C++20 Ranges

from scratch

Paweł Paruzel

Concept Hierarchy

range

concept

range concept

```
template <typename T>
concept range = requires (T& t)
{
  ranges::begin(t);
  ranges::end(t);
};
```

ranges::begin

- Customization point object is a function object that interacts with types while enforcing semantic requirements
- Each CPO type constrains its return type to model a particular concept
- P0970R1
- Poison pills for unqualified lookup

```
void begin(auto&) = delete;
void begin(const auto&) = delete;
```

```
class begin_fn
{
public:
  template <_maybe_borrowed_range T>
    requires is_array_v<remove_reference_t<T>> ||
             _has_member_begin<T> ||
             _has_adl_begin<T>
  constexpr auto operator()(T&& t) const noexcept(...)
  {
    if constexpr (is_array_v<remove_reference_t<T>>)
    {
      static_assert(is_lvalue_reference_v<T>);
      return t + 0;
    }
    else if constexpr (_has_member_begin<T>)
      return t.begin();
    else
      return begin(t);
  }
};

inline constexpr begin_fn begin{};
```

ranges::begin

```
template <typename T>
  requires /* ... */
constexpr input_or_output_iterator auto begin(T&& t);


template <typename It>
concept input_or_output_iterator =
  requires (It i)
  {
    { *i } -> _can_reference;
  } &&
  weakly_incrementable<It>
```

ranges::begin

```
template <typename It>
concept weakly_incrementable =
  movable<It> &&
  requires (It i)
  {
    typename iter_difference_t<It>;
    requires _is_signed_integer_like<iter_difference_t<It>>;
    { ++i } -> same_as<It&>;
    i++;
  };
```

range concept

```
template <typename T>
concept range = requires (T& t)
{
  ranges::begin(t);
  ranges::end(t);
};
```

ranges::end

```
template <typename T>
  requires /* ... */
constexpr sentinel_for<iterator_t<T>> auto end(T&& t);

template <typename T>
using iterator_t = decltype(ranges::begin(declval<T&>()));

template <typename Sent, typename Iter>
concept sentinel_for =
  semiregular<Sent> &&
  input_or_output_iterator<Iter> &&
  _weakly_equality_comparable_with<Sent, Iter>;
```

ranges::end

```
template <typename T, typename U>
concept _weakly_equality_comparable_with
  = requires (const std::remove_reference_t<T>& lhs,
              const std::remove_reference_t<U>& rhs)
  {
    { lhs == rhs } -> _boolean_testable;
    { lhs != rhs } -> _boolean_testable;
    { rhs == lhs } -> _boolean_testable;
    { rhs != lhs } -> _boolean_testable;
  };
```

range concept

```cpp
template <typename T>
concept range = requires (T& t)
{
  ranges::begin(t);
  ranges::end(t);
};
```

borrowed_range

concept

borrowed_range concept

```
template <typename T>
concept borrowed_range =
    range<T> &&
    (is_lvalue_reference_v<T> ||
     enable_borrowed_range<remove_cvref_t<T>>);

template <typename T>
inline constexpr bool enable_borrowed_range = false;
```

borrowed_iterator_t

Example

```
vector<int> v{1, 2};
auto it = iter_first(v);
auto first = *it;
```

```
vector<int> v{1, 2};
auto it = iter_first(move(v));
auto first = *it;
```

```
string s{"Hello"};
auto it = iter_first(string_view{s});
auto first = *it;
```

borrowed_iterator_t

Example

```
auto iter_first(auto&& cont)
{
  using std::begin;
  return begin(cont);
}
```

borrowed_iterator_t

Example

```
auto iter_first(borrowed_range auto&& cont)
{
  using std::begin;
  return begin(cont);
}
```

borrowed_iterator_t

Example

```
auto iter_first(auto&& cont)
  -> borrowed_iterator_t<decltype(cont)>
{
  using std::begin;
  return begin(cont);
}
```

borrowed_iterator_t

Example

```
vector<int> v{1, 2};
auto it = iter_first(v);
auto first = *it;
```

```
vector<int> v{1, 2};
auto it = iter_first(move(v));
auto first = *it;
```

```
string s{"Hello"};
auto it = iter_first(string_view{s});
auto first = *it;
```

borrowed_iterator_t

Example

```
error: no match for 'operator*' (operand type is 'std::ranges::dangling')
   23 |     auto first = *it;
```

```
21 |   vector<int> v{1, 2};
22 |   auto it = iter_first(move(v));
23 |   auto first = *it;
```

borrowed_iterator_t
borrowed_subrange_t

```
template <range R>
using borrowed_iterator_t =
  conditional_t<borrowed_range<R>,
    iterator_t<R>,
    dangling>;

template <range R>
using borrowed_subrange_t =
  conditional_t<borrowed_range<R>,
    subrange<iterator_t<R>>,
    dangling>;
```

sized_range

concept

sized_range concept

```
template <typename T>
concept sized_range =
  range<T> &&
  requires (T& t)
  {
    ranges::size(t);
  };
```

view

concept

view concept

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

template <typename T>
inline constexpr bool enable_view =
  derived_from<T, view_base> ||
  _is_derived_from_view_interface<T>;

struct view_base {};
```

view_interface

```
template <typename Derived>
  requires is_class_v<Derived> && same_as<Derived, remove_cv_t<Derived>>
class view_interface : public view_base
{
  empty         if Derived satisfies forward_range
  operator bool if Derived ranges::empty is applicable
  data          if Derived's iterator_t satisfies contiguous_iterator
  front         if Derived satisfies forward_range
  back          if Derived satisfies bidirectional_range and common_range
  operator[]    if Derived satisfies random_access_range
  size          if Derived satisfies forward_range and its sentinel_t and
                   iterator_t satisfy sized_sentinel_for
};

enable_view<Derived> == true
```

Range concepts based on their iterator

```
input_range
forward_range
bidirectional_range
random_access_range
contiguous_range
```

concepts

iterator-based range concepts

For a template parameter type T, we define concepts

