, 2023-02-28, 2023-03-01, 2023-03-02, 2023-
03-03, 2023-03-04, 2023-03-05, 2023-03-06, 2023-03-07, 2023-03-08, 2023-03-09, 2023-03-10, 2023-03-11, 2023-03-12, 2023-03-13, 2023-03-14, 2023-03-
15, 2023-03-16, 2023-03-17, 2023-03-18, 2023-03-19, 2023-03-20, 2023-03-21, 2023-03-22, 2023-03-23, 2023-03-24, 2023-03-25, 2023-03-26, 2023-03-27,
2023-03-28, 2023-03-29, 2023-03-30, 2023-03-31, 2023-04-01, 2023-04-02, 2023-04-03, 2023-04-04, 2023-04-05, 2023-04-06, 2023-04-07, 2023-04-08, 2023-
04-09, 2023-04-10, 2023-04-11, 2023-04-12, 2023-04-13, 2023-04-14, 2023-04-15, 2023-04-16, 2023-04-17, 2023-04-18, 2023-04-19, 2023-04-20, 2023-04-
21, 2023-04-22, 2023-04-23, 2023-04-24, 2023-04-25, 2023-04-26, 2023-04-27, 2023-04-28, 2023-04-29, 2023-04-30, 2023-05-01, 2023-05-02, 2023-05-03,
2023-05-04, 2023-05-05, 2023-05-06, 2023-05-07, 2023-05-08, 2023-05-09, 2023-05-10, 2023-05-11, 2023-05-12, 2023-05-13, 2023-05-14, 2023-05-15, 2023-
05-16, 2023-05-17, 2023-05-18, 2023-05-19, 2023-05-20, 2023-05-21, 2023-05-22, 2023-05-23, 2023-05-24, 2023-05-25, 2023-05-26, 2023-05-27, 2023-05-
28, 2023-05-29, 2023-05-30, 2023-05-31, 2023-06-01, 2023-06-02, 2023-06-03, 2023-06-04, 2023-06-05, 2023-06-06, 2023-06-07, 2023-06-08, 2023-06-09,
2023-06-10, 2023-06-11, 2023-06-12, 2023-06-13, 2023-06-14, 2023-06-15, 2023-06-16, 2023-06-17, 2023-06-18, 2023-06-19, 2023-06-20, 2023-06-21, 2023-
06-22, 2023-06-23, 2023-06-24, 2023-06-25, 2023-06-26, 2023-06-27, 2023-06-28, 2023-06-29, 2023-06-30, 2023-07-01, 2023-07-02, 2023-07-03, 2023-07-
04, 2023-07-05, 2023-07-06, 2023-07-07, 2023-07-08, 2023-07-09, 2023-07-10, 2023-07-11, 2023-07-12, 2023-07-13, 2023-07-14, 2023-07-15, 2023-07-16,
2023-07-17, 2023-07-18, 2023-07-19, 2023-07-20, 2023-07-21, 2023-07-22, 2023-07-23, 2023-07-24, 2023-07-25, 2023-07-26, 2023-07-27, 2023-07-28, 2023-
07-29, 2023-07-30, 2023-07-31, 2023-08-01, 2023-08-02, 2023-08-03, 2023-08-04, 2023-08-05, 2023-08-06, 2023-08-07, 2023-08-08, 2023-08-09, 2023-08-
10, 2023-08-11, 2023-08-12, 2023-08-13, 2023-08-14, 2023-08-15, 2023-08-16, 2023-08-17, 2023-08-18, 2023-08-19, 2023-08-20, 2023-08-21, 2023-08-22,
```

| by_month

```
[
  [2023-01-01, 2023-01-02, 2023-01-03, 2023-01-04, 2023-01-05, 2023-01-06, 2023-01-07, 2023-01-08, 2023-01-09, 2023-01-10, 2023-01-11, 2023-01-12,
2023-01-13, 2023-01-14, 2023-01-15, 2023-01-16, 2023-01-17, 2023-01-18, 2023-01-19, 2023-01-20, 2023-01-21, 2023-01-22, 2023-01-23, 2023-01-24, 2023-
01-25, 2023-01-26, 2023-01-27, 2023-01-28, 2023-01-29, 2023-01-30, 2023-01-31],
  [2023-02-01, 2023-02-02, 2023-02-03, 2023-02-04, 2023-02-05, 2023-02-06, 2023-02-07, 2023-02-08, 2023-02-09, 2023-02-10, 2023-02-11, 2023-02-12,
2023-02-13, 2023-02-14, 2023-02-15, 2023-02-16, 2023-02-17, 2023-02-18, 2023-02-19, 2023-02-20, 2023-02-21, 2023-02-22, 2023-02-23, 2023-02-24, 2023-
02-25, 2023-02-26, 2023-02-27, 2023-02-28],
  [2023-03-01, 2023-03-02, 2023-03-03, 2023-03-04, 2023-03-05, 2023-03-06, 2023-03-07, 2023-03-08, 2023-03-09, 2023-03-10, 2023-03-11, 2023-03-12,
2023-03-13, 2023-03-14, 2023-03-15, 2023-03-16, 2023-03-17, 2023-03-18, 2023-03-19, 2023-03-20, 2023-03-21, 2023-03-22, 2023-03-23, 2023-03-24, 2023-
03-25, 2023-03-26, 2023-03-27, 2023-03-28, 2023-03-29, 2023-03-30, 2023-03-31],
  [2023-04-01, 2023-04-02, 2023-04-03, 2023-04-04, 2023-04-05, 2023-04-06, 2023-04-07, 2023-04-08, 2023-04-09, 2023-04-10, 2023-04-11, 2023-04-12,
2023-04-13, 2023-04-14, 2023-04-15, 2023-04-16, 2023-04-17, 2023-04-18, 2023-04-19, 2023-04-20, 2023-04-21, 2023-04-22, 2023-04-23, 2023-04-24, 2023-
04-25, 2023-04-26, 2023-04-27, 2023-04-28, 2023-04-29, 2023-04-30],
  [2023-05-01, 2023-05-02, 2023-05-03, 2023-05-04, 2023-05-05, 2023-05-06, 2023-05-07, 2023-05-08, 2023-05-09, 2023-05-10, 2023-05-11, 2023-05-12,
2023-05-13, 2023-05-14, 2023-05-15, 2023-05-16, 2023-05-17, 2023-05-18, 2023-05-19, 2023-05-20, 2023-05-21, 2023-05-22, 2023-05-23, 2023-05-24, 2023-
05-25, 2023-05-26, 2023-05-27, 2023-05-28, 2023-05-29, 2023-05-30, 2023-05-31],
  [2023-06-01, 2023-06-02, 2023-06-03, 2023-06-04, 2023-06-05, 2023-06-06, 2023-06-07, 2023-06-08, 2023-06-09, 2023-06-10, 2023-06-11, 2023-06-12,
2023-06-13, 2023-06-14, 2023-06-15, 2023-06-16, 2023-06-17, 2023-06-18, 2023-06-19, 2023-06-20, 2023-06-21, 2023-06-22, 2023-06-23, 2023-06-24, 2023-
06-25, 2023-06-26, 2023-06-27, 2023-06-28, 2023-06-29, 2023-06-30]
```

GENERATOR

Source: Albanpix/Rex Features

std::generator

```cpp
#include <generator>

std::generator<int> fib() {
    auto a = 0, b = 1;
    while (true) {
        co_yield std::exchange(a, std::exchange(b, a + b));
    }
}

auto rng = fib() | std::views::drop(6) | std::views::take(3);
return std::ranges::fold_left(std::move(rng), 0, std::plus{});
```

std::generator

