Range views

Егор Парошин
Yandex

В C++20 появились `ranges`

```
 1 template< std::random_access_iterator I, std::sentinel_for<I> S,
 2     class Comp = ranges::less, class Proj = std::identity >
 3 requires std::sortable<I, Comp, Proj>
 4 constexpr I sort( I first, S last, Comp comp = {}, Proj proj = {} );
 5
 6 template< ranges::random_access_range R, class Comp = ranges::less,
 7     class Proj = std::identity >
 8 requires std::sortable<ranges::iterator_t<R>, Comp, Proj>
 9 constexpr ranges::borrowed_iterator_t<R>
10     sort( R&& r, Comp comp = {}, Proj proj = {} );
```

- Основаны на ranges-v3
- begin() и end() могут иметь разные типы
- новые версии алгоримов
- отображения

В C++20 появились `ranges`

```
1  template< std::random_access_iterator I, std::sentinel_for<I> S,
2      class Comp = ranges::less, class Proj = std::identity >
3  requires std::sortable<I, Comp, Proj>
4  constexpr I sort( I first, S last, Comp comp = {}, Proj proj = {} );
5
6  template< ranges::random_access_range R, class Comp = ranges::less,
7      class Proj = std::identity >
8  requires std::sortable<ranges::iterator_t<R>, Comp, Proj>
9  constexpr ranges::borrowed_iterator_t<R>
10     sort( R&& r, Comp comp = {}, Proj proj = {} );
```

- Основаны на ranges-v3
- begin() и end() могут иметь разные типы
- новые версии алгоримов
- отображения

В C++20 появились `ranges`

```
 1 template< std::random_access_iterator I, std::sentinel_for<I> S,
 2     class Comp = ranges::less, class Proj = std::identity >
 3 requires std::sortable<I, Comp, Proj>
 4 constexpr I sort( I first, S last, Comp comp = {}, Proj proj = {} );
 5
 6 template< ranges::random_access_range R, class Comp = ranges::less,
 7     class Proj = std::identity >
 8 requires std::sortable<ranges::iterator_t<R>, Comp, Proj>
 9 constexpr ranges::borrowed_iterator_t<R>
10     sort( R&& r, Comp comp = {}, Proj proj = {} );
```

- Основаны на ranges-v3
- begin() и end() могут иметь разные типы
- новые версии алгоримов
- отображения

`range views`, позволяют строить алгоритмы с помощью композиции лекговесных объектов

```
 1
 2 namespace rv = std::ranges::views;
 3 int main()
 4 {
 5     auto const ints = {0, 1, 2, 3, 4, 5};
 6     auto even = [](int i) { return 0 == i % 2; };
 7     auto square = [](int i) { return i * i; };
 8
 9     // a traditional "functional" composing syntax:
10     for (int i : rv::transform(rv::filter(ints, even), square))
11         std::cout << i << ' ';
12
13     std::cout << '\n';
14
15     // the "pipe" syntax of composing the views:
16     for (int i : ints | rv::filter(even) | rv::transform(square))
17         std::cout << i << ' ';
18 }
```

`range views`, позволяют строить алгоритмы с помощью композиции лекговесных объектов

```
 1
 2 namespace rv = std::ranges::views;
 3 int main()
 4 {
 5     auto const ints = {0, 1, 2, 3, 4, 5};
 6     auto even = [](int i) { return 0 == i % 2; };
 7     auto square = [](int i) { return i * i; };
 8
 9     // a traditional "functional" composing syntax:
10     for (int i : rv::transform(rv::filter(ints, even), square))
11         std::cout << i << ' ';
12
13     std::cout << '\n';
14
15     // the "pipe" syntax of composing the views:
16     for (int i : ints | rv::filter(even) | rv::transform(square))
17         std::cout << i << ' ';
18 }
```

`range views`, позволяют строить алгоритмы с помощью композиции лекговесных объектов

```
 1
 2 namespace rv = std::ranges::views;
 3 int main()
 4 {
 5     auto const ints = {0, 1, 2, 3, 4, 5};
 6     auto even = [](int i) { return 0 == i % 2; };
 7     auto square = [](int i) { return i * i; };
 8
 9     // a traditional "functional" composing syntax:
10     for (int i : rv::transform(rv::filter(ints, even), square))
11         std::cout << i << ' ';
12
13     std::cout << '\n';
14
15     // the "pipe" syntax of composing the views:
16     for (int i : ints | rv::filter(even) | rv::transform(square))
17         std::cout << i << ' ';
18 }
```

Композируемость

Композируемость

```
 1 vector ints{1, 2, 3, 4, 5};
 2
 3 auto is_even = [](int x) { return x % 2 == 0; };
 4
 5 vector<int> evens;
 6
 7 copy_if(
 8     ints.begin(),
 9     ints.end(),
10     back_inserter(evens),
11     is_even
12 );
```

Композируемость

```
 1 vector ints{1, 2, 3, 4, 5};
 2
 3 auto is_even = [](int x) { return x % 2 == 0; };
 4
 5 vector<int> evens;
 6
 7 copy_if(
 8     ints.begin(),
 9     ints.end(),
10     back_inserter(evens),
11     is_even
12 );
```

- Копируем только четные числа

How to combine std::copy_if and std::transform?

```
 1 vector ints{1, 2, 3, 4, 5};
 2
 3 auto is_even = [](int x) { return x % 2 == 0; };
 4
 5 vector<int> even_squares;
 6
 7 for_each(
 8     ints.begin(),
 9     ints.end(),
10     [&] (int val) {
11         if (is_even(val)) {
12             even_squares.push_back(val * val);
13         }
14     }
15 );
```

- А как комбинировать алгоритмы?

transform_if

ответ со stackoverflow

```
1  template< class InputIt, class OutputIt, class UnaryPred, class UnaryOp >
2  OutputIt transform_if( InputIt first, InputIt last, OutputIt d_first, UnaryPred
   pred, UnaryOp unary_op);
3
4  auto is_even = [](int x) { return x % 2 == 0; };
5  auto square = [](int x) { return x * x; };
6
7  transform_if(
8    ints.begin(),
9    ints.end(),
10   back_inserter(even_squares),
11   is_even,
12   square
13 );
```

- Придется писать новую функцию

transform_if

ответ со stackoverflow

```
1  template< class InputIt, class OutputIt, class UnaryPred, class UnaryOp >
2  OutputIt transform_if( InputIt first, InputIt last, OutputIt d_first, UnaryPred
   pred, UnaryOp unary_op);
3
4  auto is_even = [](int x) { return x % 2 == 0; };
5  auto square = [](int x) { return x * x; };
6
7  transform_if(
8    ints.begin(),
9    ints.end(),
10   back_inserter(even_squares),
11   is_even,
12   square
13 );
```

- Придется писать новую функцию

transform_if

ответ со stackoverflow

```
1  template< class InputIt, class OutputIt, class UnaryPred, class UnaryOp >
2  OutputIt transform_if( InputIt first, InputIt last, OutputIt d_first, UnaryPred
   pred, UnaryOp unary_op);
3
4  auto is_even = [](int x) { return x % 2 == 0; };
5  auto square = [](int x) { return x * x; };
6
7  transform_if(
8    ints.begin(),
9    ints.end(),
10   back_inserter(even_squares),
11   is_even,
12   square
13 );
```

- Придется писать новую функцию
- функции из `algorithm` не композируются

filter + transform

```
1 auto is_even = [](int x) { return x % 2 == 0; };
2 auto square = [](int x) { return x * x; };
3
4 ranges::copy(
5     rv::transform(
6         rv::filter(ints), is_even),
7         square),
8     back_inserter(even_squares));
```

filter + transform

```
1 auto is_even = [](int x) { return x % 2 == 0; };
2 auto square = [](int x) { return x * x; };
3
4 ranges::copy(
5     rv::transform(
6         rv::filter(ints), is_even),
7         square),
8     back_inserter(even_squares));
```

- Отображения композируемы

filter + transform

```
1 auto is_even = [](int x) { return x % 2 == 0; };
2 auto square = [](int x) { return x * x; };
3
4 ranges::copy(
5     rv::transform(
6         rv::filter(ints), is_even),
7         square),
8     back_inserter(even_squares));
```

