MANCHESTER 1824
The University of Manchester

COMP26020 Programming Languages and Paradigms

Lecture 39: Range Views

Pavlos Petoumenos
pavlos.petoumenos@manchester.ac.uk

Too hot for most compilers

Added only a couple of years ago

Mostly supported for a year or so

```
g++-11 -std=c++20
```

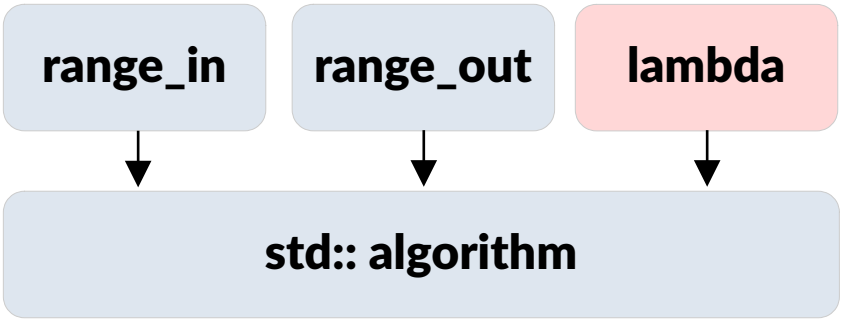
Range Views

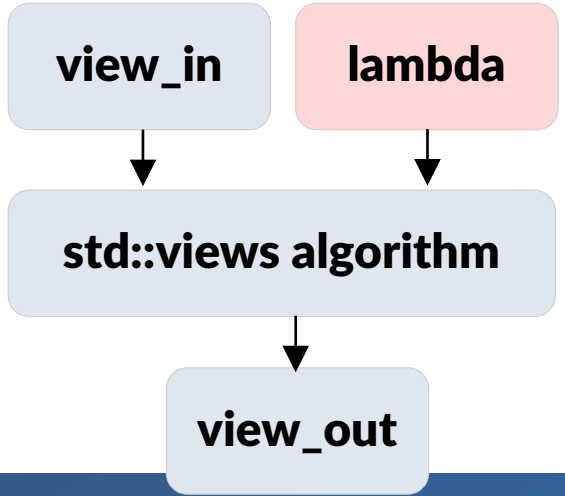
Lightweight objects that indirectly represent ranges

- Actual range
- Recipe for constructing a future range → lazy evaluation!

Range View Algorithms (std::views::)

- Like Algorithms but for Range Views





Range Views


```
std::vector<int> v(10);
std::vector<int> s(10);
…
// Range Algorithm → s = v^2
std::ranges::transform(v, s.begin(), [](int x) {return x*x;});


// Range View Algorithm → Create a view for v^2
auto view = std::views::transform(v, [](int x) {return x*x;});
// Evaluate the view step-by-step here:
std::ranges::copy(view, s.begin());
```

Meh

Range Algorithms

```
// Find the first 8 Mersenne (2^n - 1) primes

std::vector<int> output;
std::vector<int> v(63); // Is n <= 63 enough?
std::ranges::iota(v, 1);  // 1, 2, …, 63

// Get 2^n - 1 for all 63 numbers, v = 2^v - 1
std::ranges::transform(v, v.begin(), [](int x) {return (1<<x) - 1;});

// Back insert copies of prime numbers in output
std::ranges::copy_if(v, std::back_inserter(output), is_prime);

// Keep the first 8 numbers
output.resize(8);
```

Range View Algorithms

```
// Find the first 8 Mersenne (2^n - 1) primes

// View of all natural numbers
std::views::iota v(1); // 1, 2, 3, 4, 5 ...

// View for an infinite sequence of 2^n - 1 numbers
auto v2 = std::views::transform(v, [](int x) {return (1<<x) - 1;});

// View of an infinite sequence of Mersenne Primes
auto v3 = std::views::filter(v2, is_prime);

// View of the first 8 Mersenne Primes
auto v4 = std::views::take(v3, 8);

// Evaluate here and construct the output vector
std::vector<int> output(v4.begin(), v4.end());
```

Range View Algorithms

```
// Find the first 8 Mersenne (2^n - 1) primes

auto v = std::views::iota(1) |
         std::views::transform([](int x) {return (1<<x) - 1;}) |
         std::views::filter(is_prime) |
         std::views::take(8);

// Evaluate here and construct the output vector
std::vector<int> output(v.begin(), v.end());
```

Range Views

```
auto v = std::views::iota(1) |
         std::views::transform([](int x) {return (1<<x) - 1;}) |
         std::views::filter(is_prime) |
         std::views::take(8);
std::vector<int> output(v.begin(), v.end());
```

More concise and clear

Less space (no input vector)

Only needed calculations

2x faster!

Ranges

```
std::vector<int> output;
std::vector<int> v(63);
std::ranges::iota(v, 1);
std::ranges::transform(v, v.begin(), [](int x) {return (1<<x)-1;});
std::ranges::copy_if(v, std::back_inserter(output), is_prime);
output.resize(8);
```

View-based algorithms

```
auto v = std::views::iota(1) |
         std::views::transform([](int x) {return (1<<x) - 1;}) |
         std::views::filter(is_prime) |
         std::views::take(8);
std::vector<int> output(v.begin(), v.end());
```

Composition of functions vs sequence of steps

Algorithmic intention vs Algorithm

Functional vs Imperative

Recap

- Range Views
  - Recipes for creating ranges
  - Lazy evaluation!
- View-based algorithms
  - Return the recipe for creating the result
  - Can chain them together
  - Concise, clear, efficient

Up Next

New/delete considered harmful (revisited)