TDDD38/726G82:
Adv. Programming in C++
Standard Library III

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1 Algorithms
2 Ranges

1 Algorithms
2 Ranges

Algorithms

Core of STL

- Everything in STL is based around algorithms and containers
- There are 110+ algorithms defined in the STL (exact amount depends on version)
- All algorithms operate on *iterator ranges*
- Uses lambdas and function objects heavily
- Defined (mostly) in `<algorithm>`

Algorithms

Algorithm categories

- Non-modifying sequence operations
- Modifying sequence operations
- Partitioning operations
- Sorting operations
- Sorted range operations
- Set operations

Algorithms

Algorithm categories

- Heap operations
- Minmax operations
- Comparison operations
- Permutation operations
- Numeric operations (`<numeric>`)
- Uninitialized operations (`<memory>`)

Algorithms

`for_each`; the dull one of the bunch!

```
1  void put(int n)
2  {
3    cout << n << endl;
4  }
5
6  int main()
7  {
8    std::vector<int> v { 1, 2, 3 };
9    std::for_each(std::begin(v), std::end(v), put);
10 }
```

Algorithms

Possible implementation

```
1  template <typename It, typename Function>
2  Function for_each(It first, It last, Function f)
3  {
4    while (first != last)
5    {
6      f(*first++);
7    }
8    return f;
9  }
```

Algorithms

What will be printed?

```
1 std::vector<int> v {1, 2, 3};
2 auto f {[x = 0](int n) mutable
3         {
4           x += n;
5           return x;
6         }};
7 std::for_each(std::begin(v), std::end(v), f);
8 cout << f(0) << endl;
```

Algorithms

Answer

- `0` will be printed
- `x` is an internal variable accessible inside the lambda which will retain its value through each successive call to `f`
- however; `std::for_each` takes `f` as a copy, so `f` will not have been called when we reach the print statement

Algorithms

Possible fix

```
1  std::vector<int> v {1, 2, 3};
2  auto f {[x = 0](int n) mutable
3          {
4            x += n;
5            return x;
6          }};
7  std::for_each(std::begin(v), std::end(v),
8                std::ref(f));
9  cout << f(0) << endl;
```

Algorithms

`std::ref`

- `std::ref(x)` forces the compiler to interpret `x` as an lvalue
- thus forcing any template-parameters to deduce it as a reference parameter rather than a by-value parameter

Algorithms

Another possible fix

```
1 std::vector<int> v {1, 2, 3};
2 auto f {[x = 0](int n) mutable
3         {
4           x += n;
5           return x;
6         }};
7 f = std::for_each(std::begin(v), std::end(v), f);
8 cout << f(0) << endl;
```

Algorithms

`std::find`

```
1  std::vector<int> v {5, -2, 8, 4, 7};
2  auto it {
3    std::find(std::begin(v), std::end(v), 8)
4  };
5  if (it == std::end(v))
6  {
7    // we didn't find it :(
8  }
9  else
10 {
11   cout << *it << endl;
12 }
```

Algorithms

Predicate algorithms

```
1  std::set<int> m { -1, 4, 0, 3 };
2  auto p {[](auto a)
3          {
4            return a >= 0;
5          }};
6  if (std::all_of(std::begin(m), std::end(m), p))
7  {
8    // all of the numbers are positive
9  }
```

Algorithms

Predicate algorithms

```
1  template <typename It,
2            typename Predicate>
3  bool all_of(It first, It last, Predicate p)
4  {
5    while (first != last)
6    {
7      if (!p(*first++))
8      {
9        return false;
10     }
11   }
12   return true;
13 }
```

Algorithms

`std::copy`

```
1 std::vector<int> v {1, 2, 3};
2 std::set<int> s {};
3 std::copy(std::begin(v), std::end(v),
4           std::inserter(s, std::end(s)));
```

Algorithms

A cool usage of `std::copy` and stream iterators

```
1 std::vector<int> v{1, 2, 3};
2 std::copy(std::begin(v), std::end(v),
3           std::ostream_iterator<int>{cout, " "});
```

Algorithms

Another cool usage of `std::copy`

```
1 std::vector<int> v;
2 auto begin{std::istream_iterator<int>{cin}};
3 auto end{std::istream_iterator<int>{}};
4 std::copy(begin, end, std::back_inserter(v));
```