- Отображения композируемы
- Отображения ленивы

filter + transform

```
1 auto is_even = [](int x) { return x % 2 == 0; };
2 auto square = [](int x) { return x * x; };
3
4 ranges::copy(
5     rv::transform(
6         rv::filter(ints), is_even),
7         square),
8     back_inserter(even_squares));
```

- Отображения композируемы
- Отображения ленивы
- Pipe-syntax поможет повысить читаемость

filter + transform

```
1 auto is_even = [](int x) { return x % 2 == 0; };
2 auto square = [](int x) { return x * x; };
3
4 ranges::copy(
5     rv::transform(
6         ints | rv::filter(is_even),
7         square),
8     back_inserter(even_squares));
```

- Отображения композируемы
- Отображения ленивы
- Pipe-syntax поможет повысить читаемость

filter + transform

```
1 auto is_even = [](int x) { return x % 2 == 0; };
2 auto square = [](int x) { return x * x; };
3
4 ranges::copy(
5     ints | rv::filter(is_even) | rv::transform(square),
6     back_inserter(even_squares));
```

- Отображения композируемы
- Отображения ленивы
- Pipe-syntax поможет повысить читаемость

filter + transform

```
1 auto is_even = [](int x) { return x % 2 == 0; };
2 auto square = [](int x) { return x * x; };
3
4 auto even_squares = ints
5     | rv::filter(is_even)
6     | rv::transform(square)
7     | rv::to<vector<int>>();
```

- В C++23 появился `ranges::views::to`

Обрезание пробелов

```
 1 auto trim_front() {
 2     return rv::drop_while(isspace);
 3 }
 4
 5 auto trim_back() {
 6     return rv::reverse
 7             | rv::drop_while(isspace)
 8             | rv::reverse;
 9 }
10
11 string trim_string(const string& str) {
12     return str
13         | trim_front()
14         | trim_back()
15         | to<string>();
16 }
```

Источник

Чем мы платим?

- Есть ли накладные расходы?
- Неожиданное поведение?

Как тестировал

- код собирался clang 19.1.0 и gcc 14.2
- x86_64
- будем проверять на contiguous последовательностях
- попробуем посмотреть на накладные расходы на итерацию

enumerate_view

```
1 void enumerate(std::span<int> s, void (*funcPtr)(int, int)) {
2     for(auto&& [i, e] : rv::enumerate(s)) {
3         funcPtr(i, e);
4     }
5 }
```

```
1 void forloop(std::span<int> s, void (*funcPtr)(int, int)) {
2     for(int i = 0; i < s.size(); ++i) {
3         funcPtr(i, s[i]);
4     }
5 }
```

enumerate_view

```
1 void enumerate(std::span<int> s, void (*funcPtr)(int, int)) {
2     for(auto&& [i, e] : rv::enumerate(s)) {
3         funcPtr(i, e);
4     }
5 }
```

```
1 void forloop(std::span<int> s, void (*funcPtr)(int, int)) {
2     for(int i = 0; i < s.size(); ++i) {
3         funcPtr(i, s[i]);
4     }
5 }
```

- Что эффективнее?

enumerate_view

```
1 void enumerate(std::span<int> s, void (*funcPtr)(int, int)) {
2     for(auto&& [i, e] : rv::enumerate(s)) {
3         funcPtr(i, e);
4     }
5 }
```

```
1 void forloop(std::span<int> s, void (*funcPtr)(int, int)) {
2     for(int i = 0; i < s.size(); ++i) {
3         funcPtr(i, s[i]);
4     }
5 }
```

- Что эффективнее?
- Начиная с O1 и clang и gcc генерируют абсолютно одинаковый код для обоих примеров

filter_view

```
1 void filter(std::span<int> s, bool (*pred)(int), void (*func)(int)) {
2     for (auto&& e : s | rv::filter(pred)) {
3         func(e);
4     }
5 }
```

```
1
2 void forloop(std::span<int> s, bool (*pred)(int), void (*func)(int)) {
3     for (auto && e : s) {
4         if (pred(e)) {
5             func(e);
6         }
7     }
8 }
```

filter_view

```
1 void filter(std::span<int> s, bool (*pred)(int), void (*func)(int)) {
2     for (auto&& e : s | rv::filter(pred)) {
3         func(e);
4     }
5 }
```

```
1
2 void forloop(std::span<int> s, bool (*pred)(int), void (*func)(int)) {
3     for (auto && e : s) {
4         if (pred(e)) {
5             func(e);
6         }
7     }
8 }
```

- Начиная с O2 код генерируется разный, но очень похожий

filter_view

```
1 void filter(std::span<int> s, bool (*pred)(int), void (*func)(int)) {
2     for (auto&& e : s | rv::filter(pred)) {
3         func(e);
4     }
5 }
```

```
1
2 void forloop(std::span<int> s, bool (*pred)(int), void (*func)(int)) {
3     for (auto && e : s) {
4         if (pred(e)) {
5             func(e);
6         }
7     }
8 }
```

- Начиная с O2 код генерируется разный, но очень похожий
- Как понять что быстрее?

filter_view

```
1 void filter(std::span<int> s, bool (*pred)(int), void (*func)(int)) {
2     for (auto&& e : s | rv::filter(pred)) {
3         func(e);
4     }
5 }
```

```
1
2 void forloop(std::span<int> s, bool (*pred)(int), void (*func)(int)) {
3     for (auto && e : s) {
4         if (pred(e)) {
5             func(e);
6         }
7     }
8 }
```

- Начиная с O2 код генерируется разный, но очень похожий
- Как понять что быстрее?
- Бенчмарк

filter_view

	gcc	**clang**
Filter	3804561 ns	2780496 ns
ForLoop	3990171 ns	2701332 ns

С O2 разница на уровне погрешности

filter_view

А что если предикат можно заинлайнить?

```
 1 void filter(std::span<int> s, void (*func)(int)) {
 2     for (auto && e : s | rv::filter([](auto x) { return x % 2 == 0; })) {
 3         func(e);
 4     }
 5 }
 6
 7 void forloop(std::span<int> s, void (*func)(int)) {
 8     for (auto && e : s) {
 9         if (e % 2 == 0) {
10             func(e);
11         }
12     }
13 }
```

filter_view

	gcc	**clang**
Filter	3126847 ns	2786019 ns
ForLoop	3123568 ns	2704692 ns

разница опять на уровне погрешности

Выводы

Выводы

- компиляторы уже имеют генерировать эффективный код

Выводы

- компиляторы уже имеют генерировать эффективный код
- если тело цикла непустое, разница еще меньше

Выводы

- компиляторы уже имеют генерировать эффективный код
- если тело цикла непустое, разница еще меньше
- Отображения часто являются zero-cost abstraction
- Но, что может случиться при композиции?

Pull model

```
 1 void print_range(auto range) {
 2     for (const auto& elem : range) {
 3         cout << "number: " << elem << endl;
 4     }
 5 }
 6
 7 auto is_even = [](int elem) {
 8     cout << "filter: " << elem << endl;
 9     return elem % 2 == 0;
10 };
11
12 auto square = [](int elem) {
13     cout << "transform: " << elem << endl;
14     return elem * elem;
15 };
16
17 std::vector v{1, 2, 3, 4};
18
19 print_range(v
20             | rv::filter(is_even)
21             | rv::transform(square));
```

- Сколько раз будут вызваны `is_even` и `square`?

Pull model

```
 1 void print_range(auto range) {
 2     for (const auto& elem : range) {
 3         cout << "number: " << elem << endl;
 4     }
 5 }
 6
 7 auto is_even = [](int elem) {
 8     cout << "filter: " << elem << endl;
 9     return elem % 2 == 0;
10 };
11
12 auto square = [](int elem) {
13     cout << "transform: " << elem << endl;
14     return elem * elem;
15 };
16
17 std::vector v{1, 2, 3, 4};
18
19 print_range(v
20             | rv::filter(is_even)
21             | rv::transform(square));
```

- Сколько раз будут вызваны `is_even` и `square`?

Pull model

```
1  void print_range(auto range) {
2      for (const auto& elem : range) {
3          cout << "number: " << elem << endl;
4      }
5  }
6
7  auto is_even = [](int elem) {
8      cout << "filter: " << elem << endl;
9      return elem % 2 == 0;
10 };
11
12 auto square = [](int elem) {
13     cout << "transform: " << elem << endl;
14     return elem * elem;
15 };
16
17 std::vector v{1, 2, 3, 4};
18
19 print_range(v
20             | rv::filter(is_even)
21             | rv::transform(square));
```

- Сколько раз будут вызваны `is_even` и `square`?