```cpp
#include <generator>

std::generator<int> fib() {
    auto a = 0, b = 1;
    while (true) {
        co_yield std::exchange(a, std::exchange(b, a + b));
    }
}

auto rng = fib() | std::views::drop(6) | std::views::take(3);
return std::ranges::fold_left(std::move(rng), 0, std::plus{});
```

std::generator

```
template<
    class Ref,
    class V = void,
    class Allocator = void
>
class generator;

namespace pmr {
template< class Ref, class V = void >
using generator =
    generator<Ref, V, std::pmr::polymorphic_allocator<>>;
}
```

std::generator

```
template<
    class Ref,
    class V = void,
    class Allocator = void
>
class generator;

namespace pmr {
template< class Ref, class V = void >
using generator =
    generator<Ref, V, std::pmr::polymorphic_allocator<>>;
}
```

std::generator

```
template<
    class Ref,
    class V = void,
    class Allocator = void
>
class generator {

using Value = conditional_t<is_void_v<V>, remove_cvref_t<Ref>, V>;
using Reference = conditional_t<is_void_v<V>, Ref&&, Ref>;
using Yielded = conditional_t<is_reference_v<Reference>, Reference, const Reference&>;

};
```

See P2529 for details.

YIELDING RANGES

```
std::generator<std::string_view> elements() {
  std::vector<std::string> v = {"Hello", "Elements", "Of"};
  co_yield std::ranges::elements_of(v);
}
```

RECURSIVE GENERATOR

```
struct Tree {
    Tree* left;
    Tree* right;
    int value;
};

std::generator<int> visit(const Tree& tree) {
    if (tree.left)
      co_yield std::ranges::elements_of(visit(*tree.left));
    co_yield tree.value;
    if (tree.right)
      co_yield std::ranges::elements_of(visit(*tree.right));
}
```

IMPLEMENTING NEW ADAPTORS

```
template <std::ranges::range Head, std::ranges::range... Tail>
inline std::generator<
    std::common_type_t<std::ranges::range_reference_t<Head>,
                       std::ranges::range_reference_t<Tail>...>,
    std::common_type_t<std::ranges::range_value_t<Head>,
                       std::ranges::range_value_t<Tail>...>>
concat(Head head, Tail... tail) {
    co_yield std::ranges::elements_of(std::move(head));
    if constexpr (sizeof...(tail) > 0) {
        co_yield std::ranges::elements_of(concat(std::move(tail)...));
    }
};

std::vector<int> v1{1, 2, 3}, v2{4, 5}, v3{};
std::array<const int,3> a{6, 7, 8};
auto s = std::views::single(9);
std::println("{}", concat(v1, v2, v3, a, s));
```

IMPLEMENTING NEW ADAPTORS

```
template <std::ranges::range Head, std::ranges::range... Tail>
inline std::generator<
    std::common_type_t<std::ranges::range_reference_t<Head>,
                       std::ranges::range_reference_t<Tail>...>,
    std::common_type_t<std::ranges::range_value_t<Head>,
                       std::ranges::range_value_t<Tail>...>>
concat(Head head, Tail... tail) {
    co_yield std::ranges::elements_of(std::move(head));
    if constexpr (sizeof...(tail) > 0) {
        co_yield std::ranges::elements_of(concat(std::move(tail)...));
    }
};

std::vector<int> v1{1, 2, 3}, v2{4, 5}, v3{};
std::array<const int,3> a{6, 7, 8};
auto s = std::views::single(9);
std::println("{}", concat(v1, v2, v3, a, s));
```

CHANGING POINT OF VIEW

Source: Mark Sutcliffe

IS THIS A VIEW

```
template <std::ranges::range Head, std::ranges::range... Tail>
inline std::generator<
    std::common_type_t<std::ranges::range_reference_t<Head>,
                       std::ranges::range_reference_t<Tail>...>,
    std::common_type_t<std::ranges::range_value_t<Head>,
                       std::ranges::range_value_t<Tail>...>>
concat(Head head, Tail... tail) {
    co_yield std::ranges::elements_of(std::move(head));
    if constexpr (sizeof...(tail) > 0) {
        co_yield std::ranges::elements_of(concat(std::move(tail)...));
    }
};
```

WHAT IS A VIEW?

```
concept Range<typename T> =
    Iterable<T> &&
    Semiregular<T> &&
    is_range<T>::value;

// Copying and assignment execute in constant time
// is_range detects shallow constness
```

N4128: Ranges for the Standard Library

WHAT IS A VIEW?

```
template<class T>
concept View =
    Range<T> &&
    Semiregular<T> &&
    enable_view<T>;

// has constant time copy, move and assignment operators
// enable_view detects shallow constness
```

P0896: The One Ranges Proposal

WHAT IS A VIEW?

```
template<class T>
concept view =
    range<T> &&
    movable<T> &&
    default_initializable<T> &&
    enable_view<T>;

// move assignment and destruction operators
// has constant time move construction,
// enable_view detects shallow constness
```

P1456: Move-only views

WHAT IS A VIEW?

```
template<class T>
concept view =
    range<T> &&
    movable<T> &&
    default_initializable<T> &&
    enable_view<T>;

// has constant time move construction,
// move assignment and destruction operators
```

LWG3326: `enable_view` has false positives

WHAT IS A VIEW?

```
template<class T>
concept view =
    range<T> &&
    movable<T> &&
    enable_view<T>;

// has constant time move construction,
// move assignment and destruction operators
```

P2325: Views should not be required to be default constructible

VIEWING R-VALUE CONTAINERS

```
std::vector<int> get_ints();

auto rng = get_ints()
         | views::filter([](int i){ return i > 0; })
         | views::transform([](int i){ return i * i; });
```

VIEWING R-VALUE CONTAINERS

```
std::vector<int> get_ints();

auto ints = get_ints();
auto rng = ints
         | views::filter([](int i){ return i > 0; })
         | views::transform([](int i){ return i * i; });
```

owning_view

```
template<std::ranges::range T>
struct owning_view : ranges::view_interface<owning_view<T>> {
    T t;

    owning_view(T t) : t(std::move(t)) { }

    owning_view(owning_view const&) = delete;
    owning_view(owning_view&&) = default;
    owning_view& operator=(owning_view const&) = delete;
    owning_view& operator=(owning_view&&) = default;

    auto begin() { return begin(t); }
    auto end()   { return end(t); }
};
```

VIEWING R-VALUE CONTAINERS

```
std::vector<int> get_ints();

auto rng = owning_view{get_ints()}
         | views::filter([](int i){ return i > 0; })
         | views::transform([](int i){ return i * i; });
```

`views::all`

Given a subexpression `E`, the expression `views::all(E)` is expression-equivalent to:

- *`decay-copy(E)` if the decayed type of `E` models view.*
- *Otherwise, `ref_view{E}` if that expression is well-formed.*
- *Otherwise, `owning_view{E}`.*

WHAT IS A VIEW?