Algorithms

`std::transform`

```
1  std::vector<std::string> v{"1", "2", "3"};
2  std::vector<int> target{};
3
4  std::transform(std::begin(v), std::end(v),
5                 std::back_inserter(target),
6                 [](std::string const& s)
7                 {
8                   return std::stoi(s);
9                 });
```

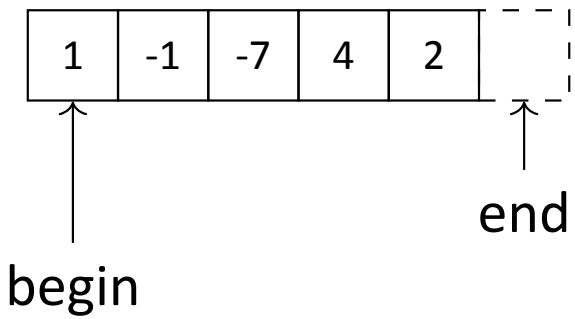
Algorithms

`std::remove_if`

```
1  std::vector<int> v{1, -1, -7, 4, 2};
2  v.erase(
3      std::remove_if(std::begin(v), std::end(v),
4                     [](int n)
5                     {
6                       return n < 0;
7                     }), std::end(v));
```

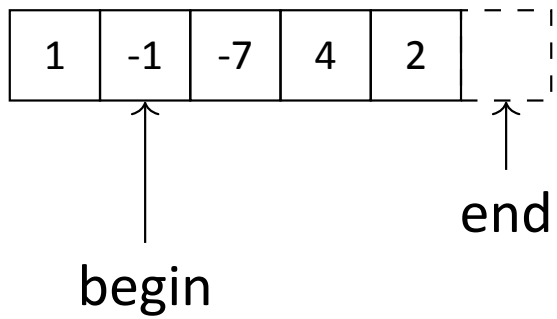
Algorithms

`std::remove_if`



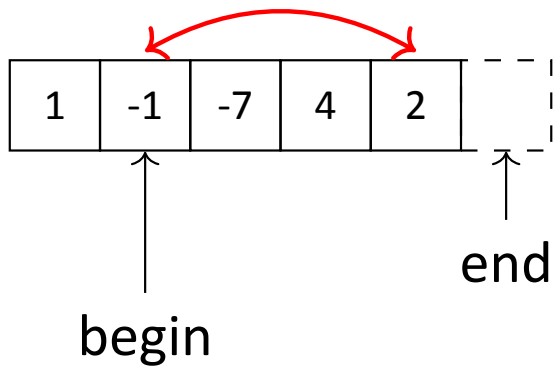
Algorithms

`std::remove_if`



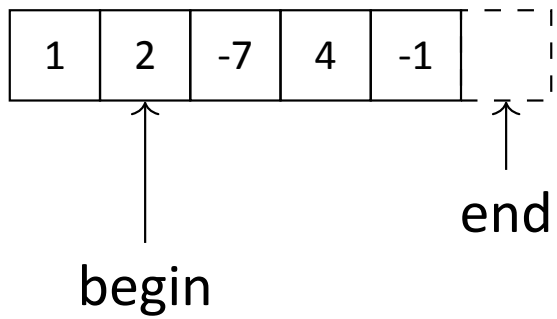
Algorithms

`std::remove_if`



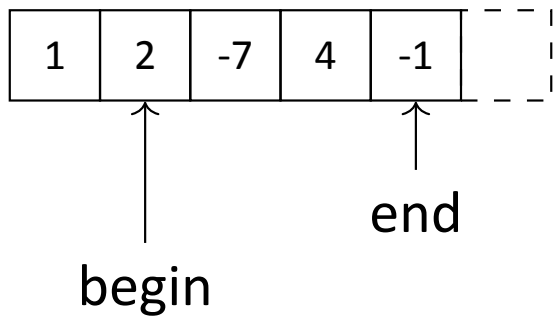
Algorithms

`std::remove_if`



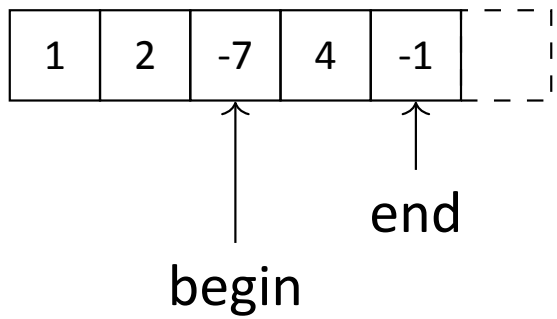
Algorithms

`std::remove_if`



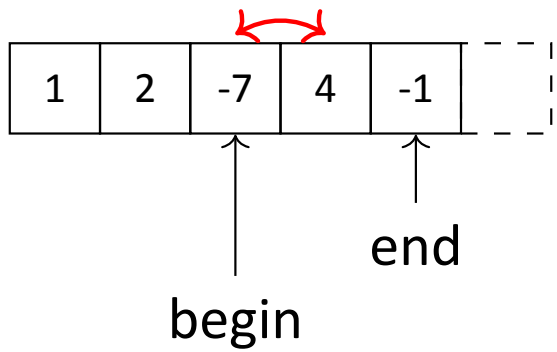
Algorithms

`std::remove_if`



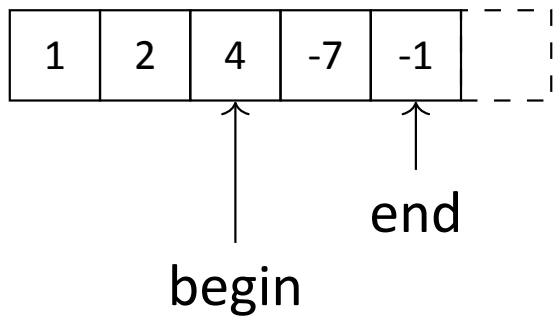
Algorithms

`std::remove_if`



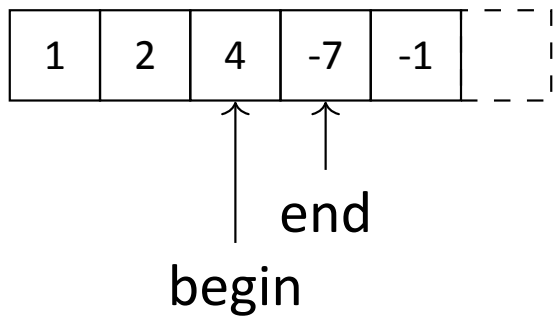
Algorithms

`std::remove_if`



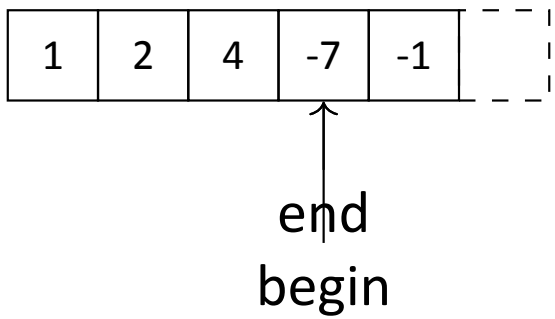
Algorithms

`std::remove_if`



Algorithms

`std::remove_if`



Algorithms

A note on modifying sequence operations

- algorithms operate on iterator ranges