```
input_range := range<T> and input_iterator<iterator_t<T>>

forward_range := input_range<T> and forward_iterator<iterator_t<T>>

bidirectional_range :=
  forward_range<T> and bidirectional_iterator<iterator_t<T>>

random_access_range :=
  bidirectional_range<T> and random_access_iterator<iterator_t<T>>

contiguous_range :=
  random_access_range<T> and
  contiguous_iterator<iterator_t<T>> and
  for an object t of type T&:
    the return type of ranges::data(t) is add_pointer_t<range_reference_t<T>>
```

iterator-based range concepts

For a template parameter type T, we define concepts

```
input_range := range<T> and input_iterator<iterator_t<T>>

forward_range := input_range<T> and forward_iterator<iterator_t<T>>

bidirectional_range :=
  forward_range<T> and bidirectional_iterator<iterator_t<T>>

random_access_range :=
  bidirectional_range<T> and random_access_iterator<iterator_t<T>>

contiguous_range :=
  random_access_range<T> and
  contiguous_iterator<iterator_t<T>> and
  ranges::data(t) returns raw data pointer (no proxy type)
```

<iterator>

library

<iterator>

library

- Iterator concepts (as in C++20 concepts)

<iterator>

library

- Iterator concepts (as in C++20 concepts)
  - √ `input_iterator`
  - √ `output_iterator`
  - √ `forward_iterator`
  - √ `bidirectional_iterator`
  - √ `random_access_iterator`
  - √ `contiguous_iterator`

`<iterator>`

library

- Iterator concepts (as in C++20 concepts)
- `contiguous_iterator_tag`
- `default_sentinel_t`, `unreachable_sentinel_t`
- `common_iterator`
- `input_or_output_iterator`, `sentinel_for`
- `iter_*_t` and their respective `range_*_t` (`<ranges>`)

`iter_*_t`
`range_*_t`
aliases

- `iter_value_t` – `value_type` type trait
- `iter_reference_t` – `reference` type trait
- `iter_difference_t` – `difference_type` type trait
- `iter_rvalue_reference_t` – r-value reference type
- `iter_common_reference_t` – common reference type

`iter_*_t`
`range_*_t`
aliases

- `iterator_t` – iterator type of a range
- `sentinel_t` – sentinel type of a range
- `range_value_t` – `iter_value_t<iterator_t<R>>` of a range R
- `range_reference_t` – `iter_reference_t<iterator_t<R>>` of a range R
- `range_difference_t` – `iter_difference_t<iterator_t<R>>` of a range R
- `range_size_t` – size type of a sized range
- `range_rvalue_reference_t` – `iter_rvalue_reference_t<iterator_t<R>>` of a range R

Views

and view adaptors

subrange

helper view

`ranges::subrange`

- This is not a concept (see `ranges::range`)
- This is a view (inherits from `view_interface`)
- Satisfies a `borrowed_range`
- Constructs a subrange view from an iterator-sentinel pair
- Can convert to any pair-like type (`std::pair`, `std::tuple` of two, etc.)
- Can take a whole range as a parameter

```
auto [begin, end] = ranges::subrange(vec);
```

all, all_t

view adaptors

views::all, views::all_t

```
string_view{} | views::all

decltype:
 string_view
```

views::all, views::all_t

```
vector v{1, 2}
v | views::all

decltype:
 ranges::ref_view<vector<int>>
```

views::all, views::all_t

```
int arr[512]
arr | views::all

decltype:
 ranges::ref_view<int [512]>
```

views::all, views::all_t

```
vector{1, 2} | views::all


decltype:
 ranges::owning_view<vector<int>>
```

take

take_view

views::take, ranges::take_view

```
vector{1, 2} | views::all | views::take(10)
```

views::take, ranges::take_view

```
vector{1, 2} | views::take(10)
```

views::take, ranges::take_view

```
vector{1, 2} | views::take(10)

decltype:
take_view<owning_view<vector<int>>>
```

views::take, ranges::take_view

```
string_view{"Hello"} | views::take(10)

decltype:
 string_view
```

gcc 12+

Take

Example

```
string_view text = "Hello, World";
auto hello = text | views::take(5);

cout << hello;
```

```
Output:
 Hello
```

reverse

reverse_view

views::reverse, ranges::reverse_view

```
string_view{"Hello, World"}
  | views::reverse
  | views::take(5)  // Hello, World
  | views::reverse

decltype:
reverse_view<take_view<reverse_view<string_view>>>
```

views::reverse, ranges::reverse_view

```
views::reverse
  | views::take(5)
  | views::reverse

decltype (for gcc):
_Pipe<_Pipe<_ReverseAdaptor,
            _PartialAdaptor<_TakeAdaptor, int>>,
      _ReverseAdaptor>
```

views::reverse, ranges::reverse_view