```
template<class T>
concept view =
    range<T> &&
    movable<T> &&
    enable_view<T>;

// 1. has O(1) move construction
// 2. move assignment is no more complex than
//    destruction followed by move construction
// 3. N copies of M elements have O(N+M) destruction
//    [implies that a moved-from object has O(1) destruction]
// 4. if copy construcible, has O(1) copy construction
// 5. if copyable, copy assignment is no more complex than
//    destruction followed by copy construction.
```

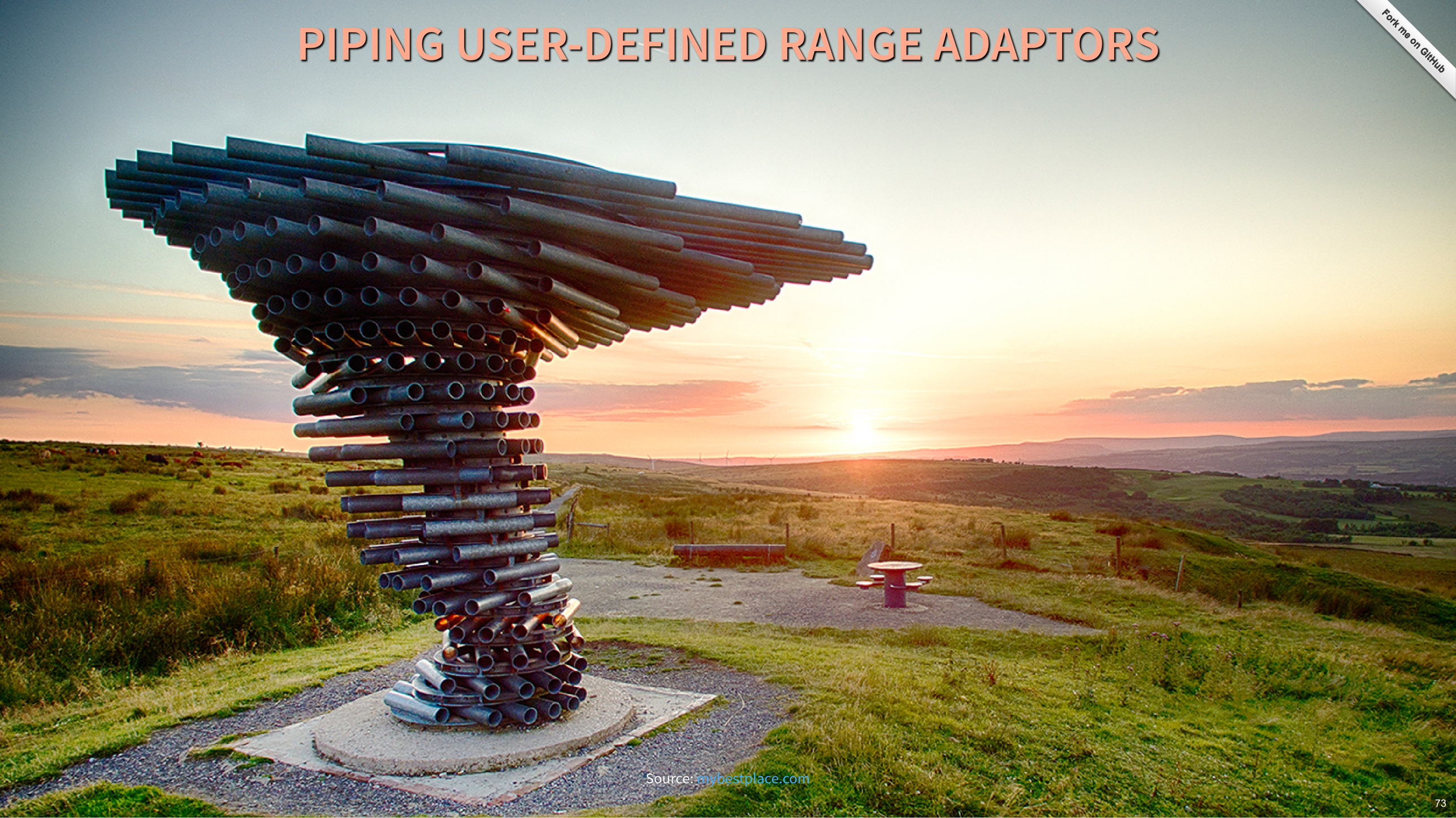
P2415: What is a `view`?

VIEWING R-VALUE CONTAINERS

```
std::vector<int> get_ints();

auto rng = get_ints()
         | views::filter([](int i){ return i > 0; })
         | views::transform([](int i){ return i * i; });
```

PIPING USER-DEFINED RANGE ADAPTORS



Source: mybestplace.com

RANGE ADAPTOR OBJECT

$$adaptor(range, args...) \equiv$$
$$adaptor(args...)(range) \equiv$$
$$range \mid adaptor(args...)$$

e.g. `views::transform`

RANGE ADAPTOR CLOSURE OBJECT

$$C(R) \equiv R \mid C$$

e.g. `views::reverse`, `views::transform(f)`

RANGE ADAPTOR CLOSURE OBJECT

$$R \mid C \mid D \equiv R \mid (C \mid D)$$

```
auto reverse_trasform(auto f) {
  return views::reverse | views::transform(f);
}
```

MORE COMPLEX EXAMPLE

```
auto reverse_tail = [](ranges::forward_range auto&& R) {
  return concat(R | views::take(1),
                R | views::drop(1) | views::reverse);
};

std::println("{}", std::vector{1, 2, 3, 4} | reverse_tail);
```

MORE COMPLEX EXAMPLE

```
auto reverse_tail = [](ranges::forward_range auto&& R) {
  return concat(R | views::take(1),
                R | views::drop(1) | views::reverse);
};

std::println("{}", std::vector{1, 2, 3, 4} | reverse_tail);
```

ranges::range_adaptor_closure

```
1 struct reverse_tail_fn
2   : std::ranges::range_adaptor_closure<reverse_tail_fn> {
3   auto operator()(ranges::forward_range auto&& R) {
4     return concat(R | views::take(1),
5                   R | views::drop(1) | views::reverse);
6   };
7 }  reverse_tail;
8
9 std::println("{}", std::vector{1, 2, 3, 4} | reverse_tail);
```

ranges::range_adaptor_closure

```
1 struct reverse_tail_fn
2   : std::ranges::range_adaptor_closure<reverse_tail_fn> {
3   auto operator()(ranges::forward_range auto&& R) {
4     return concat(R | views::take(1),
5                   R | views::drop(1) | views::reverse);
6   };
7 }  reverse_tail;
8
9 std::println("{}", std::vector{1, 2, 3, 4} | reverse_tail);
```

BACK TO LAMBDA

```
template <typename F>
class closure
  : public std::ranges::range_adaptor_closure<closure<F>> {
    F f;
public:
    constexpr closure(F f) : f(f) { }

    template <std::ranges::viewable_range R>
        requires std::invocable<F const&, R>
    constexpr auto operator()(R&& r) const {
        return f(std::forward<R>(r));
    }
};

closure reverse_tail = [](ranges::forward_range auto&& R) {
```

BACK TO LAMBDA

```
public:
    constexpr closure(F f) : f(f) { }

    template <std::ranges::viewable_range R>
        requires std::invocable<F const&, R>
    constexpr auto operator()(R&& r) const {
        return f(std::forward<R>(r));
    }
};

closure reverse_tail = [](ranges::forward_range auto&& R) {
  return concat(R | views::take(1),
                R | views::drop(1) | views::reverse);
};

std::println("{}", std::vector{1, 2, 3, 4} | reverse_tail);
```

CASE FOR ADAPTOR

```cpp
template <typename F>
class adaptor {
    F f;
public:
    constexpr adaptor(F f) : f(f) { }

    template <typename... Args>
    constexpr auto operator()(Args&&... args) const {
        if constexpr (std::invocable<F const&, Args...>) {
            return f(std::forward<Args>(args)...);
        } else {
            return closure(std::bind_back(f,
              std::forward<Args>(args)...));
        }
    }
```

CASE FOR ADAPTOR

```cpp
template <typename F>
class adaptor {
    F f;
public:
    constexpr adaptor(F f) : f(f) { }

    template <typename... Args>
    constexpr auto operator()(Args&&... args) const {
        if constexpr (std::invocable<F const&, Args...>) {
            return f(std::forward<Args>(args)...);
        } else {
            return closure(std::bind_back(f,
              std::forward<Args>(args)...));
        }
    }
```

FORMATTING MONTHS

```
auto by_week() {
  return views::chunk_by([](date a, date b) {
    return detail::start_of_week(a) == detail::start_of_week(b);
  });
}