- it is not possible to remove or add elements through arbitrary iterators
- due to this, no algorithm is able to remove elements from containers
- (they are able to add elements with the use of output iterators)

Algorithms

Modifying sequence operations

- All algorithms that are meant to remove elements will instead move them to the end of the range and return an iterator to the first removed element
- with that iterator we can now call the `erase` function of the underlying container to actually remove those elements

Algorithms

`std::accumulate`

```
1  #include <numeric>
2  // ...
3  int main()
4  {
5    std::vector<int> v{1, 2, 3, 4, 5};
6    int sum{
7      std::accumulate(std::begin(v), std::end(v), 0)
8    };
9    cout << sum << endl; // will print 15
10 }
```

Algorithms

`std::accumulate`

```
1 std::set<std::string> v{"1", "2", "3"};
2 int result{
3   std::accumulate(std::begin(v), std::end(v), 4,
4                   [](int n, std::string const& s)
5                   {
6                     return n + std::stoi(s);
7                   })
8 };
```

Algorithms

`std::accumulate`

- `std::accumulate` is like a fold-expression
- but it operates during runtime on iterator ranges
- very flexible when combining values into a single value

Algorithms

Final words

- There are a lot of algorithms
- You are not expected to memorize them all
- However you must be able to find suitable algorithms and use them

Algorithms

Now go forth and use this great power!