```
constexpr auto tail(auto n) noexcept
{
  return views::reverse
       | views::take(n)
       | views::reverse;
}
```

Reverse
Example

```
string_view text = "Hello, World";
auto world = text
  | views::reverse
  | views::take(5)  // Hello, World
  | views::reverse
  ;
for (auto c : world)
  cout << c;

Output:
 World
```

Reverse

Example

```
string_view text = "Hello, World";
auto world =
  views::reverse(views::take(views::reverse(text), 5));



for (auto c : world)
  cout << c;
```

```
Output:
 World
```

View adaptors

`View adaptors`

- Adaptors can be composed in two ways
  - via pipelining syntax: R | C
  - via functional invocations: C(R)
  - assuming C is a range adaptor (closure) object (e.g., `views::take`)
  - and R is a `viewable_range`
- Both ways are equivalent
- Not only is pipelining intuitive but also nice and clean

If C_1 and C_2 are range adaptor closure objects, then $C_1 | C_2$ is also a range adaptor closure object if it is valid

$R \mid C_1 \mid C_2 \mid C_3 \mid C_4$

$$R \mid C_1 \mid C_2 \mid (C_3 \mid C_4)$$

$$R \mid C_1 \mid (C_2 \mid (C_3 \mid C_4))$$

$$R \mid (C_1 \mid (C_2 \mid (C_3 \mid C_4)))$$

$$(R \mid (C_1 \mid (C_2 \mid (C_3 \mid C_4))))$$

$$(R \mid (C_1 \mid (C_2 \mid (C_3 \mid C_4))))$$

Pipes do not resolve in that order. They should work this way as well, nevertheless.

```
r | views::take(5)
views::take(r, 5)
views::reverse(r)
r | views::reverse
```

~~r | views::reverse()~~

View adaptors

- Adaptors can be composed in two ways
  - via pipelining syntax: R | C
  - via functional invocations: C(R)
  - assuming C is a range adaptor (closure) object (e.g., `views::take`)
  - and R is a `viewable_range`
- Both ways are equivalent
- Not only is pipelining intuitive but also nice and clean

`viewable_range` is intended to accept types for which views::all is well-formed `viewable_range` accepts range types that can be safely converted to a view

drop

drop_view

views::drop, ranges::drop_view

```
string_view text = "C++98";
auto cpp = text
  | views::reverse
  | views::drop(2)  // C++98
  | views::reverse
  ;
for (auto c : cpp)
  cout << c;
```

Output:

```
C++
```

drop_while, take_while

drop_while_view, take_while_view

views::drop_while, ranges::drop_while_view

```
string_view text = "Goodbye C++98 - Welcome Modern C++ Era";
auto welcome = text
  | views::drop_while([](char c) { return c != 'W'; })
  ;

for (auto c : welcome)
  cout << c;
```

Output:

```
Welcome Modern C++ Era
```

views::take_while, ranges::take_while_view

```cpp
string_view text = "Goodbye C++98 - Welcome Modern C++ Era";
auto welcome = text
  | views::take_while([](char c) { return c != '-'; })
  ;

for (auto c : welcome)
  cout << c;
```

Output:

```
Goodbye C++98
```

transform

transform_view

views::transform, ranges::transform_view

```
auto to_string = [](auto i) { return to_string(i); }
vector{1, 2} | views::transform(to_string)

decltype:
transform_view<owning_view<vector<int>,
                           lambda_1(auto)>
```

Transform

Example

```
string_view text = "Hello, World";
auto v = text



  ;
for (char c : v)
  cout << c << '-';
```

```
Output:
 H-e-l-l-o-,- -W-o-r-l-d-
```

Transform

Example

```
string_view text = "Hello, World";
auto v = text
  | tail(5)



  ;
for (char c : v)
  cout << c << '-';
```

```
Output:
 W-o-r-l-d-
```

Transform

Example

```
string_view text = "Hello, World";
auto v = text
  | tail(5)
  | views::drop(2)

  ;
for (char c : v)
  cout << c << '-';
```

Output:

r-l-d-

Transform

Example

```
string_view text = "Hello, World";
auto v = text
  | tail(5)
  | views::drop(2)
  | views::transform(triple_char)
  ;
for (string c : v)
  cout << c << '-';

Output:
 rrr-lll-ddd-
```

Transform

Example

```
auto triple_char = [](char c)
{
  return string(3, c);
};
```

Transform

Example

```
string_view text = "Hello, World";
auto v = text
  | views::tail(5)
  | views::drop(2)
  | views::transform(triple_char)
  ;
for (string c : v)
  cout << c << '-';
```

Output:

```
rrr-lll-ddd-
```

join

join_view

Join

Example

```
string_view text = "Hello, World";
auto v = text
  | views::tail(5)
  | views::drop(2)
  | views::transform(triple_char)
  ;
for (string c : v)
  cout << c << '-';
```

Output:

```
rrr-lll-ddd-
```

Join
Example

```
string_view text = "Hello, World";
auto v = text
  | views::tail(5)
  | views::drop(2)
  | views::transform(triple_char)
  | ...;
for (char c : v)
  cout << c << '-';
```

Output:

```
r-r-r-l-l-l-d-d-d-
```

Join
Example

```
string_view text = "Hello, World";
auto v = text
  | views::tail(5)
  | views::drop(2)
  | views::transform(triple_char)
  | views::join;
for (char c : v)
  cout << c << '-';
```