// In:  range<range<date>>: month grouped by weeks.
// Out: range<std::string>: month with formatted weeks.
inline constexpr closure format_weeks =
    [](nested_range_of<date, 2> auto month) {
      const auto format_day = [](date d) {
        return std::format("{:3}", d.day());
      };
      const auto format_week = views::transform(format_day) | views::join;
      return concat(
          month | views::take(1) |
              views::transform([&](range_of<date> auto week) {
                return detail::format_as_string("{:>21} ", week | format_week);
              }),
          month | views::drop(1) |
              views::transform([&](range_of<date> auto week) {
```

FORMATTING MONTHS

```
            }),
          month | views::drop(1) |
              views::transform([&](range_of<date> auto week) {
                return detail::format_as_string("{:22}", week | format_week);
              }));
    };

// Return a formatted string with the title of the month
// corresponding to a date.
std::string month_title(date d) { return std::format("{:^22%B}", d.month()); }

auto layout_months() {
  return views::transform([](range_of<date> auto month) {
    const auto week_count = ranges::distance(month | by_week());
    static const std::string empty_week(22, ' ');
    return concat(
        views::single(month_title(month.front())),
        month | by_week() | format_weeks,
        views::repeat(empty_week, 6 - week_count));
  });
}
```

FORMATTING MONTHS

```
                views::transform([&](range_of<date> auto week) {
                  return detail::format_as_string("{:>21} ", week | format_week);
                }),
            month | views::drop(1) |
                views::transform([&](range_of<date> auto week) {
                  return detail::format_as_string("{:22}", week | format_week);
                }));
    };

// Return a formatted string with the title of the month
// corresponding to a date.
std::string month_title(date d) { return std::format("{:^22%B}", d.month()); }

auto layout_months() {
  return views::transform([](range_of<date> auto month) {
    const auto week_count = ranges::distance(month | by_week());
    static const std::string empty_week(22, ' ');
    return concat(
        views::single(month_title(month.front())),
        month | by_week() | format_weeks,
        views::repeat(empty_week, 6 - week_count));
  });
```

FORMATTING MONTHS

```
auto by_week() {
  return views::chunk_by([](date a, date b) {
    return detail::start_of_week(a) == detail::start_of_week(b);
  });
}

// In:  range<range<date>>: month grouped by weeks.
// Out: range<std::string>: month with formatted weeks.
inline constexpr closure format_weeks =
    [](nested_range_of<date, 2> auto month) {
      const auto format_day = [](date d) {
        return std::format("{:3}", d.day());
      };
      const auto format_week = views::transform(format_day) | views::join;
      return concat(
          month | views::take(1) |
              views::transform([&](range_of<date> auto week) {
                return detail::format_as_string("{:>21} ", week | format_week);
              }),
          month | views::drop(1) |
              views::transform([&](range_of<date> auto week) {
```

FORMATTING MONTHS

```
 6
 7 // In:  range<range<date>>: month grouped by weeks.
 8 // Out: range<std::string>: month with formatted weeks.
 9 inline constexpr closure format_weeks =
10     [](nested_range_of<date, 2> auto month) {
11       const auto format_day = [](date d) {
12         return std::format("{:3}", d.day());
13       };
14       const auto format_week = views::transform(format_day) | views::join;
15       return concat(
16           month | views::take(1) |
17               views::transform([&](range_of<date> auto week) {
18                 return detail::format_as_string("{:>21} ", week | format_week);
19               }),
20           month | views::drop(1) |
21               views::transform([&](range_of<date> auto week) {
22                 return detail::format_as_string("{:22}", week | format_week);
23               }));
24     };
25
26 // Return a formatted string with the title of the month
27 // corresponding to a date.
```

| by_month