Pull model

```
 1 void print_range(auto range) {
 2     for (const auto& elem : range) {
 3         cout << "number: " << elem << endl;
 4     }
 5 }
 6
 7 auto is_even = [](int elem) {
 8     cout << "filter: " << elem << endl;
 9     return elem % 2 == 0;
10 };
11
12 auto square = [](int elem) {
13     cout << "transform: " << elem << endl;
14     return elem * elem;
15 };
16
17 std::vector v{1, 2, 3, 4};
18
19 print_range(v
20             | rv::filter(is_even)
21             | rv::transform(square));
```

- Сколько раз будут вызваны `is_even` и `square`?

Pull model

```
1  void print_range(auto range) {
2      for (const auto& elem : range) {
3          cout << "number: " << elem << endl;
4      }
5  }
6
7  auto is_even = [](int elem) {
8      cout << "filter: " << elem << endl;
9      return elem % 2 == 0;
10 };
11
12 auto square = [](int elem) {
13     cout << "transform: " << elem << endl;
14     return elem * elem;
15 };
16
17 std::vector v{1, 2, 3, 4};
18
19 print_range(v
20             | rv::filter(is_even)
21             | rv::transform(square));
```

- Сколько раз будут вызваны `is_even` и `square`?

Pull model

```
 1 void print_range(auto range) {
 2     for (const auto& elem : range) {
 3         cout << "number: " << elem << endl;
 4     }
 5 }
 6
 7 auto is_even = [](int elem) {
 8     cout << "filter: " << elem << endl;
 9     return elem % 2 == 0;
10 };
11
12 auto square = [](int elem) {
13     cout << "transform: " << elem << endl;
14     return elem * elem;
15 };
16
17 std::vector v{1, 2, 3, 4};
18
19 print_range(v
20             | rv::filter(is_even)
21             | rv::transform(square));
```

```
1 filter: 1
2 filter: 2
3 transform: 2
4 number: 4
5 filter: 3
6 filter: 4
7 transform: 4
8 number: 16
```

- Сколько раз будут вызваны `is_even` и `square`?

Pull model

```cpp
void print_range(auto range) {
    for (const auto& elem : range) {
        cout << "number: " << elem << endl;
    }
}

auto is_even = [](int elem) {
    cout << "filter: " << elem << endl;
    return elem % 2 == 0;
};

auto square = [](int elem) {
    cout << "transform: " << elem << endl;
    return elem * elem;
};

std::vector v{1, 2, 3, 4};

print_range(v
            | rv::filter(is_even)
            | rv::transform(square));
```

```
filter: 1
filter: 2
transform: 2
number: 4
filter: 3
filter: 4
transform: 4
number: 16
```

- Сколько раз будут вызваны `is_even` и `square`?

Pull model

```cpp
void print_range(auto range) {
    for (const auto& elem : range) {
        cout << "number: " << elem << endl;
    }
}

auto is_even = [](int elem) {
    cout << "filter: " << elem << endl;
    return elem % 2 == 0;
};

auto square = [](int elem) {
    cout << "transform: " << elem << endl;
    return elem * elem;
};

std::vector v{1, 2, 3, 4};

print_range(v
            | rv::filter(is_even)
            | rv::transform(square));
```

```
filter: 1
filter: 2
transform: 2
number: 4
filter: 3
filter: 4
transform: 4
number: 16
```

- Сколько раз будут вызваны `is_even` и `square`?

Pull model

```cpp
void print_range(auto range) {
    for (const auto& elem : range) {
        cout << "number: " << elem << endl;
    }
}

auto is_even = [](int elem) {
    cout << "filter: " << elem << endl;
    return elem % 2 == 0;
};

auto square = [](int elem) {
    cout << "transform: " << elem << endl;
    return elem * elem;
};

std::vector v{1, 2, 3, 4};

print_range(v
            | rv::filter(is_even)
            | rv::transform(square));
```

```
filter: 1
filter: 2
transform: 2
number: 4
filter: 3
filter: 4
transform: 4
number: 16
```

- Сколько раз будут вызваны `is_even` и `square`?

Pull model

```
 1 void print_range(auto range) {
 2     for (const auto& elem : range) {
 3         cout << "number: " << elem << endl;
 4     }
 5 }
 6
 7 auto is_even = [](int elem) {
 8     cout << "filter: " << elem << endl;
 9     return elem % 2 == 0;
10 };
11
12 auto square = [](int elem) {
13     cout << "transform: " << elem << endl;
14     return elem * elem;
15 };
16
17 std::vector v{1, 2, 3, 4};
18
19 print_range(v
20             | rv::transform(square)
21             | rv::filter(is_even));
```

- А теперь?

Pull model

```
 1 void print_range(auto range) {
 2     for (const auto& elem : range) {
 3         cout << "number: " << elem << endl;
 4     }
 5 }
 6
 7 auto is_even = [](int elem) {
 8     cout << "filter: " << elem << endl;
 9     return elem % 2 == 0;
10 };
11
12 auto square = [](int elem) {
13     cout << "transform: " << elem << endl;
14     return elem * elem;
15 };
16
17 std::vector v{1, 2, 3, 4};
18
19 print_range(v
20             | rv::transform(square)
21             | rv::filter(is_even));
```

- А теперь?

Pull model

```cpp
void print_range(auto range) {
    for (const auto& elem : range) {
        cout << "number: " << elem << endl;
    }
}

auto is_even = [](int elem) {
    cout << "filter: " << elem << endl;
    return elem % 2 == 0;
};

auto square = [](int elem) {
    cout << "transform: " << elem << endl;
    return elem * elem;
};

std::vector v{1, 2, 3, 4};

print_range(v
            | rv::transform(square)
            | rv::filter(is_even));
```

```
transform: 1
filter: 1
transform: 2
filter: 4
transform: 2
number: 4
transform: 3
filter: 9
transform: 4
filter: 16
transform: 4
number: 16
```

- А теперь?

Pull model

```cpp
void print_range(auto range) {
    for (const auto& elem : range) {
        cout << "number: " << elem << endl;
    }
}

auto is_even = [](int elem) {
    cout << "filter: " << elem << endl;
    return elem % 2 == 0;
};

auto square = [](int elem) {
    cout << "transform: " << elem << endl;
    return elem * elem;
};

std::vector v{1, 2, 3, 4};

print_range(v
            | rv::transform(square)
            | rv::filter(is_even));
```

```
transform: 1
filter: 1
transform: 2
filter: 4
transform: 2
number: 4
transform: 3
filter: 9
transform: 4
filter: 16
transform: 4
number: 16
```

- А теперь?

Pull model

```cpp
void print_range(auto range) {
    for (const auto& elem : range) {
        cout << "number: " << elem << endl;
    }
}

auto is_even = [](int elem) {
    cout << "filter: " << elem << endl;
    return elem % 2 == 0;
};

auto square = [](int elem) {
    cout << "transform: " << elem << endl;
    return elem * elem;
};

std::vector v{1, 2, 3, 4};

print_range(v
            | rv::transform(square)
            | rv::filter(is_even));
```

```
transform: 1
filter: 1
transform: 2
filter: 4
transform: 2
number: 4
transform: 3
filter: 9
transform: 4
filter: 16
transform: 4
number: 16
```

- А теперь?

Pull model

```cpp
void print_range(auto range) {
    for (const auto& elem : range) {
        cout << "number: " << elem << endl;
    }
}

auto is_even = [](int elem) {
    cout << "filter: " << elem << endl;
    return elem % 2 == 0;
};

auto square = [](int elem) {
    cout << "transform: " << elem << endl;
    return elem * elem;
};

std::vector v{1, 2, 3, 4};

print_range(v
            | rv::transform(square)
            | rv::filter(is_even));
```

```
transform: 1
filter: 1
transform: 2
filter: 4
transform: 2
number: 4
transform: 3
filter: 9
transform: 4
filter: 16
transform: 4
number: 16
```

- А теперь?