```
Output:
  r-r-r-l-l-l-d-d-d-
```

Join

Example

```
string_view text = "Hello, World";
auto v = text
  | views::tail(5)
  | views::drop(2)
  | views::transform(triple_char)
  | views::join_with('-');  // C++23
for (char c : v)
  cout << c;
```

Output:

```
r-r-r-l-l-l-d-d-d
```

common

common_view

Common
Example

```
string_view text = "Hello, World";
auto v = text
  | views::tail(5) | views::drop(2)
  | views::transform(triple_char)
  | views::join

  ;

string(v.begin(), v.end());  // ❌ fails to compile
```

```
error: no matching function for call to
'std::__cxx11::basic_string<char>::basic_string(std::ranges::join_view<std::ranges::transform_view<std::
ranges::drop_view<std::ranges::reverse_view<std::ranges::take_view<std::ranges::reverse_view<std::basic_
string_view<char> > > > >, main()::<lambda(char)> >::_Iterator<false>,
std::ranges::join_view<std::ranges::transform_view<std::ranges::drop_view<std::ranges::reverse_view<std:
:ranges::take_view<std::ranges::reverse_view<std::basic_string_view<char> > > > >,
main()::<lambda(char)> > >::_Sentinel<false>)'

   43 |   string(v.begin(), v.end());
      |                             ^
```

```
let V be
 join_view<
  transform_view<
   drop_view<
     reverse_view<
       take_view<
         reverse_view<string_view>>>>,
   lambda(char)>>

error: no matching function for call to
  'string::string(V::_Iterator, V::_Sentinel)'

  43 |   string(v.begin(), v.end());
     |                            ^
```

```cpp
template <typename InputIt>
constexpr string(InputIt first, InputIt last);
```

```
template <input_iterator It, sentinel_for<It> Sent>
constexpr string(It first, Sent last);
```

Common
Example

```
string_view text = "Hello, World";
auto v = text
  | views::tail(5) | views::drop(2)
  | views::transform(triple_char)
  | views::join

  ;

string(v.begin(), v.end());  // ❌ fails to compile
```

Common

Example

```
string_view text = "Hello, World";
auto v = text
  | views::tail(5) | views::drop(2)
  | views::transform(triple_char)
  | views::join
  | views::common
  ;

string(v.begin(), v.end());  // ✅ compiles
```

filter

filter_view

Filter
Example

```
string_view text = "Hello, World";
auto v = text
  | views::tail(5) | views::drop(2)
  | views::transform(triple_char) | views::join

  ;
```

```
Content of the view:
  r, r, r, l, l, l, d, d, d
```

Filter

Example

```
string_view text = "Hello, World";
auto v = text
  | views::tail(5) | views::drop(2)
  | views::transform(triple_char) | views::join
  | views::filter([i = 0](char) mutable { return ++i & 1; })
  ;
```

Content of the view:

r, r, r, l, l, l, d, d, d

Filter
Example

```
string_view text = "Hello, World";
auto v = text
  | views::tail(5) | views::drop(2)
  | views::transform(triple_char) | views::join
  | views::filter([i = 0](char) mutable { return ++i & 1; })
  ;
```

```
Content of the view:
  r, r, l, d, d
```

Converting views to containers

view converter adaptor `to<C>`
did not make it into C++20

DISCLAIMER

This is a simple implementation and so it is not as sophisticated as the one Range-v3 implements

View conversion

```
template <typename T>
struct _RangeConverter
{
  template <range R>
  friend constexpr auto operator|(R&& r, _RangeConverter)
  { return T(ranges::begin(r), ranges::end(r)); }







};

template <typename T>
auto to() { return _RangeConverter<T>{}; }
```

View conversion

Example

```
string_view text = "Hello, World";
auto s = text
  | views::tail(5) | views::drop(2)
  | views::transform(triple_char) | views::join
  | views::filter([i = 0](char) mutable { return ++i & 1; })
  | views::common | to<string>()
  ;
cout << s;
```

```
Output:
  rrldd
```

View conversion

Example

```
string_view text = "Hello, World";
auto s = text
  | views::tail(5) | views::drop(2)
  | views::transform(triple_char) | views::join
  | views::filter([i = 0](char) mutable { return ++i & 1; })
  | views::common | to<vector<char>>()
  ;
ranges::copy(s, ostream_iterator<char>{cout});
```

```
Output:
  rrldd
```

View conversion

```
template <typename T, template <typename> typename C = void_t>
struct _RangeConverter
{
  template <range R>
      requires (!same_as<T, void>)
  friend constexpr auto operator|(R&& r, _RangeConverter)
  { return T(ranges::begin(r), ranges::end(r)); }

  template <range R>
      requires (same_as<T, void>)
  friend constexpr auto operator|(R&& r, _RangeConverter)
  { return C(ranges::begin(r), ranges::end(r)); }
};

template <typename T>
auto to() { return _RangeConverter<T>{}; }

template <template <typename> typename C>
auto to() { return _RangeConverter<void, C>{}; }
```

View conversion

Example

```
string_view text = "Hello, World";
auto s = text
  | views::tail(5) | views::drop(2)
  | views::transform(triple_char) | views::join
  | views::filter([i = 0](char) mutable { return ++i & 1; })
  | views::common | to<vector<char>>()
  ;
ranges::copy(s, ostream_iterator<char>{cout});
```

Output:

```
rrldd
```

View conversion

Example

```
string_view text = "Hello, World";
auto s = text
  | views::tail(5) | views::drop(2)
  | views::transform(triple_char) | views::join
  | views::filter([i = 0](char) mutable { return ++i & 1; })
  | views::common | to<vector>()
  ;
ranges::copy(s, ostream_iterator<char>{cout});
```

Output:

```
 rrldd
```

View conversion

Example

```
string_view text = "Hello, World";
auto s = text
  | views::tail(5) | views::drop(2)
  | views::transform(triple_char) | views::join
  | views::filter([i = 0](char) mutable { return ++i & 1; })
  | views::common | to<vector>()
  ;
ranges::copy(s, ostream_iterator<char>{cout});
```

```
Output:
 rrldd
```

Constrained algorithms

library

Constrained algorithms

library

1. Defined in header `<algorithm>` in namespace `std::ranges`

Constrained algorithms

library

1. Defined in header `<algorithm>` in namespace `std::ranges`

```
#include <algorithm>

std::copy        -> std::ranges::copy

std::copy_if     -> std::ranges::copy_if

std::min_element -> std::ranges::min_element
     ^                            ^
     functions                    niebloids (CPOs)
```

Constrained algorithms

library

1. Defined in header `<algorithm>` in namespace `std::ranges`
2. An algorithm invocation parameter can be specified as:
   - an iterator-sentinel pair
   - a range

Constrained algorithms
library

2. An algorithm invocation parameter can be specified as:

- an iterator-sentinel pair, e.g. `std::ranges::copy`

```
input_iterator<Iter>
sentinel_for<Sent, Iter>
weakly_incrementable<Out>
indirectly_copyable<Iter, Out>
```

  ... the call operator will look like this:

```
operator()(Iter first, Sent last, Out result)
```

- a range, e.g. `std::ranges::copy`

```
input_range<Range>
weakly_incrementable<Out>
indirectly_copyable<iterator_t<Range>, Out>
operator()(Range&& r, Out result)
```

Constrained algorithms

library

1. Defined in header `<algorithm>` in namespace `std::ranges`
2. An algorithm invocation parameter can be specified as:
   - an iterator-sentinel pair
   - a range
3. Support for projections and pointer-to-member callables

Constrained algorithms

library

3. Support for projections and pointer-to-member callables

```
struct Integer
{
  int value;
};

vector<Integer> vi = {{1}, {2}, {3}, {4}, {0}, {5}};
auto pos = ranges::min_element(vi, {}, &Integer::value);

distance(vi.begin(), pos) == 4
```

Constrained algorithms

library

1. Defined in header `<algorithm>` in namespace `std::ranges`
2. An algorithm invocation parameter can be specified as:
   - an iterator-sentinel pair
   - a range
3. Support for projections and pointer-to-member callables
4. Return types of most algorithms have been changed to return all potentially useful information computed during the execution of the algorithm

Constrained algorithms

library

4. Return types of most algorithms have been changed to return all potentially useful information computed during the execution of the algorithm

ranges::copy niebloid call operator

```
template <input_iterator Iter,
          sentinel_for<Iter> Sent,
          weakly_incrementable Out>
  requires indirectly_copyable<Iter, Out>
constexpr auto operator()(Iter first, Sent last, Out result) const
  -> ranges::copy_result<Iter, Out>
```

`std::ranges::copy_result`

Constrained algorithms

library

4. Return types of most algorithms have been changed to return all potentially useful information computed during the execution of the algorithm

ranges::copy_result alias

```
template <typename I, typename O>
using copy_result = ranges::in_out_result<I, O>;
```

Constrained algorithms

library

4. Return types of most algorithms have been changed to return all potentially useful information computed during the execution of the algorithm

Return types of constrained algorithms

```
ranges::in_fun_result
ranges::in_in_result
ranges::in_out_result
ranges::in_in_out_result
ranges::in_out_out_result
ranges::min_max_result
ranges::in_found_result
```

Constrained algorithms

library

4. Return types of most algorithms have been changed to return all potentially useful information computed during the execution of the algorithm

Example of the return type of a constrained algorithm

```
auto [in, out] =
  ranges::copy("This is C++20"sv | views::take(8), target);
                     ^
                     |__ in
addressof(*in) == "C++20"sv
addressof(*target) == "This is "sv
addressof(*out) == addressof(target[8])
```

```python
def is_palindrome(s):
  return s == s[::-1]
```

```
def is_palindrome(s):
    return s == s[::-1]
```

```
auto is_palindrome(auto&& v)
{
  return ranges::equal(v, views::reverse(v));
}
```

Constrained algorithms

library

```
constexpr auto is_palindrome = [](ranges::view auto v)
{
  return ranges::equal(v, v | views::reverse);
};

static_assert(is_palindrome("racecar"sv));
```

Constrained algorithms

library

```
constexpr auto is_palindromish = [](ranges::view auto v)
{
  auto r = v
    | views::filter(is_letter)
    | views::transform(to_lowercase);
  return ranges::mismatch(r, r | views::reverse).in1 == r.end();
  //                      ^
};

static_assert(is_palindromish("A Santa lived as a devil at NASA"sv));
```

split

split_view

Split
Example

```
string_view ip_string = "192.168.0.1";
auto str_to_uint = [](auto chars)
{
  unsigned i;
  std::from_chars(chars.data(), chars.data() + chars.size(), i);
  return i;
};
auto v = ip_string
  | views::split('.')
  | views::transform(str_to_uint)  // ❌ does not compile in gcc 11
  ;
```