1 Algorithms
2 Ranges

Ranges

Example

```
1  using namespace std;
2
3  vector<int> v { 3, 1, -5, 4 };
4  sort(begin(v), end(v));
5  copy(begin(v), end(v),
6       ostream_iterator<int>{cout, " "});
```

Ranges

Example

- In this example we sort a vector of integers and print the result to `std::cout`.
- This is quite a good usage of the STL.
- But, it is quite annoying having to write `begin(v)` and `end(v)` whenever we want to do something with the entire vector `v`.

Ranges

Same example in C++ 20

```
1  using namespace std;
2
3  vector<int> v { 3, 1, -5, 4 };
4  ranges::sort(v);
5  ranges::copy(v, ostream_iterator<int>{cout, " "});
```

Ranges

Same example in C++ 20

- C++ 20 introduces a new concept called *ranges*.
- This allows us to call algorithms with entire containers instead of having to pass iterators everywhere.
- *ranges* are not supposed to replace iterators, but will instead complement them.

Ranges

What are ranges?

```
1 template <typename T>
2 concept range = requires(T& t)
3 {
4   std::begin(t);
5   std::end  (t);
6 };
```

Ranges

What are ranges?

- A *range* is a type which we can call `std::begin` and `std::end` on. This means that we can think of ranges as objects that have an iterator pair. This means that for example all *containers* are ranges.
- All (most) algorithms now has an *iterator* version and a *range* version.
- So what's the big deal? Sure we get a bit cleaner syntax, but besides that?

Ranges

Let’s look at an example from before C++ 20

```
1  struct Point
2  {
3    int x;
4    int y;
5  };
6
7  int main()
8  {
9    std::vector<Point> points { /* ... */ };
10
11   auto filter = [](Point p) { return p.x < 0 || p.y < 0; };
12
13   points.erase(
14     std::remove_if(std::begin(points), std::end(points), filter),
15     std::end(points));
16
17   auto printer = [](Point p) { cout << p.x << ", " << p.y << endl; };
18
19   std::for_each(std::begin(points), std::end(points), printer);
20 }
```

Ranges

Let's look at an example

- We remove all points were either `x` or `y` are negative and then print the remaining points. However, this could be done better for several reasons.
- For starters, it is very annoying having to write `std::begin(v)` and `std::end(v)` everytime we want to operate on the vector.
- But there is also a performance issue here: we have to remove elements from the container, which is quite slow. For this example wouldn't it be enough to just *not* print the negative values? No need for actual removal.

Ranges

Enter range adaptors!

```
1  struct Point
2  {
3    int x;
4    int y;
5  };
6
7  using namespace std;
8
9  int main()
10 {
11   vector<Point> const points { /* ... */ };
12
13   auto filter = [](Point p) { return p.x >= 0 && p.y >= 0; };
14   auto result = points | ranges::views::filter(filter);
15   for (Point p : result)
16   {
17     cout << p.x << ", " << p.y << endl;
18   }
19
20 }
```

Ranges

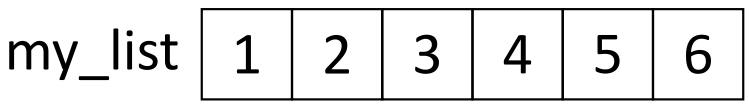
Let’s break it down

- **views**
- range adaptors

Ranges

Views

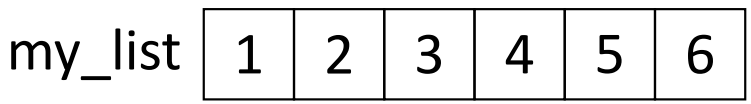
```
1
2 list<int> my_list { 1, 2, 3, 4, 5, 6 };
```



Ranges

Views

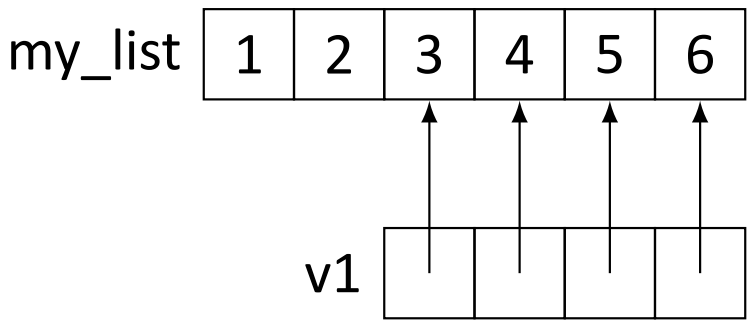
```
1
2 auto v1 = ranges::views::drop(my_list, 2);
```