Pull model

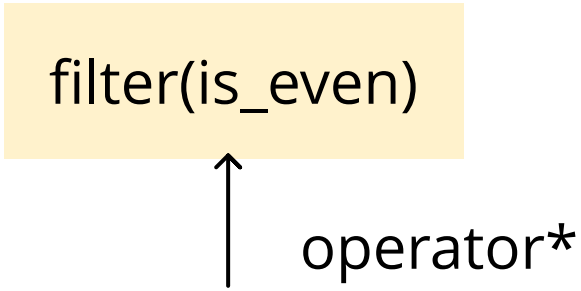
```
 1 void print_range(auto range) {
 2     for (const auto& elem : range) {
 3         cout << "number: " << elem << endl;
 4     }
 5 }
 6
 7 auto is_even = [](int elem) {
 8     cout << "filter: " << elem << endl;
 9     return elem % 2 == 0;
10 };
11
12 auto square = [](int elem) {
13     cout << "transform: " << elem << endl;
14     return elem * elem;
15 };
16
17 std::vector v{1, 2, 3, 4};
18
19 print_range(v
20             | rv::transform(square)
21             | rv::filter(is_even));
```

```
 1 transform: 1
 2 filter: 1
 3 transform: 2
 4 filter: 4
 5 transform: 2
 6 number: 4
 7 transform: 3
 8 filter: 9
 9 transform: 4
10 filter: 16
11 transform: 4
12 number: 16
```

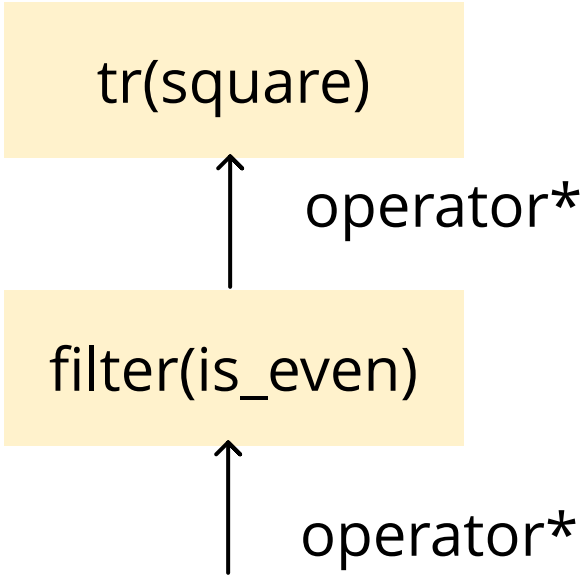
- А теперь?