```
Content of the view:
  192, 168, 0, 1
```

why?

Split
Example

```
string_view ip_string = "192.168.0.1";
auto str_to_uint = [](auto chars)
{
  unsigned i;
  std::from_chars(chars.data(), chars.data() + chars.size(), i);
  return i;
};
auto v = ip_string
  | views::split('.')
  | views::transform(str_to_uint)  // does not compile in gcc 11
  ;

Content of the view:
  192, 168, 0, 1
```

view_interface

```
template <typename Derived>
  requires is_class_v<Derived> && same_as<Derived, remove_cv_t<Derived>>
class view_interface : public view_base
{
  empty          if Derived satisfies forward_range
  operator bool  if Derived ranges::empty is applicable
  data           if Derived's iterator_t satisfies contiguous_iterator
  front          if Derived satisfies forward_range
  back           if Derived satisfies bidirectional_range and common_range
  operator[]     if Derived satisfies random_access_range
  size           if Derived satisfies forward_range and its sentinel_t and
                    iterator_t satisfy sized_sentinel_for
};
```

does this compile in gcc 12?

Split

Example

```
string_view ip_string = "192.168.0.1";
auto str_to_uint = [](auto chars)
{
  unsigned i;
  std::from_chars(chars.data(), chars.data() + chars.size(), i);
  return i;
};
auto v = ip_string
  | views::split('.')
  | views::transform(str_to_uint)  // ✅ compiles in gcc 12
  ;
```

```
Content of the view:
  192, 168, 0, 1
```

lazy_split

lazy_split_view

Lazy split

Example

```
string_view ip_string = "192.168.0.1";
auto str_to_uint = [](auto chars)
{
  unsigned i;
  std::from_chars(chars.data(), chars.data() + chars.size(), i);
  return i;
};
auto v = ip_string
  | views::lazy_split('.')
  | views::transform(str_to_uint)  // ❌ does not compile (as expected ✅)
  ;
```

```
Content of the view:
  192, 168, 0, 1
```

aren't views lazy already?

split vs lazy_split

```
template <typename It, typename V>
concept subview_value_output_iterator =
  view<range_value_t<V>> &&
  output_iterator<It, range_value_t<range_value_t<V>>>;

template <view V, subview_value_output_iterator<V> Out>
void read_and_copy_subview(V view, Out out)
{
  for (auto subview : view)
  {
    ranges::copy(subview, out);
  }
}
```

split vs lazy_split

```
auto ip_string = "192.168.0.1"sv;
auto buffer = vector<char>{};
auto out = back_inserter(buffer);
auto v = ip_string
  | views::transform(CountCalls{})
  | views::split('.')
  ;
read_and_copy_subview(v, out);
```

split vs lazy_split

credit: bit.ly/3Lbv9fg

```
struct CountCalls
{
  int count = 0;

  auto operator()(auto value)
  { ++count; return value; }

  ~CountCalls()
  {
    if (count == 0) { return; }
    cout << "Called " << count << " times\n";
  }
};
```

split vs lazy_split

```
auto ip_string = "192.168.0.1"sv;
auto buffer = vector<char>{};
auto out = back_inserter(buffer);
auto split_contiguous = ip_string
  | views::transform(CountCalls{})
  | views::split('.')
  ;
read_and_copy_subview(split_contiguous, out);
```

```
Output:
Called 19 times
```

split vs lazy_split

```
auto ip_string = "192.168.0.1"sv;
auto buffer = vector<char>{};
auto out = back_inserter(buffer);
auto lazy_split_contiguous = ip_string
  | views::transform(CountCalls{})
  | views::lazy_split('.')
  ;
read_and_copy_subview(lazy_split_contiguous, out);
```

```
Output:
Called 30 times
```

what about single-pass views?

what about `input_range` views?

istream_view

split vs lazy_split

```
auto ip_string_stream = istringstream{"192.168.0.1"};
auto ip_stream = views::istream<char>(ip_string_stream);
auto buffer = vector<char>{};
auto out = back_inserter(buffer);
auto lazy_split_input = ip_stream
  | views::transform(CountCalls{})
  | views::lazy_split('.')
  ;
read_and_copy_subview(lazy_split_input, out);
```

```
Output:
  Called 22 times
```

split vs lazy_split

```
auto ip_string_stream = istringstream{"192.168.0.1"};
auto ip_stream = views::istream<char>(ip_string_stream);
auto buffer = vector<char>{};
auto out = back_inserter(buffer);
auto split_input = ip_stream
  | views::transform(CountCalls{})
  | views::split('.')  // ❌ does not compile (as designed ✅)
  ;
read_and_copy_subview(split_input, out);
```

P221OR2

split vs lazy_split

```
for (auto _ : split_contiguous);
for (auto _ : lazy_split_contiguous);
for (auto _ : lazy_split_input);
// for (auto _ : split_input);
```

```
Output after rerunning previous tests:
Called 11 times
Called 11 times
Called 11 times
```

elements

elements_view
and specializations

Elements

Example

```
map<string, int> physicists = {
  {"Albert Einstein", 1879},
  {"Stephen Hawking", 1948},
  {"Niels Bohr", 1885},
};

for (auto name : physicists | views::elements<0>)
{
  cout << name << '\n';
}
```

Elements

Example