```
[
  [2023-01-01, 2023-01-02, 2023-01-03, 2023-01-04, 2023-01-05, 2023-01-06, 2023-01-07, 2023-01-08, 2023-01-09, 2023-01-10, 2023-01-11, 2023-01-12,
2023-01-13, 2023-01-14, 2023-01-15, 2023-01-16, 2023-01-17, 2023-01-18, 2023-01-19, 2023-01-20, 2023-01-21, 2023-01-22, 2023-01-23, 2023-01-24, 2023-
01-25, 2023-01-26, 2023-01-27, 2023-01-28, 2023-01-29, 2023-01-30, 2023-01-31],
  [2023-02-01, 2023-02-02, 2023-02-03, 2023-02-04, 2023-02-05, 2023-02-06, 2023-02-07, 2023-02-08, 2023-02-09, 2023-02-10, 2023-02-11, 2023-02-12,
2023-02-13, 2023-02-14, 2023-02-15, 2023-02-16, 2023-02-17, 2023-02-18, 2023-02-19, 2023-02-20, 2023-02-21, 2023-02-22, 2023-02-23, 2023-02-24, 2023-
02-25, 2023-02-26, 2023-02-27, 2023-02-28],
  [2023-03-01, 2023-03-02, 2023-03-03, 2023-03-04, 2023-03-05, 2023-03-06, 2023-03-07, 2023-03-08, 2023-03-09, 2023-03-10, 2023-03-11, 2023-03-12,
2023-03-13, 2023-03-14, 2023-03-15, 2023-03-16, 2023-03-17, 2023-03-18, 2023-03-19, 2023-03-20, 2023-03-21, 2023-03-22, 2023-03-23, 2023-03-24, 2023-
03-25, 2023-03-26, 2023-03-27, 2023-03-28, 2023-03-29, 2023-03-30, 2023-03-31],
  [2023-04-01, 2023-04-02, 2023-04-03, 2023-04-04, 2023-04-05, 2023-04-06, 2023-04-07, 2023-04-08, 2023-04-09, 2023-04-10, 2023-04-11, 2023-04-12,
2023-04-13, 2023-04-14, 2023-04-15, 2023-04-16, 2023-04-17, 2023-04-18, 2023-04-19, 2023-04-20, 2023-04-21, 2023-04-22, 2023-04-23, 2023-04-24, 2023-
04-25, 2023-04-26, 2023-04-27, 2023-04-28, 2023-04-29, 2023-04-30],
  [2023-05-01, 2023-05-02, 2023-05-03, 2023-05-04, 2023-05-05, 2023-05-06, 2023-05-07, 2023-05-08, 2023-05-09, 2023-05-10, 2023-05-11, 2023-05-12,
2023-05-13, 2023-05-14, 2023-05-15, 2023-05-16, 2023-05-17, 2023-05-18, 2023-05-19, 2023-05-20, 2023-05-21, 2023-05-22, 2023-05-23, 2023-05-24, 2023-
05-25, 2023-05-26, 2023-05-27, 2023-05-28, 2023-05-29, 2023-05-30, 2023-05-31],
  [2023-06-01, 2023-06-02, 2023-06-03, 2023-06-04, 2023-06-05, 2023-06-06, 2023-06-07, 2023-06-08, 2023-06-09, 2023-06-10, 2023-06-11, 2023-06-12,
2023-06-13, 2023-06-14, 2023-06-15, 2023-06-16, 2023-06-17, 2023-06-18, 2023-06-19, 2023-06-20, 2023-06-21, 2023-06-22, 2023-06-23, 2023-06-24, 2023-
06-25, 2023-06-26, 2023-06-27, 2023-06-28, 2023-06-29, 2023-06-30]
```

| layout_months

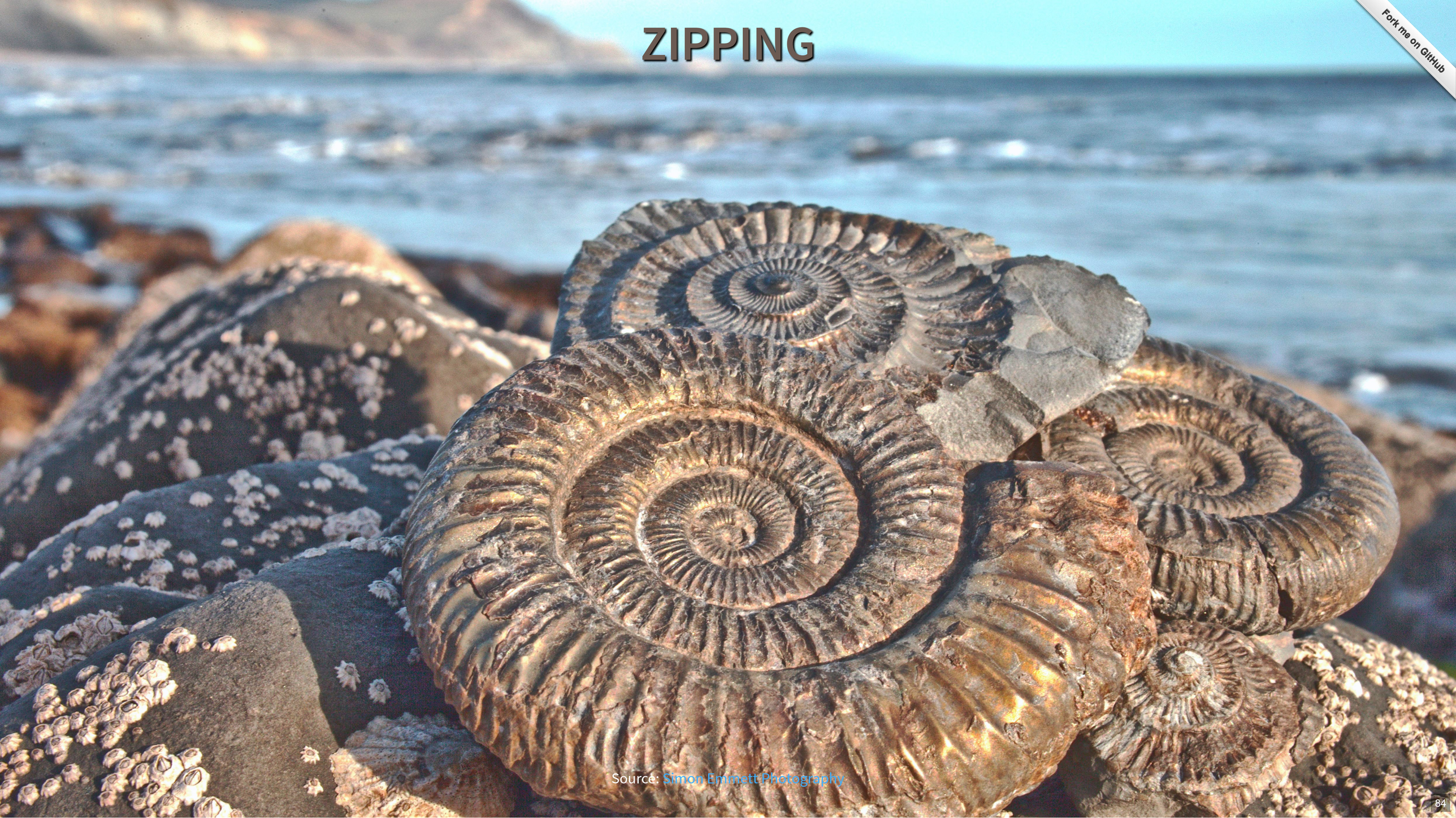
```
[
  ["       January        ", "01 02 03 04 05 06 07  ", "08 09 10 11 12 13 14  ", "15 16 17 18 19 20 21  ", "22 23 24 25 26 27 28  ", "29 30 31
", "                      "],
  ["       February       ", "         01 02 03 04  ", "05 06 07 08 09 10 11  ", "12 13 14 15 16 17 18  ", "19 20 21 22 23 24 25  ", "26 27 28
", "                      "],
  ["        March         ", "         01 02 03 04  ", "05 06 07 08 09 10 11  ", "12 13 14 15 16 17 18  ", "19 20 21 22 23 24 25  ", "26 27 28 29 30
31     ", "                      "],
  ["        April         ", "                  01  ", "02 03 04 05 06 07 08  ", "09 10 11 12 13 14 15  ", "16 17 18 19 20 21 22  ", "23 24 25 26 27
28 29  ", "30                    "],
  ["         May          ", "   01 02 03 04 05 06  ", "07 08 09 10 11 12 13  ", "14 15 16 17 18 19 20  ", "21 22 23 24 25 26 27  ", "28 29 30 31
", "                      "],
  ["         June         ", "            01 02 03  ", "04 05 06 07 08 09 10  ", "11 12 13 14 15 16 17  ", "18 19 20 21 22 23 24  ", "25 26 27 28 29
30     ", "                      "],
  ["         July         ", "                  01  ", "02 03 04 05 06 07 08  ", "09 10 11 12 13 14 15  ", "16 17 18 19 20 21 22  ", "23 24 25 26 27
28 29  ", "30 31                 "],
  ["        August        ", "      01 02 03 04 05  ", "06 07 08 09 10 11 12  ", "13 14 15 16 17 18 19  ", "20 21 22 23 24 25 26  ", "27 28 29 30 31
", "                      "],
  ["      September       ", "               01 02  ", "03 04 05 06 07 08 09  ", "10 11 12 13 14 15 16  ", "17 18 19 20 21 22 23  ", "24 25 26 27 28
29 30  ", "                      "]
```

to a list of formatted months.

chunk(3)

```
[
  [
    ["       January        ", " 01 02 03 04 05 06 07 ", " 08 09 10 11 12 13 14 ", " 15 16 17 18 19 20 21 ", " 22 23 24 25 26 27 28 ", " 29 30 31
", "                      "],
    ["       February       ", "          01 02 03 04 ", " 05 06 07 08 09 10 11 ", " 12 13 14 15 16 17 18 ", " 19 20 21 22 23 24 25 ", " 26 27 28
", "                      "],
    ["        March         ", "          01 02 03 04 ", " 05 06 07 08 09 10 11 ", " 12 13 14 15 16 17 18 ", " 19 20 21 22 23 24 25 ", " 26 27 28 29
30 31    ", "                      "]
  ],
  [
    ["        April         ", "                   01 ", " 02 03 04 05 06 07 08 ", " 09 10 11 12 13 14 15 ", " 16 17 18 19 20 21 22 ", " 23 24 25 26
27 28 29 ", " 30                 "],
    ["         May          ", "    01 02 03 04 05 06 ", " 07 08 09 10 11 12 13 ", " 14 15 16 17 18 19 20 ", " 21 22 23 24 25 26 27 ", " 28 29 30 31
", "                      "],
    ["         June         ", "             01 02 03 ", " 04 05 06 07 08 09 10 ", " 11 12 13 14 15 16 17 ", " 18 19 20 21 22 23 24 ", " 25 26 27 28
29 30    ", "                      "]
  ],
  [
    ["         July         ", "                   01 ", " 02 03 04 05 06 07 08 ", " 09 10 11 12 13 14 15 ", " 16 17 18 19 20 21 22 ", " 23 24 25 26
```

ZIPPING



Source: Simon Emmett Photography

GRADING

```
std::vector names{"Joan"s, "Ben"s, "Gina"s, "Tim"s};
std::vector grades{8.5, 7.1, 9.0, 9.5};

std::println("student grades are: {}",
  std::views::zip(names, grades));
```

GRADING