Pull model

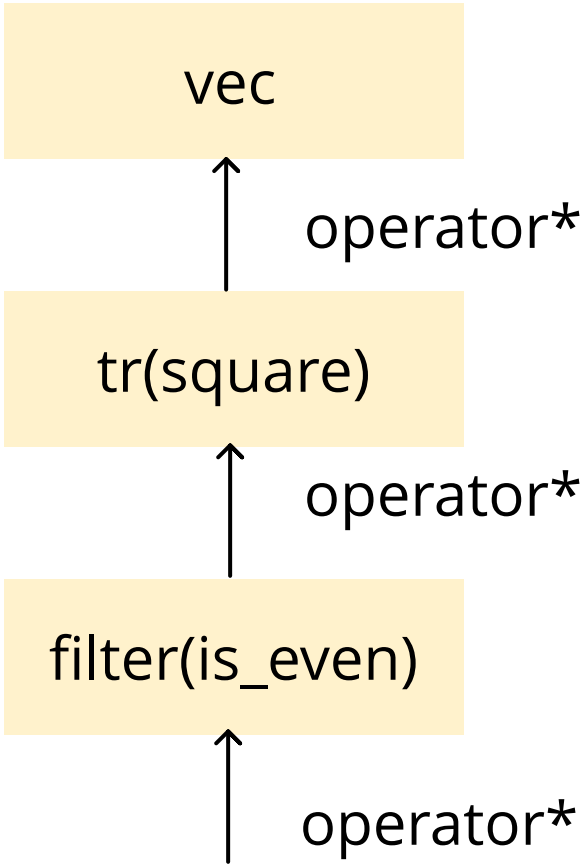
Pull model



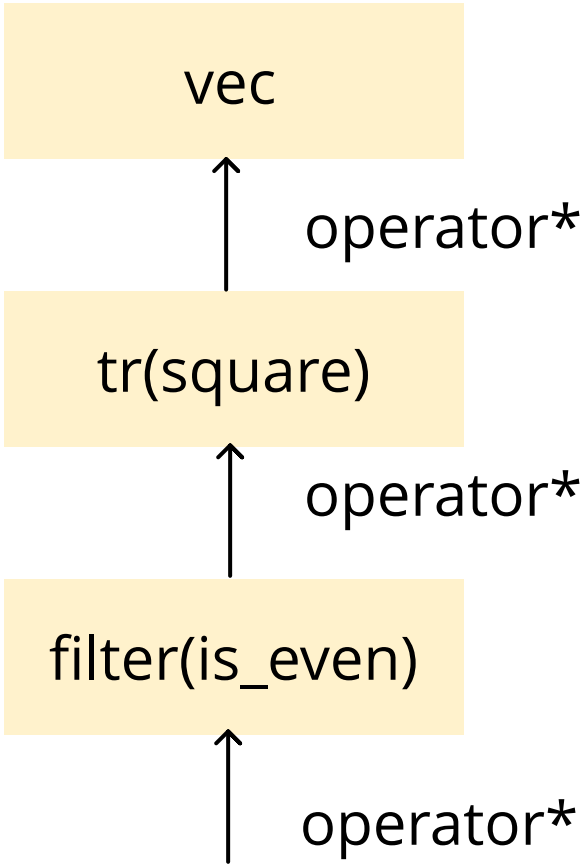
Pull model

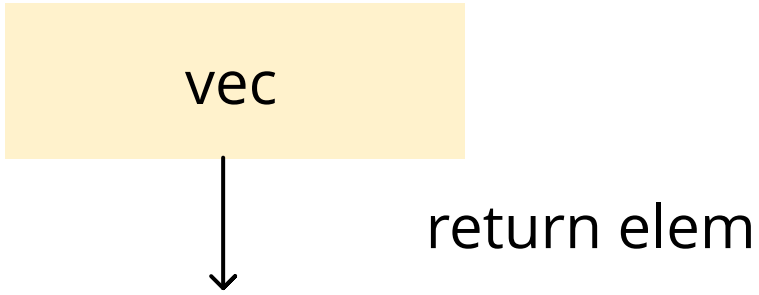


Pull model

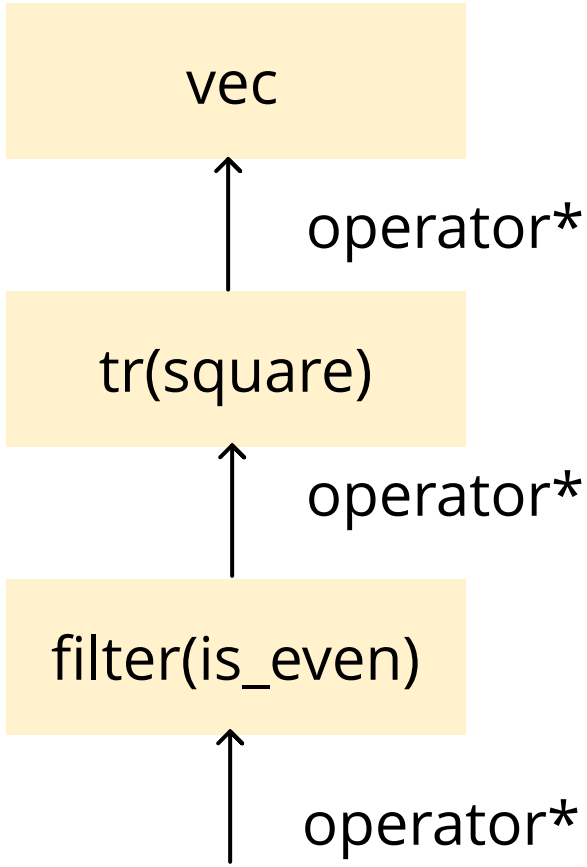


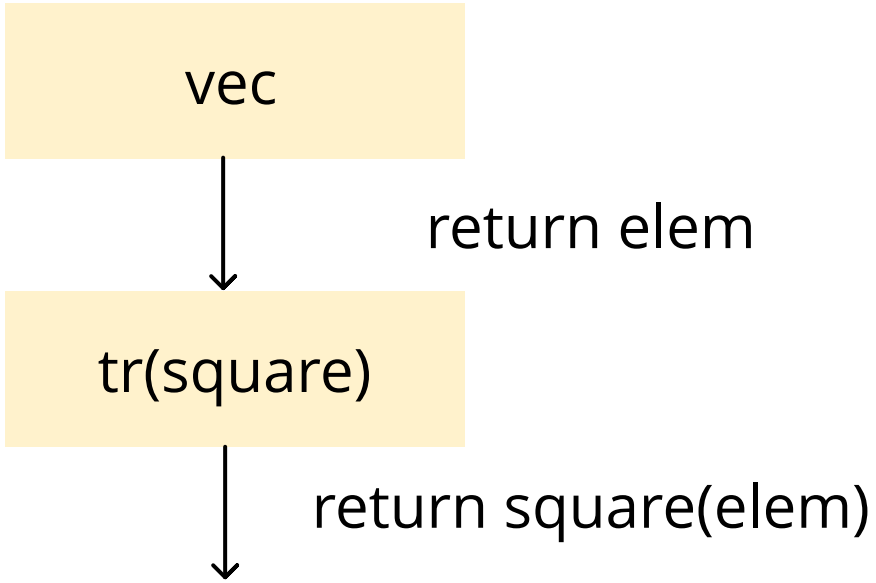
Pull model


vec
operator*
tr(square)
operator*
filter(is_even)
operator*

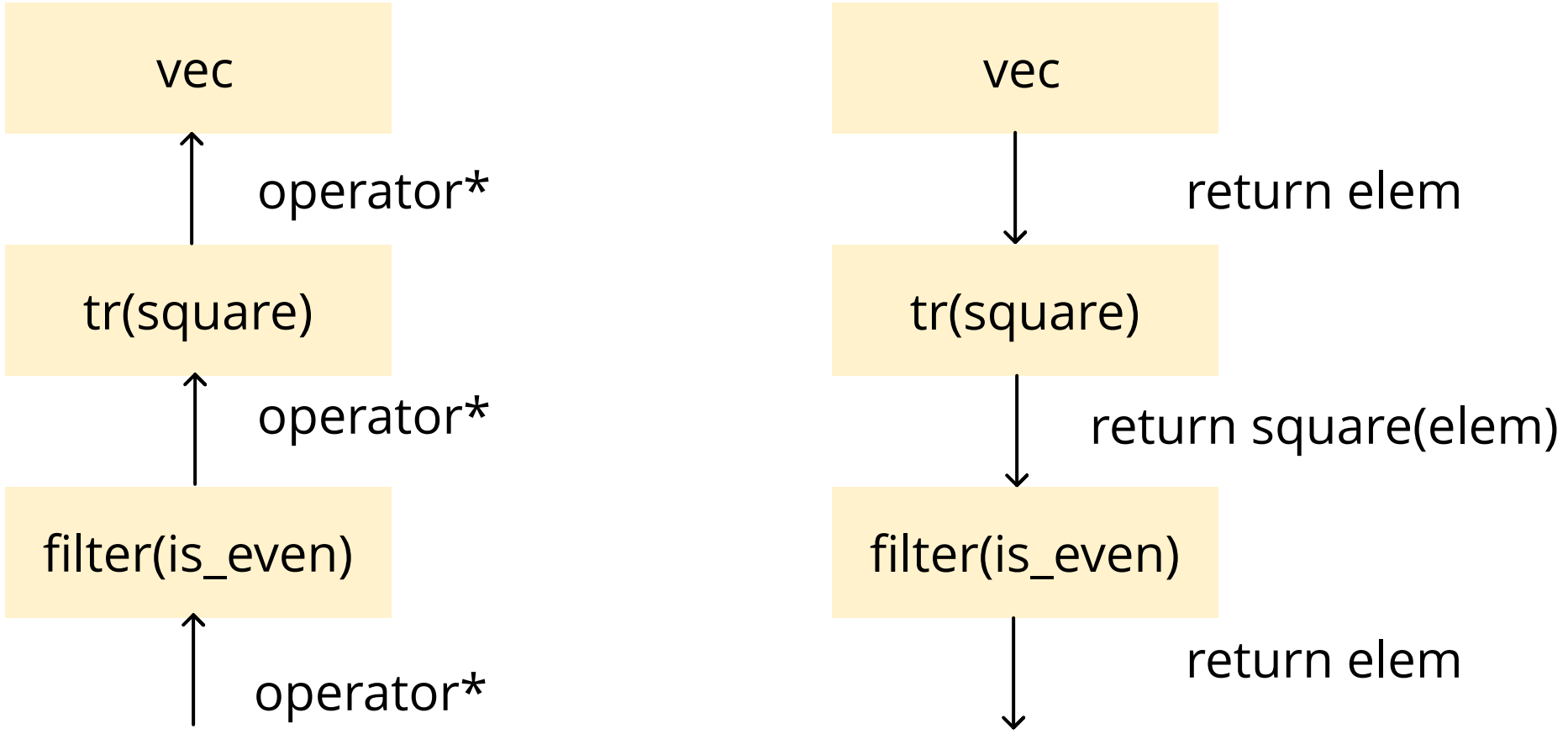

vec
return elem

Pull model


vec
operator*
tr(square)
operator*
filter(is_even)
operator*


vec
return elem
tr(square)
return square(elem)

Pull model


vec
operator*
tr(square)
operator*
filter(is_even)
operator*
vec
return elem
tr(square)
return square(elem)
filter(is_even)
return elem

Pull model

- transform вызывает унарную операцию на каждое разыменование итератора
- почему бы не кешировать значение?
- кеширование приводит к неожиданным эффектам

Sliding window и pull model

```
1 auto sliding_rng = v
2          | rv::transform(heavy_computation)
3          | rv::slide(n));
```

При итерации по скользящему окну
`heavy_computation` будет вызываться
`n` раз для большинства элементов

begin()

Что выведет этот код?

```
1 void print_range(const auto& range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 vector vec{1, 2, 3};
8 print_range(vec);
```

begin()

Что выведет этот код?

```
1 void print_range(const auto& range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 vector vec{1, 2, 3};
8 print_range(vec);
```

```
1 $: c++ example.cpp && ./a.out
2 >:  1 2 3
```

begin()

Что выведет этот код?

```
1 void print_range(const auto& range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 vector vec{1, 2, 3};
8 print_range(vec | rv::drop(1));
```

begin()

Что выведет этот код?

```
1 void print_range(const auto& range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 vector vec{1, 2, 3};
8 print_range(vec | rv::drop(1));
```

```
1 $: c++ example.cpp && ./a.out
2 >:  2 3
```

begin()

Что выведет этот код?

```
1 void print_range(const auto& range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 list l{1, 2, 3};
8 print_range(l);
```

begin()

Что выведет этот код?

```
1 void print_range(const auto& range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 list l{1, 2, 3};
8 print_range(l);
```

```
1 $: c++ example.cpp && ./a.out
2 >:  1 2 3
```

begin()

Что выведет этот код?

```
1 void print_range(const auto& range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 list l{1, 2, 3};
8 print_range(l | rv::drop(1));
```

begin()