```
map<string, int> physicists = {
  {"Albert Einstein", 1879},
  {"Stephen Hawking", 1948},
  {"Niels Bohr", 1885},
};

for (auto name : physicists | views::keys)
{
  cout << name << '\n';
}
```

Elements

Example

```
map<string, int> physicists = {
  {"Albert Einstein", 1879},
  {"Stephen Hawking", 1948},
  {"Niels Bohr", 1885},
};

for (auto year_born : physicists | views::elements<1>)
{
  cout << year_born << '\n';
}
```

Elements

Example

```
map<string, int> physicists = {
  {"Albert Einstein", 1879},
  {"Stephen Hawking", 1948},
  {"Niels Bohr", 1885},
};

for (auto year_born : physicists | views::values)
{
  cout << year_born << '\n';
}
```

Elements
Example

```
vector<tuple<string, int, int>> physicists = {
  {"Albert Einstein", 1879, 1955},
  {"Stephen Hawking", 1879, 2018},
  {"Niels Bohr", 1879, 1962},
};

for (auto year_died : physicists | views::elements<2>)
{
  cout << year_died << '\n';
}
```

Elements
Example

```
vector<tuple<string_view, int, int>> physicists = {
  {"Albert Einstein", 1879, 1955},
  {"Stephen Hawking", 1879, 2018},
  {"Niels Bohr", 1879, 1962},
};
constexpr auto names_only = views::keys | views::transform([](auto s)
  { return views::take_while(s, [](auto c) { return c != ' '; }); });
for (auto name : physicists | names_only)
{
  ranges::copy(name, std::ostream_iterator<char>(cout));
  cout << ' ';
}
Output:
  Albert Niels Stephen
```

Elements

Example

```
vector<tuple<string_view, int, int>> physicists = {
  {"Albert Einstein", 1879, 1955},
  {"Stephen Hawking", 1879, 2018},
  {"Niels Bohr", 1879, 1962},
};
constexpr auto names_only = views::keys | views::transform([](auto s)
  { return views::take(s, s.find(' ')); });
for (auto name : physicists | names_only)
{
 cout << name << ' ';
}
```

Output:

```
 Albert Niels Stephen
```

counted

view adaptor

`views::counted`

- Only an adaptor closure
- Takes iterator and count as parameters
- `constexpr auto counted(Iterator&& it, DifferenceType&& count)`
- Reuses available views
  - If the iterator is contiguous, produces a `span(to_address(it), count)`
  - If the iterator is random-access, produces a `subrange(it, it + count)`
  - Otherwise: `subrange(counted_iterator(it, count), default_sentinel)`

`counted_iterator` holds its distance to the end of a range

iterator sentinels

iterator sentinels

- Defined in header `<iterator>` in namespace `std`
- Comparisons with these types return whether to stop iterating or continue
- `default_sentinel_t`, `unreachable_sentinel_t`
- `default_sentinel`, `unreachable_sentinel`
- Empty types
- Default sentinel – iterator itself knows when to stop
  - Like a counted iterator that stores the count to the end
- Unreachable sentinel – iterator will never stop iterating
  - Like an infinite series (e.g., `iota_view`)
  - iter == sent is always false

iota

iota_view

views::iota, ranges::iota_view

```
for (auto i : views::iota(10))
{
  cout << i << '\n';
}

auto i = 10;
while (true)
{
  cout << i++ << '\n';
}
```

views::iota, ranges::iota_view

```
for (auto i : views::iota(10, unreachable_sentinel))
{
  cout << i << '\n';
}

auto i = 10;
while (true)
{
  cout << i++ << '\n';
}
```

views::iota, ranges::iota_view

```
for (auto i : views::iota(10, 100))
{
  cout << i << '\n';
}

auto i = 10, bound = 100;
while (i < bound)
{
  cout << i++ << '\n';
}
```

```cpp
auto zstring_length(const char* cstr)
{
  return ranges::find(cstr, unreachable_sentinel, '\0') - cstr;
}
```

views::iota, ranges::iota_view

```cpp
constexpr auto is_prime(auto n)
{
  if (n <= 0) { return false; }
  if (n == 1) { return true; }
  return ranges::none_of(
    views::iota(2, n),
    [=](auto i) { return n % i == 0; }
  );
};
```

views::iota, ranges::iota_view

```
constexpr auto is_prime(auto n)
{
  if (n <= 0) { return false; }
  if (n == 1) { return true; }
  return n == *ranges::find_if(
    views::iota(2, n),
    [=](auto i) { return n % i == 0; }
  );
};
```

Iota

Example

```
auto begin = vec.begin(), it = begin;
auto end = vec.end();
while (it != end)
{
  algorithm_on_iterator(it);
  algorithm_on_value(*it);
  auto index = ranges::distance(begin, it);
  algorithm_on_index(index);
}
```

Iota

Example

```
auto begin = vec.begin(), it = begin;
auto end = vec.end();
while (it != end)
{
  process_adjacent_values(it);
  process_current_value(*it);
  auto len = ranges::distance(begin, it);
  process_current_length(len);
}
```

Iota

Example

```
for (auto it : views::iota(vec.begin(), vec.end()))
{
  process_adjacent_values(it);
  process_current_value(*it);
  auto len = ranges::distance(vec.begin(), it);
  process_current_length(len);
}
```

Iota

Example