Ranges

Views

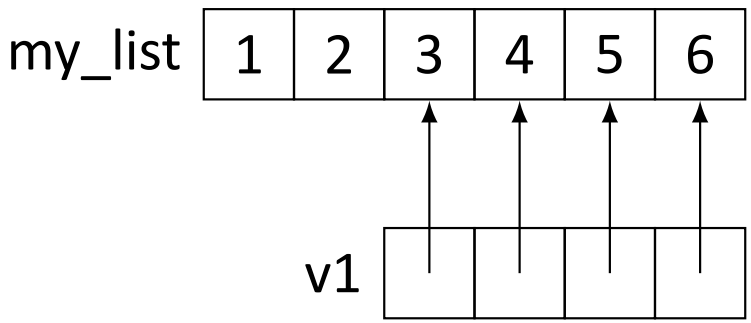
```
1
2 auto v1 = ranges::views::drop(my_list, 2);
```



Ranges

Views

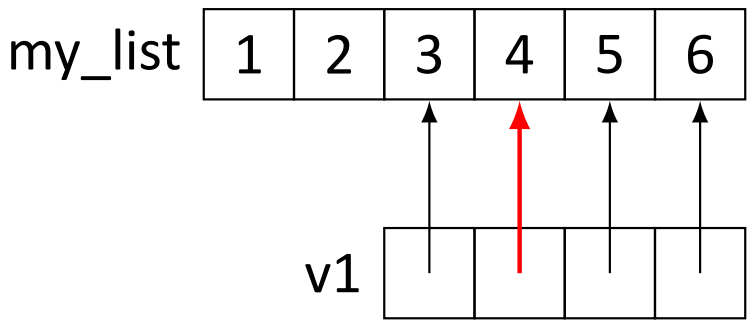
```
1 auto f = [](int x) { return x % 2 == 0; };
2 auto v2 = ranges::views::filter(v1, f);
```



Ranges

Views

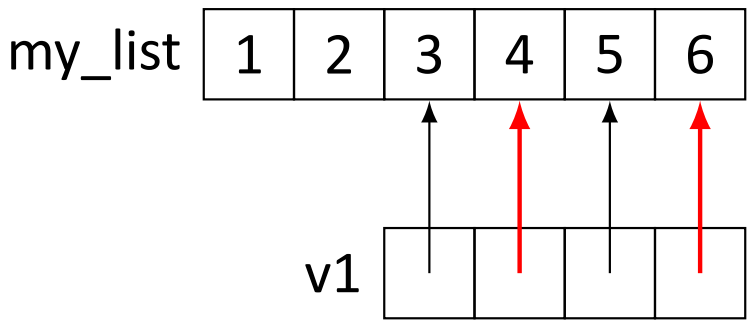
```
1 auto f = [](int x) { return x % 2 == 0; };
2 auto v2 = ranges::views::filter(v1, f);
```



Ranges

Views

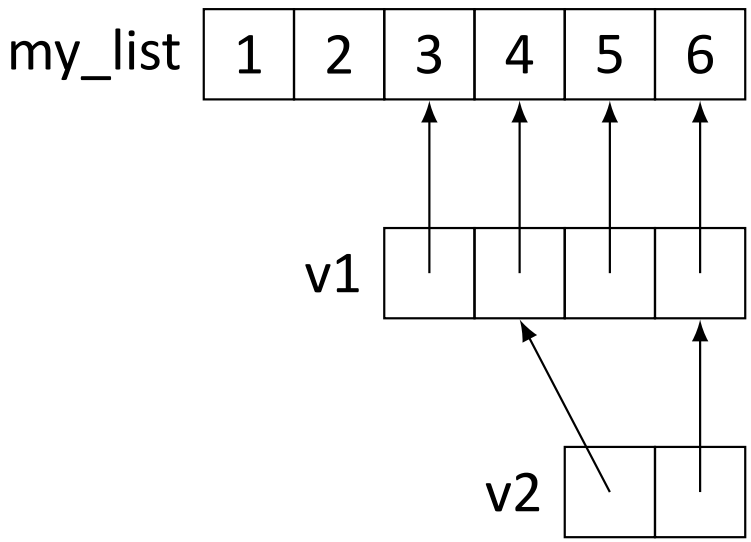
```
1 auto f = [](int x) { return x % 2 == 0; };
2 auto v2 = ranges::views::filter(v1, f);
```



Ranges

Views

```
1 auto f = [](int x) { return x % 2 == 0; };
2 auto v2 = ranges::views::filter(v1, f);
```



Ranges

What?