Что выведет этот код?

```
1 void print_range(const auto& range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 list l{1, 2, 3};
8 print_range(l | rv::drop(1));
```

```
1 $: c++ example.cpp && ./a.out
2 >:  error: passing 'const
  std::ranges::drop_view<std::ranges::ref_view<std::
  __cxx11::list<int> > >' as 'this' argument
  discards qualifiers [-fpermissive]
```

begin()

```
1  constexpr auto begin() const
2  requires random_access_range<const View> && sized_range<const View> {
3      const auto dist = min(ranges::distance(base), count);
4      return ranges::begin(base) + dist;
5  }
6
7  constexpr auto begin() {
8    if constexpr (random_access_range<_View> && sized_range<_View>) {
9      const auto dist = std::min(ranges::distance(base), count);
10     return ranges::begin(base) + dist;
11   } else {
12     if (cached_begin.has_value())
13       return *cached_begin;
14
15     auto tmp = ranges::next(ranges::begin(base), count, ranges::end(base));
16     cached_begin.emplace(tmp);
17     return tmp;
18   }
19 }
20
```

begin()

```
1  constexpr auto begin() const
2  requires random_access_range<const View> && sized_range<const View> {
3      const auto dist = min(ranges::distance(base), count);
4      return ranges::begin(base) + dist;
5  }
6
7  constexpr auto begin() {
8    if constexpr (random_access_range<_View> && sized_range<_View>) {
9      const auto dist = std::min(ranges::distance(base), count);
10     return ranges::begin(base) + dist;
11   } else {
12     if (cached_begin.has_value())
13       return *cached_begin;
14
15     auto tmp = ranges::next(ranges::begin(base), count, ranges::end(base));
16     cached_begin.emplace(tmp);
17     return tmp;
18   }
19 }
20
```

begin()

```
constexpr auto begin() const
requires random_access_range<const View> && sized_range<const View> {
    const auto dist = min(ranges::distance(base), count);
    return ranges::begin(base) + dist;
}

constexpr auto begin() {
  if constexpr (random_access_range<_View> && sized_range<_View>) {
    const auto dist = std::min(ranges::distance(base), count);
    return ranges::begin(base) + dist;
  } else {
    if (cached_begin.has_value())
      return *cached_begin;

    auto tmp = ranges::next(ranges::begin(base), count, ranges::end(base));
    cached_begin.emplace(tmp);
    return tmp;
  }
}

```

begin()

```
constexpr auto begin() const
requires random_access_range<const View> && sized_range<const View> {
    const auto dist = min(ranges::distance(base), count);
    return ranges::begin(base) + dist;
}

constexpr auto begin() {
  if constexpr (random_access_range<_View> && sized_range<_View>) {
    const auto dist = std::min(ranges::distance(base), count);
    return ranges::begin(base) + dist;
  } else {
    if (cached_begin.has_value())
      return *cached_begin;

    auto tmp = ranges::next(ranges::begin(base), count, ranges::end(base));
    cached_begin.emplace(tmp);
    return tmp;
  }
}

```

begin()

```
constexpr auto begin() const
requires random_access_range<const View> && sized_range<const View> {
    const auto dist = min(ranges::distance(base), count);
    return ranges::begin(base) + dist;
}

constexpr auto begin() {
  if constexpr (random_access_range<_View> && sized_range<_View>) {
    const auto dist = std::min(ranges::distance(base), count);
    return ranges::begin(base) + dist;
  } else {
    if (cached_begin.has_value())
      return *cached_begin;

    auto tmp = ranges::next(ranges::begin(base), count, ranges::end(base));
    cached_begin.emplace(tmp);
    return tmp;
  }
}

```

begin()

```
1  constexpr auto begin() const
2  requires random_access_range<const View> && sized_range<const View> {
3      const auto dist = min(ranges::distance(base), count);
4      return ranges::begin(base) + dist;
5  }
6
7  constexpr auto begin() {
8    if constexpr (random_access_range<_View> && sized_range<_View>) {
9      const auto dist = std::min(ranges::distance(base), count);
10     return ranges::begin(base) + dist;
11   } else {
12     if (cached_begin.has_value())
13       return *cached_begin;
14
15     auto tmp = ranges::next(ranges::begin(base), count, ranges::end(base));
16     cached_begin.emplace(tmp);
17     return tmp;
18   }
19 }
20
```

Зачем тут cached_begin?

begin()

```
1  constexpr auto begin() const
2  requires random_access_range<const View> && sized_range<const View> {
3      const auto dist = min(ranges::distance(base), count);
4      return ranges::begin(base) + dist;
5  }
6
7  constexpr auto begin() {
8    if constexpr (random_access_range<_View> && sized_range<_View>) {
9      const auto dist = std::min(ranges::distance(base), count);
10     return ranges::begin(base) + dist;
11   } else {
12     if (cached_begin.has_value())
13       return *cached_begin;
14
15     auto tmp = ranges::next(ranges::begin(base), count, ranges::end(base));
16     cached_begin.emplace(tmp);
17     return tmp;
18   }
19 }
20
```

Зачем тут cached_begin?

- both ranges::begin(t) and ranges::end(t) are amortized constant time
- если begin() не работает за O(1), придется кешировать

Когда сложность `begin()` линейная?

- `filter`, `drop_while`
- `drop`, когда `base` не `random_access_range`
- ...

Что делать?

```
1 void print_range(auto&& range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
```

```
1 print_range(std::list{1, 2, 3})
2 std::list l{1, 2, 3};
3 print_range(l | rv::drop(1));
```

- Уберем константость

Кеширование и pull model

```
1     vector v = {1, 2, 3, 4};
2     auto filtered = v | filter(is_even); // 2, 4
3     for (auto e : filtered) {
4         cout << e << ' '; // 2 4
5     }
6
7     v[1] = 3;
8     for (auto e : filtered) {
9         cout << e << ' '; // 3 4
10    }
```

- изменение `v` и повторная итерация по отображению `filtered` это `undefined behavior`

Кеширование и pull model

```
1     vector v = {1, 2, 3, 4};
2     auto filtered = v | filter(is_even); // 2, 4
3     for (auto e : filtered) {
4         cout << e << ' '; // 2 4
5     }
6
7     v[1] = 3;
8     for (auto e : filtered) {
9         cout << e << ' '; // 3 4
10    }
```

- изменение `v` и повторная итерация по отображению `filtered` это `undefined behavior`

Кеширование и pull model

```
vector v = {1, 2, 3, 4};
auto filtered = v | filter(is_even); // 2, 4
for (auto e : filtered) {
    cout << e << ' '; // 2 4
}

v[1] = 3;
auto filtered_copy = filtered;
for (auto e : filtered_copy) {
    cout << e << ' '; // 4
}
```

Кеширование и pull model

```
1     vector v = {1, 2, 3, 4};
2     auto filtered = v | filter(is_even); // 2, 4
3     for (auto e : filtered) {
4         cout << e << ' '; // 2 4
5     }
6
7     v[1] = 3;
8     auto filtered_copy = filtered;
9     for (auto e : filtered_copy) {
10        cout << e << ' '; // 4
11    }
```

Кеширование и pull model

```
1     vector v = {1, 2, 3, 4};
2     auto filtered = v | filter(is_even); // 2, 4
3     for (auto e : filtered) {
4         cout << e << ' '; // 2 4
5     }
6
7     v[1] = 3;
8     auto filtered_copy = filtered;
9     for (auto e : filtered_copy) {
10        cout << e << ' '; // 4
11    }
```

- *non-propagating-cache*<T> ведет себя точно так же, как `optional<T>`, но при копировании и перемещении не сохраняет значение

Кеширование и pull model

```
1     vector v = {1, 2, 3, 4};
2     auto filtered = v | filter(is_even); // 2, 4
3     for (auto e : filtered) {
4         cout << e << ' '; // 2 4
5     }
6
7     v[1] = 3;
8     auto filtered_copy = filtered;
9     for (auto e : filtered_copy) {
10        cout << e << ' '; // 4
11    }
```

- *non-propagating-cache*<T> ведет себя точно так же, как `optional<T>`, но при копировании и перемещении не сохраняет значение
- после копирования отображения, `итерация` будет вызван корректной

Кеширование и pull model

```
1     vector v = {1, 2, 3, 4};
2     auto filtered = v | filter(is_even); // 2, 4
3     for (auto e : filtered) {
4         cout << e << ' '; // 2 4
5     }
6
7     v[1] = 3;
8     for (auto e : auto(filtered)) {
9         cout << e << ' '; // 4
10    }
```

- Еще удобнее с C++23

Кеширование и pull model

```
vector v = {1, 2, 3, 4};
auto filtered = v | filter(is_even); // 2, 4
for (auto e : filtered) {
    cout << e << ' '; // 2 4
}

v[1] = 3;
for (auto e : auto(filtered)) {
    cout << e << ' '; // 4
}
```

- Еще удобнее с C++23

Всегда принимаем `range` по значению

```
1 void print_range(auto range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
```

```
1 print_range(std::list{1, 2, 3})
2 std::list l{1, 2, 3};
3 print_range(l | rv::drop(1));
```

Всегда принимаем `range` по значению

```
1 void print_range(auto range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
```

```
1 print_range(std::list{1, 2, 3})
2 std::list l{1, 2, 3};
3 print_range(l | rv::drop(1));
```

Как избежать лишних копирований?

```
1 std::list l{1, 2, 3};
2 print_range(l);
```

Концепт `ranges::view`

```cpp
template<class T>
concept view = ranges::range<T> && std::movable<T> && ranges::enable_view<T>;

template<class T>
constexpr bool enable_view =
    std::derived_from<T, view_base> || /*is-derived-from-view-interface*/<T>;
```

- move - конструктор за O(1)
- copy - конструктор за O(1) (если есть)
- деструктор за O(1)*
- ...