```
auto begin = vec.begin(), it = begin;
auto end = vec.end();
while (it != end && !invalid(*it))
{
  process_adjacent_values(it);
  process_current_value(*it);
  auto len = ranges::distance(begin, it);
  process_current_length(len);
}
```

Iota

Example

```
for (auto it : views::iota(vec.begin()))
{
  if (it == vec.end() || invalid(*it)) { break; }
  process_adjacent_values(it);
  process_current_value(*it);
  auto len = ranges::distance(vec.begin(), it);
  process_current_length(len);
}
```

empty

empty_view

`views::empty, ranges::empty_view`

- `empty_view<T>` – range factory that produces a view of a specified type with no elements
- `empty<T>` – variable template for `empty_view`
- Satisfies a borrowed range
- Like other views, provides a set of member functions
- Some of the member functions are unsafe e.g., `operator[]`, `front()`, `back()`
- `constexpr`, thus, very likely to be optimized out

Empty

Example

```
template <std::ranges::range Msgs = ranges::empty_view<Message>>
  requires same_as<ranges::range_value_t<Msgs>, Message>
constexpr void flush_messages(Msgs&& leftover = {})
{
  for (Message message : leftover)
  {
    // imagine sophisticated logic here...
    notify_all(message);
  }
  notify_all(msg::flush);
}

flush_messages();
flush_messages(views::empty<Message>);
flush_messages(vector<Message>{"Threshold exceeded", "Unauthorized user"});
```

single

single_view

basically useless

basically useless

yet

basically useless

yet

as of C++20

views::single, ranges::single_view

- Takes its argument by value
- Copying the view, copies the element
- Will probably start to get more useful when `views::concat` is introduced
- Sample usage:

```
auto question = views::single("What is the meaning of life?"s);
auto answer = views::single(42) | views::transform(to_string);
auto q_a = views::concat(question, answer);  // not C++20
for (string sentence : q_a)
  cout << sentence << '\n';
```

```
What is the meaning of Life?
42
```

Implementing a view

Live coding

```cpp
#include <tuple>
#include <ranges>
#include <iostream>
#include <iterator>
#include <string_view>

using namespace std::literals;

namespace myranges
{
template <std::ranges::view Prompt>
class prompt_line_view : public std::ranges::view_interface<prompt_line_view<Prompt>>
{
public:
  struct iterator;

  explicit prompt_line_view(Prompt prompt, std::istream& stream)
    : prompt_{prompt}
    , stream_{&stream}
  {
  }

  auto begin()
  {
    yield_with_prompt();
    return iterator{this};
  }

  auto end()
  {
    return std::default_sentinel;
  }

  void yield_with_prompt()
  {
    std::cout << prompt_ << std::flush;
    std::getline(*stream_, line_);
  }

private:
  Prompt prompt_;
  std::istream* stream_{nullptr};
  std::string line_{};
};

template <typename T>
prompt_line_view(T prompt, auto&&...) -> prompt_line_view<std::views::all_t<T>>;

template <typename Prompt>
struct prompt_line_view<Prompt>::iterator
{
  using value_type = std::string;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::input_iterator_tag;

  prompt_line_view* parent;

  auto& operator++()
  {
    parent->yield_with_prompt();
    return *this;
  }

  void operator++(int) { ++*this; }
  const auto& operator*() const { return parent->line_; }
  bool operator==(std::default_sentinel_t) const { return !*(parent->stream_); }
};
}  // namespace myranges
```

```cpp
namespace myviews
{
namespace detail
{
template <typename Adaptor, typename... Args>
struct partial_fn
{
  std::tuple<Args...> args;

  partial_fn(Args... args)
    : args{std::move(args)...}
  {
  }

  template <typename R>
  friend decltype(auto) operator|(R&& r, partial_fn adaptor)
  {
    return std::apply([&r](auto&&... args)
      {
        return Adaptor{}(std::forward<R>(r), static_cast<std::unwrap_ref_decay_t<Args>&&>(args)...);
      }, adaptor.args);
  }
};
}
struct prompt_line_fn
{
  template <typename R>
  auto operator()(R&& r, std::istream& stream) const
  {
    return myranges::prompt_line_view{std::forward<R>(r), stream};
  }

  auto operator()(std::istream& stream) const
  {
    return detail::partial_fn<prompt_line_fn, std::reference_wrapper<std::istream>>{stream};
  }

  template <typename R>
  friend decltype(auto) operator|(R&& r, prompt_line_fn adaptor)
  {
    return adaptor(std::forward<R>(r), std::cin);
  }
};

inline constexpr prompt_line_fn prompt_line{};
}  // namespace myviews

template <typename T>
inline constexpr bool std::ranges::enable_borrowed_range<myranges::prompt_line_view<T>> = true;

int main()
{
  using V = decltype(myranges::prompt_line_view("test">"sv, std::cin));
  static_assert(std::input_or_output_iterator<std::ranges::iterator_t<V>>);
  static_assert(std::ranges::range<V>);
  static_assert(std::ranges::view<V>);
  static_assert(std::ranges::input_range<V>);
  static_assert(std::ranges::borrowed_range<V>);

  auto cli_loop =
    // myranges::prompt_line_view("sqlite> "sv, std::cin)
    "sqlite> "sv
    | myviews::prompt_line(std::cin)
    | std::views::take_while([](auto command) { return command != ".exit"sv; });
  for (auto&& line : cli_loop)
  {
    std::cout << "handle_command(\"" << line << "\")\n";
  }
}
```

Thank you