```
std::vector names{"Joan"s, "Ben"s, "Gina"s, "Tim"s};
std::vector grades{8.5, 7.1, 9.0, 9.5};

std::println("student grades are: {}",
  std::views::zip(names, grades));
```

```
[("Joan", 8.5), ("Ben", 7.1), ("Gina", 9), ("Tim", 9.5)]
```

TRANSFORMING

```
std::vector names{"Joan"s, "Ben"s, "Gina"s, "Tim"s};
std::vector grades{8.5, 7.1, 9.0, 9.5};

std::println("{}", std::views::zip_transform(
  [](const std::string_view name, float grade) {
    return std::format("{} got {}", name, grade);
  },
  names, grades));
```

```
["Joan got 8.5", "Ben got 7.1", "Gina got 9", "Tim got 9.5"]
```

REFERENCE AND VALUE TYPE

```
std::vector names{"Joan"s, "Ben"s, "Gina"s, "Tim"s};
std::vector grades{8.5, 7.1, 9.0, 9.5};

using zipped = decltype(std::views::zip(names, grades));
static_assert(std::same_as<
  std::ranges::range_reference_t<zipped>,
  std::tuple<std::string&, double&>>);
static_assert(std::same_as<
  std::ranges::range_value_t<zipped>,
  std::tuple<std::string, double>>);
```

SORTING

```
std::vector names{"Joan"s, "Ben"s, "Gina"s, "Tim"s};
std::vector grades{8.5, 7.1, 9.0, 9.5};

std::ranges::sort(std::views::zip(grades, names),
                  std::ranges::greater{});

std::println("{}", names);
```

```
["Tim", "Gina", "Joan", "Ben"]
```

MORE `tuple` ASSIGNMENT OPERATORS

```
  // [tuple.assign], tuple assignment
  constexpr tuple& operator=(const tuple&);
+ constexpr tuple& operator=(const tuple&) const;
  constexpr tuple& operator=(tuple&&) noexcept(see below);
+ constexpr tuple& operator=(tuple&&) const noexcept(see below);

  // [tuple.special], specialized algorithms
  template<class... Types>
  constexpr void swap(tuple<Types...>& x, tuple<Types...>& y)
    noexcept(see below);
+ <class... Types>
+ constexpr void swap(const tuple<Types...>& x, const tuple<Types...>& y)
+   noexcept(see below);
```

MORE `tuple` CONSTRUCTORS

```
     // [tuple.cnstr], tuple construction
     constexpr explicit(see below) tuple();
     constexpr explicit(see below) tuple(const Types&...);
     template<class... UTypes>
       constexpr explicit(see below) tuple(UTypes&&...);

     tuple(const tuple&) = default;
     tuple(tuple&&) = default;

+    template<class... UTypes>
+      constexpr explicit(see below) tuple(tuple<UTypes...>&);
     template<class... UTypes>
       constexpr explicit(see below) tuple(const tuple<UTypes...>&);
     template<class... UTypes>
       constexpr explicit(see below) tuple(tuple<UTypes...>&&);
+    template<class... UTypes>
+      constexpr explicit(see below) tuple(const tuple<UTypes...>&&);
```

MORE TUPLE VIEWS

```
constexpr std::array v {1, 2, 3, 4, 5, 6};
std::println("{}", v);
for (auto const [index, window]:
  std::views::enumerate(v | std::views::adjacent<3>))
{
  std::println("{:>{}}{}", "", 3*index, window);
}
```

```
[1, 2, 3, 4, 5, 6]
(1, 2, 3)
   (2, 3, 4)
      (3, 4, 5)
         (4, 5, 6)
```

MORE TUPLE VIEWS

```
constexpr std::array v {1, 2, 3, 4, 5, 6};
std::println("{}", v);
for (auto const [index, window]:
  std::views::enumerate(v | std::views::pairwise))
{
  std::println("{:>{}}{}", "", 3*index, window);
}
```

```
[1, 2, 3, 4, 5, 6]
(1, 2)
   (2, 3)
      (3, 4)
         (4, 5)
            (5, 6)
```

MORE TUPLE VIEWS

```
constexpr std::array v {1, 2, 3, 4, 5, 6};
std::println("all pairs = {}",
    std::views::cartesian_product(v, v));
```

```
all pairs = [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 1), ...
```

TRANSPOSING MONTHS

```
// In:  range<range<range<string>>>
// Out: range<range<range<string>>>, transposing months.
auto transpose_months() {
  return views::transform(
    []<nested_range_of<std::string, 2> Rng>(Rng&& rng) {
      const auto begin = ranges::begin(rng);
      return views::zip_transform(
          concat, *begin, *ranges::next(begin),
          *ranges::next(begin, 2));
    }
  );
}
```

TRANSPOSING MONTHS

```
// In:  range<range<range<string>>>
// Out: range<range<range<string>>>, transposing months.
auto transpose_months() {
  return views::transform(
    []<nested_range_of<std::string, 2> Rng>(Rng&& rng) {
      const auto begin = ranges::begin(rng);
      return views::zip_transform(
          concat, *begin, *ranges::next(begin),
          *ranges::next(begin, 2));
    }
  );
}
```

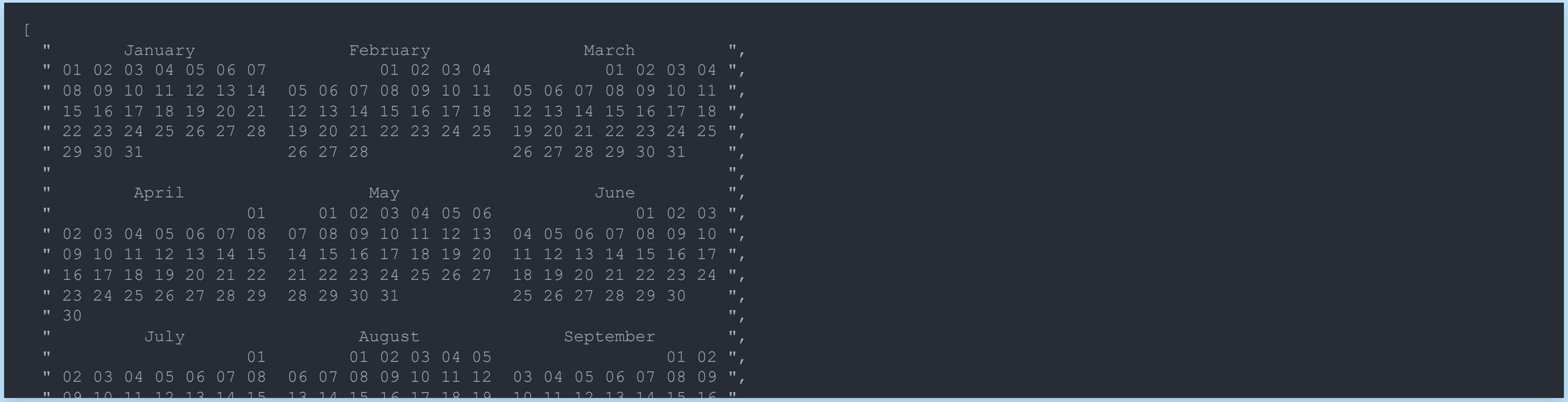
chunk(3)