Воспользуемся концептами и ranges::views::all

```
1 void print_range(std::ranges::view auto range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 std::list l{1, 2, 3};
8 print_range(rv::all(l));
```

- `decay_t<Rng>` для `view`
- `ref_view` для lvalue
- иначе `owning_view`

Воспользуемся концептами и ranges::views::all

```
1 void print_range(std::ranges::view auto range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 std::list l{1, 2, 3};
8 print_range(rv::all(l));
```

- `decay_t<Rng>` для `view`
- `ref_view` для lvalue
- иначе `owning_view`

Воспользуемся концептами и ranges::views::all

```
1 void print_range(std::ranges::view auto range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 std::list l{1, 2, 3};
8 print_range(rv::all(l));
```

- `decay_t<Rng>` для `view`
- `ref_view` для lvalue
- иначе `owning_view`

Воспользуемся концептами и ranges::views::all

```
1 void print_range(std::ranges::view auto range) {
2     for (const auto& elem : range) {
3         cout << elem << ' ';
4     }
5 }
6
7 std::list l{1, 2, 3};
8 print_range(rv::all(l));
```

- `decay_t<Rng>` для `view`
- `ref_view` для lvalue
- иначе `owning_view`

Подытожим

- Отображения иногда кешируют результат
- Изменять последовательность после вызова итерации опасно
- Копирование отображения сбрасывает кеш
- Чтобы не провоцировать неопределенное поведение, стоит сбрасывать кеш копированиями

На cppreference написано, что:

The ranges library is an extension and generalization of the algorithms and iterator libraries that makes them more powerful by making them **composable and less error-prone.**

А как быть с константностью?

```
1 void print_range(std::ranges::view auto range) {
2     for (auto& elem : range) {
3         cout << elem += 2 << ' ';
4     }
5 }
6
7 std::list l{1, 2, 3};
8 print_range(l | filter(is_even)); // 4
9 print_range(l | filter(is_even)); // 6
```

- На самом деле константость сломана
- Далее мы в этом убедимся

А как быть с константностью?

```
1 void print_range(std::ranges::view auto range) {
2     for (auto& elem : range) {
3         cout << elem += 2 << ' ';
4     }
5 }
6
7 std::list l{1, 2, 3};
8 print_range(l | filter(is_even)); // 4
9 print_range(l | filter(is_even)); // 6
```

- На самом деле константость сломана
- Далее мы в этом убедимся

А как быть с константностью?

```
1 void print_range(std::ranges::view auto range) {
2     for (auto& elem : range) {
3         cout << elem += 2 << ' ';
4     }
5 }
6
7 std::list l{1, 2, 3};
8 print_range(l | filter(is_even)); // 4
9 print_range(l | filter(is_even)); // 6
```

- На самом деле константость сломана
- Далее мы в этом убедимся

zip family

У нас есть:

- `zip`
- `zip_transform ~ zip | transform`
- `enumerate ~ zip(iota(0), rng)`
- `adjacent ~ zip(rng, rng | drop(1))`
- `adjacent_transform ~ adjacent | transform`

zip_view

```
 1      void foo(const auto& rng) {
 2          for (const auto& [a, b] : rng) {
 3              a = "1234";
 4              b = 1234;
 5          }
 6      }
 7
 8      vector<string> str{"1", "2", "3", "4"};
 9      vector ints{1, 2, 3, 4};
10
11      foo(zip(str, ints));
```

- Куда пропала константность?
- элементы `zip_view` имеют тип `tuple<string&, int&>`

zip_view

- наверное эта ошибка дизайна, давайте ее исправим
- теперь константность работает правильно
- `elem = ...` больше не компилируется,

indirectly_writable

```
1       vector vec = {5, 4, -3, 2, 1};
2
3       ranges::sort(vec | rv::transform([](int x) {return x * x;}));
```

- скомпилируется ли код?
- какой результат сортировки мы ожидаем?
- как бы мы проверяли `constraints`?

indirectly_writable

```
1     template <class Out, class T>
2     concept Writable = requires(Out& o, T&& t) {
3      *o = std::forward<T>(t);
4     }
```

- тогда `int() = int()`

indirectly_writable

```
template< class Out, class T >
    concept indirectly_writable =
        requires(Out&& o, T&& t) {
            *o = std::forward<T>(t);
            *std::forward<Out>(o) = std::forward<T>(t);
            const_cast<const std::iter_reference_t<Out>&&>(*o) = std::forward<T>(t);
            const_cast<const std::iter_reference_t<Out>&&>(*std::forward<Out>(o)) =
                std::forward<T>(t);
        };
```

- indirectly_writable iterator - итератор, `value_type` которого является proxy ссылкой
- iterator для zip_view является indirectly_writable

indirectly_writable

```
template< class Out, class T >
    concept indirectly_writable =
        requires(Out&& o, T&& t) {
            *o = std::forward<T>(t);
            *std::forward<Out>(o) = std::forward<T>(t);
            const_cast<const std::iter_reference_t<Out>&&>(*o) = std::forward<T>(t);
            const_cast<const std::iter_reference_t<Out>&&>(*std::forward<Out>(o)) =
                std::forward<T>(t);
        };
```

- indirectly_writable iterator - итератор, `value_type` которого является proxy ссылкой
- iterator для zip_view является indirectly_writable

indirectly_writable

```
1 vector v{5, 4, 3, 2, 1};
2 vector<string> s{"b", "d", "s", "q", "f"};
3
4 ranges::sort(rv::zip(v, s));
```

Благодаря indirectly_writable концепту мы можем сортировать склеенные последовательности

ranges::constant_range

```
1 template< class T >
2 concept constant_range =
3     ranges::input_range<T> &&
4     /*constant-iterator*/<ranges::iterator_t<T>>;
5
6 template< class T >
7 concept /*constant-iterator*/ =
8     std::input_iterator<T> &&
9     std::same_as<std::iter_const_reference_t<T>, std::iter_reference_t<T>>;
```

zip_view

```
1  template <class Rng>
2  requires(ranges::constant_range<Rng> && ranges::view<Rng>)
3  void print_range(Rng rng) {
4      ...
5  }
6
7  vector v{5, 4, 3, 2, 1};
8  vector<string> s{"b", "d", "s", "q", "f"};
9
10 print_range(rv::zip(v, s) | rv::as_const);
11 print_range(v| rv::as_const);
```

- Как гарантировать константность view?
- `std::ranges::constant_range`
- `views::as_const`
- Что-то напоминает?

zip_view

```cpp
template <class Rng>
requires(ranges::constant_range<Rng> && ranges::view<Rng>)
void print_range(Rng rng) {
    ...
}

vector v{5, 4, 3, 2, 1};
vector<string> s{"b", "d", "s", "q", "f"};

print_range(rv::zip(v, s) | rv::as_const);
print_range(v| rv::as_const);
```

- Как гарантировать константность view?
- `std::ranges::constant_range`
- `views::as_const`
- Что-то напоминает?

zip_view

```
1  template <class Rng>
2  requires(ranges::constant_range<Rng> && ranges::view<Rng>)
3  void print_range(Rng rng) {
4      ...
5  }
6
7  vector v{5, 4, 3, 2, 1};
8  vector<string> s{"b", "d", "s", "q", "f"};
9
10 print_range(rv::zip(v, s) | rv::as_const);
11 print_range(v| rv::as_const);
```

- Как гарантировать константность view?
- `std::ranges::constant_range`
- `views::as_const`
- Что-то напоминает?

zip_view

```cpp
template <class Rng>
requires(ranges::constant_range<Rng> && ranges::view<Rng>)
void print_range(Rng rng) {
    ...
}

vector v{5, 4, 3, 2, 1};
vector<string> s{"b", "d", "s", "q", "f"};

print_range(rv::zip(v, s) | rv::as_const);
print_range(v| rv::as_const);
```

- Как гарантировать константность view?
- `std::ranges::constant_range`
- `views::as_const`
- Что-то напоминает?