```
[
  [
    ["       January        ", " 01 02 03 04 05 06 07 ", " 08 09 10 11 12 13 14 ", " 15 16 17 18 19 20 21 ", " 22 23 24 25 26 27 28 ", " 29 30 31
", "                      "],
    ["       February       ", "          01 02 03 04 ", " 05 06 07 08 09 10 11 ", " 12 13 14 15 16 17 18 ", " 19 20 21 22 23 24 25 ", " 26 27 28
", "                      "],
    ["        March         ", "          01 02 03 04 ", " 05 06 07 08 09 10 11 ", " 12 13 14 15 16 17 18 ", " 19 20 21 22 23 24 25 ", " 26 27 28 29
30 31    ", "                      "]
  ],
  [
    ["        April         ", "                   01 ", " 02 03 04 05 06 07 08 ", " 09 10 11 12 13 14 15 ", " 16 17 18 19 20 21 22 ", " 23 24 25 26
27 28 29 ", " 30                 "],
    ["         May          ", "    01 02 03 04 05 06 ", " 07 08 09 10 11 12 13 ", " 14 15 16 17 18 19 20 ", " 21 22 23 24 25 26 27 ", " 28 29 30 31
", "                      "],
    ["         June         ", "             01 02 03 ", " 04 05 06 07 08 09 10 ", " 11 12 13 14 15 16 17 ", " 18 19 20 21 22 23 24 ", " 25 26 27 28
29 30    ", "                      "]
  ],
  [
    ["         July         ", "                   01 ", " 02 03 04 05 06 07 08 ", " 09 10 11 12 13 14 15 ", " 16 17 18 19 20 21 22 ", " 23 24 25 26
```

| transpose_months

```
[
  ["       January               February              March         ", " 01 02 03 04 05 06 07           01 02 03 04           01 02 03 04 ", " 08
09 10 11 12 13 14  05 06 07 08 09 10 11  05 06 07 08 09 10 11 ", " 15 16 17 18 19 20 21  12 13 14 15 16 17 18  12 13 14 15 16 17 18 ", " 22 23 24 25
26 27 28  19 20 21 22 23 24 25  19 20 21 22 23 24 25 ", " 29 30 31              26 27 28              26 27 28 29 30 31    ", "
"],
  ["        April                 May                  June         ", "                   01     01 02 03 04 05 06              01 02 03 ", " 02
03 04 05 06 07 08  07 08 09 10 11 12 13  04 05 06 07 08 09 10 ", " 09 10 11 12 13 14 15  14 15 16 17 18 19 20  11 12 13 14 15 16 17 ", " 16 17 18 19
20 21 22  21 22 23 24 25 26 27  18 19 20 21 22 23 24 ", " 23 24 25 26 27 28 29  28 29 30 31           25 26 27 28 29 30    ", " 30
"],
  ["         July                 August              September       ", "                   01        01 02 03 04 05                 01 02 ", " 02
03 04 05 06 07 08  06 07 08 09 10 11 12  03 04 05 06 07 08 09 ", " 09 10 11 12 13 14 15  13 14 15 16 17 18 19  10 11 12 13 14 15 16 ", " 16 17 18 19
20 21 22  20 21 22 23 24 25 26  17 18 19 20 21 22 23 ", " 23 24 25 26 27 28 29  27 28 29 30 31        24 25 26 27 28 29 30 ", " 30 31
"],
  ["       October               November             December       ", " 01 02 03 04 05 06 07           01 02 03 04                 01 02 ", " 08
09 10 11 12 13 14  05 06 07 08 09 10 11  03 04 05 06 07 08 09 ", " 15 16 17 18 19 20 21  12 13 14 15 16 17 18  10 11 12 13 14 15 16 ", " 22 23 24 25
26 27 28  19 20 21 22 23 24 25  17 18 19 20 21 22 23 ", " 29 30 31              26 27 28 29 30        24 25 26 27 28 29 30 ", "
31                  "]
]
```

| join



```
[
  "       January               February              March          ",
  " 01 02 03 04 05 06 07           01 02 03 04          01 02 03 04 ",
  " 08 09 10 11 12 13 14  05 06 07 08 09 10 11  05 06 07 08 09 10 11 ",
  " 15 16 17 18 19 20 21  12 13 14 15 16 17 18  12 13 14 15 16 17 18 ",
  " 22 23 24 25 26 27 28  19 20 21 22 23 24 25  19 20 21 22 23 24 25 ",
  " 29 30 31              26 27 28              26 27 28 29 30 31    ",
  "                                                                  ",
  "        April                  May                   June         ",
  "                       01        01 02 03 04 05 06           01 02 03 ",
  " 02 03 04 05 06 07 08  07 08 09 10 11 12 13  04 05 06 07 08 09 10 ",
  " 09 10 11 12 13 14 15  14 15 16 17 18 19 20  11 12 13 14 15 16 17 ",
  " 16 17 18 19 20 21 22  21 22 23 24 25 26 27  18 19 20 21 22 23 24 ",
  " 23 24 25 26 27 28 29  28 29 30 31           25 26 27 28 29 30    ",
  " 30                                                               ",
  "         July                August              September        ",
  "                       01           01 02 03 04 05                01 02 ",
  " 02 03 04 05 06 07 08  06 07 08 09 10 11 12  03 04 05 06 07 08 09 ",
  " 09 10 11 12 13 14 15  13 14 15 16 17 18 19  10 11 12 13 14 15 16 "
```

| join_with('\n')

```
      January               February              March
01 02 03 04 05 06 07             01 02 03 04          01 02 03 04
08 09 10 11 12 13 14  05 06 07 08 09 10 11  05 06 07 08 09 10 11
15 16 17 18 19 20 21  12 13 14 15 16 17 18  12 13 14 15 16 17 18
22 23 24 25 26 27 28  19 20 21 22 23 24 25  19 20 21 22 23 24 25
29 30 31              26 27 28              26 27 28 29 30 31

       April                  May                   June
                  01     01 02 03 04 05 06               01 02 03
02 03 04 05 06 07 08  07 08 09 10 11 12 13  04 05 06 07 08 09 10
09 10 11 12 13 14 15  14 15 16 17 18 19 20  11 12 13 14 15 16 17
16 17 18 19 20 21 22  21 22 23 24 25 26 27  18 19 20 21 22 23 24
23 24 25 26 27 28 29  28 29 30 31           25 26 27 28 29 30
30
        July                 August               September
                  01           01 02 03 04 05                  01 02
02 03 04 05 06 07 08  06 07 08 09 10 11 12  03 04 05 06 07 08 09
09 10 11 12 13 14 15  13 14 15 16 17 18 19  10 11 12 13 14 15 16
16 17 18 19 20 21 22  20 21 22 23 24 25 26  17 18 19 20 21 22 23
```

MATERIALIZE

Source: pmtoday.co.uk

TO VECTOR

```
auto numbers = std::views::iota(1, 10);

auto vec = std::ranges::to<std::vector<int>>(numbers);
```

ALLOCATOR ANYONE?

```
auto numbers = std::views::iota(1, 10);

auto vec = std::ranges::to<std::vector<int, Alloc>>(numbers, alloc);
```

TYPE DEDUCTION

```
auto numbers = std::views::iota(1, 10);

auto vec = std::ranges::to<std::vector>(numbers);
```

RANGE CONSTRUCTOR

```
auto numbers = std::views::iota(1, 10);

auto vec = std::vector{std::from_range, numbers};
```

ALGORITHM

`ranges::to<C>(r, args...)`

- constructs C using the first valid from
  - `C{r, args...}`
  - `C{std::from_range, r, args...}`
  - `C{ranges::begin(r), ranges::end(r), args...}`
  - `C c{args...}; c.reserve(std::ranges::size(r)); std::ranges::copy(r, std::back_inserter(c));`
- recurse if needed.

WHY from_range_t?

```
std::list<int> l;
std::vector v{l};
static_assert(std::same_as<decltype(v), std::vector<std::list<int>>>);
```

WHY from_range_t?

```
std::list<int> l;
std::vector v{std::from_range, l};
static_assert(std::same_as<decltype(v), std::vector<int>>);
```

COMPOSABLE

```
auto numbers = std::views::iota(1, 10);

auto vec = numbers | std::ranges::to<std::vector>();
```

VALUE TYPE CONVERSION

```
auto numbers = std::views::iota(1, 10);

auto vec = numbers | std::ranges::to<std::vector<double>>();
```

ASSOCIATIVE CONTAINERS

```
std::vector names{"Joan"s, "Ben"s, "Gina"s, "Tim"s};
std::vector grades{8.5, 7.1, 9.0, 9.5};

auto map = std::views::zip(names, grades)
  | std::ranges::to<std::unordered_map>();
```

NESTED RANGES

```
std::list<std::forward_list<int>> lst = {{0, 1, 2, 3}, {4, 5, 6, 7}};
auto vec = std::ranges::to<std::vector<std::deque<double>>>(lst);
```

USER DEFINED TYPES

```
template<typename T>
class MyContainer {
public:
  MyContainer(std::ranges::range auto&& r);

  T* begin() const;
  T* end() const;
};

template<std::ranges::range Rng>
MyContainer(Rng&& r)
  -> MyContainer<std::ranges::range_value_t<Rng>>;

auto numbers = std::views::iota(1, 10);
auto cont = std::ranges::to<MyContainer>(numbers);
```

MORE METHODS

```
std::vector<int> v;
v.insert_range(v.end(), std::views::iota(1, 10));
v.append_range(std::views::iota(20, 30));
v.assign_range(std::views::iota(30, 100));
```

format_as_string

```
namespace detail {
template <typename Rng>
auto format_as_string(const std::string_view fmt, Rng&& rng) {
  return std::vformat(fmt, std::make_format_args(
#if __cpp_lib_format_ranges < 202207L
    std::forward<Rng>(rng) | ranges::to<std::string>()
#else
  std::forward<Rng>(rng)
#endif
  ));
}
}  // namespace detail
```

LIVE DEMO

FOLDING

Source: www.seco.ac.rgeology.soton.ac.uk

std::fold_left

```cpp
template<input_iterator I, sentinel_for<I> S, class T,
         indirectly_binary_left_foldable<T, I> F>
constexpr auto fold_left(I first, S last, T init, F f);

template<input_range R, class T,
         indirectly_binary_left_foldable<T, iterator_t<R>> F>
constexpr auto fold_left(R&& r, T init, F f);
```

$$f(f(f(f(init, x_1), x_2), \dots), x_n)$$

EXAMPLES

```cpp
#include <algorithm>
#include <vector>

std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};

int sum = std::ranges::fold_left(v, 0, std::plus<int>());
std::println("sum: {}", sum);

int mul = std::ranges::fold_left(v, 1, std::multiplies<int>());
std::println("mul: {}", mul);
```

EXAMPLES

```cpp
#include <algorithm>
#include <vector>

std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};

int sum = std::ranges::fold_left(v, 0, std::plus<int>());
std::println("sum: {}", sum);

int mul = std::ranges::fold_left(v, 1, std::multiplies<int>());
std::println("mul: {}", mul);
```

EXAMPLES

```
std::println("{}",
  std::ranges::fold_left(
    std::views::iota(0, 10),
    "init"s,
    [](std::string&& accum, const int index){
      return std::format("f({},x{})",
                         std::move(accum),
                         utf32_to_str(U'₀' + index));
    }));
```

EXAMPLES

```
template<
  std::input_iterator I, std::sentinel_for<I> S,
  class Proj = std::identity,
  std::indirect_unary_predicate<std::projected<I, Proj>> Pred
>
constexpr std::iter_difference_t<I>
count_if( I first, S last, Pred pred, Proj proj = {} ) {
  return std::ranges::fold_left(first, last,
    std::iter_difference_t<I>{0},
    [&](auto accum, auto&& val) {
      return std::invoke(pred, std::invoke(proj, FWD(val)))
        ? accum + 1
        : accum;
  });
}
```

EXAMPLES

```cpp
template<
  std::input_iterator I, std::sentinel_for<I> S,
  class Proj = std::identity,
  std::indirect_unary_predicate<std::projected<I, Proj>> Pred
>
constexpr std::iter_difference_t<I>
count_if( I first, S last, Pred pred, Proj proj = {} ) {
  return std::ranges::fold_left(first, last,
    std::iter_difference_t<I>{0},
    [&](auto accum, auto&& val) {
      return std::invoke(pred, std::invoke(proj, FWD(val)))
        ? accum + 1
        : accum;
  });
}
```

std::fold_right

```
template<bidirectional_iterator I, sentinel_for<I> S, class T,
         indirectly_binary_right_foldable<T, I> F>
constexpr auto fold_right(I first, S last, T init, F f);

template<bidirectional_range R, class T,
         indirectly_binary_right_foldable<T, iterator_t<R>> F>
constexpr auto fold_right(R&& r, T init, F f);;
```

$$f(x_1, f(x_2, \dots f(x_n, init)))$$

EXAMPLES

```
std::println("{}",
  std::ranges::fold_right(
    std::views::iota(0, 10),
    "init"s,
    [](const int index, std::string&& accum){
      return std::format("f(x{},{})",
                         utf32_to_str(U'₀' + index),
                         std::move(accum));
    }));
```

std::ranges::min

```
template< std::ranges::input_range R >
constexpr std::ranges::range_value_t<R> min( R&& r ) {
  return std::ranges::fold_left(FWD(r),
    ???,
    [&](auto&& min, auto&& next) {
      return next < min ? next : min;
    });
}
```

std::ranges::min

```
template< std::ranges::input_range R >
constexpr std::ranges::range_value_t<R> min( R&& r ) {
  return std::ranges::fold_left(FWD(r),
    ???,
    [&](auto&& min, auto&& next) {
      return next < min ? next : min;
    });
}
```

std::ranges::min

```
1 template< std::ranges::input_range R >
2 constexpr std::ranges::range_value_t<R> min( R&& r ) {
3   return std::ranges::fold_left(FWD(r),
4     *std::ranges::begin(r),
5     [&](auto&& min, auto&& next) {
6       return next < min ? next : min;
7     });
8 }
```

std::ranges::min

```
template< std::ranges::input_range R >
constexpr std::ranges::range_value_t<R> min( R&& r ) {
  return std::ranges::fold_left_first(FWD(r),
    [&](auto&& min, auto&& next) {
      return next < min ? next : min;
    });
}
```

std::ranges::min

```
template< std::ranges::input_range R >
constexpr std::ranges::range_value_t<R> min( R&& r ) {
  return *std::ranges::fold_left_first(FWD(r),
    [&](auto&& min, auto&& next) {
      return next < min ? next : min;
    });
}
```

The law of useful return:

A procedure should return all the potentially useful information it computed.

Alexander Stepanov

The law of useful return:

A procedure should return all the potentially useful information it computed.

Alexander Stepanov

```
auto [_, total] = ranges::fold_left(numbers, 0, std::plus<>{});
```

The law of useful return:

A procedure should return all the potentially useful information it computed.

Alexander Stepanov

```
auto [it, total] = ranges::fold_left_with_iter(numbers, 0, std::plus<>{});
```

ALL THE FOLDS

algorithm	direction	init	return
`fold_left`	left	+	T
`fold_left_first`	left	-	T
`fold_right`	right	+	T
`fold_right_last`	right	-	T
`fold_left_with_iter`	left	+	(it, T)
`fold_left_first_with_iter`	left	-	(it, T)

RESOURCES

- C++23 calendar: https://godbolt.org/z/qvqW8vYrq
- range-v3 calendar: https://github.com/ericniebler/range-v3/blob/master/example/calendar.cpp
- Eric's 2015 talk: https://youtu.be/mFUXNMfaciE
- Berry's 2022 talk about ranges formatting: https://youtu.be/EQELdyecZlU
- A Plan for C++23 Ranges: https://wg21.link/p2214

THANK YOU

dvirtz@gmail.com

github.com/dvirtz

@dvirtzwastaken