begin() и cbegin()

```
1 void foo(const auto it) {
2     *it = 10;
3 }
4
5 std::vector ints{9, 8, 7, 5, 5};
6 foo(ints.begin());
7 foo(ints.cbegin()); //Compilation error
```

Итераторы тоже являются proxy-объектами

Split и Lazy split

split_view

```
 1 constexpr iterator& operator++() {
 2     cur = next.begin();
 3     if (cur != ranges::end(parent->base)) {
 4       cur = next_.end();
 5       if (cur == ranges::end(parent->base)) {
 6         trailing_empty = true;
 7         next           = {cur, cur};
 8       } else {
 9         next = parent->find_next(cur);
10       }
11     } else {
12       trailing_empty = false;
13     }
14     return *this;
15   }
```

- оставляет гарантии родителя
- для получения пары `iterator-sentinel`, сразу считает `sentinel`
- когда это может быть проблемой?

split_view

```
constexpr iterator& operator++() {
    cur = next.begin();
    if (cur != ranges::end(parent->base)) {
      cur = next_.end();
      if (cur == ranges::end(parent->base)) {
        trailing_empty = true;
        next           = {cur, cur};
      } else {
        next = parent->find_next(cur);
      }
    } else {
      trailing_empty = false;
    }
    return *this;
  }
```

- оставляет гарантии родителя
- для получения пары `iterator-sentinel`, сразу считает `sentinel`
- когда это может быть проблемой?

split_view

```
1 read_from_socket(socket) | split(' ') | ...
2
3 generator(heavy_computation) | split(...) | ...
```

lazy_split_view

```
1  constexpr outer_iterator& operator++() {
2       ...
3        } else {
4          // General case for n-element pattern.
5          do {
6            const auto [b, p] = ranges::mismatch(current(), end, begin, pend);
7            if (p == pend) {
8              current() = b;
9              if (current() == end) {
10               trailing_empty = true;
11             }
12             break; // The pattern matched; skip it.
13           }
14         } while (++current() != end);
15       }
16
17       return *this;
18     }
```

- не дает гарантий `contigous range`
- работает противоположно split_view - проверяет следущий элемент на несовпадение с `delimiter`

lazy_split_view

```
constexpr outer_iterator& operator++() {
     ...
      } else {
        // General case for n-element pattern.
        do {
          const auto [b, p] = ranges::mismatch(current(), end, begin, pend);
          if (p == pend) {
            current() = b;
            if (current() == end) {
              trailing_empty = true;
            }
            break; // The pattern matched; skip it.
          }
        } while (++current() != end);
      }

      return *this;
    }
```

- не дает гарантий `contigous range`
- работает противоположно split_view - проверяет следущий элемент на несовпадение с `delimiter`

Split и Lazy Split

- lazy_split_view медленнее split_view
- разница зависит от того насколько часто встречается `delimiter`
- ориентировочно от 1.5 до 3 раз
- не используем lazy_split_view пока это не необходимо

join_view

```
1  void print(const auto& rng) {
2      for (const auto& e : rng) {
3          cout << e << endl;
4      }
5  }
6
7
8  vector<string> vec{"1", "2", "3"};
9
10 auto a = vec | rv::join;
11 print(a); //Ok
12 print(a); //Ok
13 auto func = [](auto s) { return s;};
14
15 auto b = vec | rv::transform(func) | rv::join
16 print(b); //Ok
17 print(b); // UB
```

- иногда кеширует временные объекты, и становится `input range`

join_view

```
1  //class iterator
2  constexpr void satisfy() {
3      ...
4        auto&& inner = [this]() -> auto&& {
5            return parent->inner.emplace_from([&]() -> decltype(auto) {
6              return *get_outer();
7            });
8        }();
9      ...
10 }
11
12 iterator& operator++() {
13     ...
14     if (inner_it == inner_end) {
15       satisfy();
16       ...
17     }
18     ...
19 }
```

- Чтобы join работал с `rvalue` последовательностями приходится кешировать текущую последовательность в поле `inner`
- инкремент итератора меняет состояние объекта `join`

join_view

```
1  //class iterator
2  constexpr void satisfy() {
3      ...
4        auto&& inner = [this]() -> auto&& {
5            return parent->inner.emplace_from([&]() -> decltype(auto) {
6              return *get_outer();
7            });
8        }();
9      ...
10 }
11
12 iterator& operator++() {
13     ...
14     if (inner_it == inner_end) {
15       satisfy();
16       ...
17     }
18     ...
19 }
```

- Чтобы join работал с `rvalue` последовательностями приходится кешировать текущую последовательность в поле `inner`
- инкремент итератора меняет состояние объекта `join`

join_view

```
1  //class iterator
2  constexpr void satisfy() {
3      ...
4        auto&& inner = [this]() -> auto&& {
5            return parent->inner.emplace_from([&]() -> decltype(auto) {
6              return *get_outer();
7            });
8        }();
9      ...
10 }
11
12 iterator& operator++() {
13     ...
14     if (inner_it == inner_end) {
15       satisfy();
16       ...
17     }
18     ...
19 }
```

- Чтобы join работал с `rvalue` последовательностями приходится кешировать текущую последовательность в поле `inner`
- инкремент итератора меняет состояние объекта `join`

std::generator

```cpp
template<typename T>
struct Tree {
    T value;
    Tree *left{}, *right{};
    std::generator<const T&> traverse_inorder() const {
        if (left)
            co_yield std::ranges::elements_of(left->traverse_inorder());
        co_yield value;
        if (right)
            co_yield std::ranges::elements_of(right->traverse_inorder());
    }
};

Tree<char> tree;

for (char x : tree->traverse_inorder())
    std::cout << x << ' ';
std::cout << '\n';
```

- генерация последовательности с помощью корутин
- вместо стека использует std::stack

std::generator

```cpp
template<typename T>
struct Tree {
    T value;
    Tree *left{}, *right{};
    std::generator<const T&> traverse_inorder() const {
        if (left)
            co_yield std::ranges::elements_of(left->traverse_inorder());
        co_yield value;
        if (right)
            co_yield std::ranges::elements_of(right->traverse_inorder());
    }
};

Tree<char> tree;

for (char x : tree->traverse_inorder())
    std::cout << x << ' ';
std::cout << '\n';
```

- генерация последовательности с помощью корутин
- вместо стека использует std::stack

std::generator

```cpp
template<typename T>
struct Tree {
    T value;
    Tree *left{}, *right{};
    std::generator<const T&> traverse_inorder() const {
        if (left)
            co_yield std::ranges::elements_of(left->traverse_inorder());
        co_yield value;
        if (right)
            co_yield std::ranges::elements_of(right->traverse_inorder());
    }
};

Tree<char> tree;

for (char x : tree->traverse_inorder())
    std::cout << x << ' ';
std::cout << '\n';
```

- генерация последовательности с помощью корутин
- вместо стека использует std::stack

std::generator

```cpp
template<typename T>
struct Tree {
    T value;
    Tree *left{}, *right{};
    std::generator<const T&> traverse_inorder() const {
        if (left)
            co_yield std::ranges::elements_of(left->traverse_inorder());
        co_yield value;
        if (right)
            co_yield std::ranges::elements_of(right->traverse_inorder());
    }
};

Tree<char> tree;

for (char x : tree->traverse_inorder())
    std::cout << x << ' ';
std::cout << '\n';
```

- генерация последовательности с помощью корутин
- вместо стека использует std::stack

std::generator

```
 1 template<typename T>
 2 struct Tree {
 3     T value;
 4     Tree *left{}, *right{};
 5     std::generator<const T&> traverse_inorder() const {
 6         if (left)
 7             co_yield std::ranges::elements_of(left->traverse_inorder());
 8         co_yield value;
 9         if (right)
10             co_yield std::ranges::elements_of(right->traverse_inorder());
11     }
12 };
13
14 Tree<char> tree;
15
16 for (char x : tree->traverse_inorder())
17     std::cout << x << ' ';
18 std::cout << '\n';
```

- генерация последовательности с помощью корутин
- вместо стека использует std::stack

Выводы

- Для отображений не работает const-семантика и reference-семантика
- Отображения это zero-cost abstraction
- Если знать про подводные камни, они действительно позволяют уменьшить вероятность ошибок
- Отображения могут внезапно поменять гарантии при композиции
- Важно активно пользоваться концептами

Вопросы