- A *view* is a window into a subset of the elements in some range. It allows us to construct subranges procedurally.
- It's important to note that a view is much more powerful than just a pair of iterators. As we saw with `views::filter`, it can produce gaps from the original range.
- **Note:** a view never copies any elements, nor does it keep track of which elements it is pointing to (as the image might suggest). Instead, a view is computed lazily.

Ranges

Lazy computation of views

```
1 auto v1 = std::ranges::views::iota(1);
2 for (int i : v1)
3 {
4   cout << i << endl;
5 }
```

Ranges

Lazy computation of views

```
1  auto v1 = std::ranges::views::iota(1);
2  for (int i : v1)
3  {
4    cout << i << endl;
5  }
```

Infinite loop!

Ranges

Lazy computation of views

- `std::ranges::views::iota` is a view that *lazily* generates a sequence of numbers. In this case it will generate the numbers: `1, 2, 3, ...` and so on *forever*.
- This implies that views doesn't construct all its elements in memory, since `iota` will literally generate numbers forever!
- This means that it *must* generate each number as it is requested. This is what's meant when we say it is *lazy*.

Ranges

Lazy computation of views

```
1  auto even = [](int x) { return x % 2 == 0; };
2
3  auto v1 = std::ranges::views::iota(1);
4  auto v2 = std::ranges::views::filter(v1, even);
5  for (int i : v2)
6  {
7    cout << i << endl;
8  }
```

Ranges

Lazy computation of views

```
1  auto even = [](int x) { return x % 2 == 0; };
2
3  auto v1 = std::ranges::views::iota(1);
4  auto v2 = std::ranges::views::filter(v1, even);
5  for (int i : v2)
6  {
7    cout << i << endl;
8  }
```

Infinite loop!

Ranges

Lazy computation of views

- `v2` is also lazily evaluated, but this time we will filter out all values that are *not* even.
- This means that `v2` will generate the numbers `2, 4, 6, ...` and so on.
- What *filter* does whenever we request a number from it, is that it will request a number `n` from `v1`. If `even(n)` returns `false` it will request a new number and repeat the process until it retrieves a number for which `even` returns `true`. That number will then be handed to the caller.

Ranges

Lazy computation of views

```
1  auto even = [](int x) { return x % 2 == 0; };
2
3  auto v1 = std::ranges::views::iota(1);
4  auto v2 = std::ranges::views::filter(v1, even);
5  auto v3 = std::ranges::views::take(v2, 4);
6  for (int i : v3)
7  {
8    cout << i << endl;
9  }
```

Ranges

Lazy computation of views

```
1  auto even = [](int x) { return x % 2 == 0; };
2
3  auto v1 = std::ranges::views::iota(1);
4  auto v2 = std::ranges::views::filter(v1, even);
5  auto v3 = std::ranges::views::take(v2, 4);
6  for (int i : v3)
7  {
8    cout << i << endl;
9  }
```

Halts!

Ranges

Lazy computation of views

- Finally, if we apply `take`, we will get a terminating loop.
- This is because `take(v2, 4)` will not deliver any more numbers after the user has requested 4 numbers.
- It is still done lazily though. So it doesn't know what the next number will be, all it knows is how many have been requested so far.
- In this case it will print: `2, 4, 6, 8` and then terminate.

Ranges

Let’s break it down

- views
- **range adaptors**

Ranges

Range adaptors!

```
1  using namespace std::ranges;
2
3  auto even = [](int x) { return x % 2 == 0; };
4  auto inc  = [](int x) { return x + 1; };
5
6  auto v1 = views::iota(1);
7  auto v2 = views::filter(v1, even);
8  auto v3 = views::transform(v2, inc);
9  auto v4 = views::take(v3, 10);
10 for (int i : v4)
11 {
12   cout << i << endl;
13 }
```

Ranges

Range adaptors!

```
1  using namespace std::ranges;
2
3  auto even = [](int x) { return x % 2 == 0; };
4  auto inc  = [](int x) { return x + 1; };
5
6  auto v = views::iota(1)
7           | views::filter(even)
8           | views::transform(inc)
9           | views::take(10);
10 for (int i : v)
11 {
12   cout << i << endl;
13 }
```

Ranges

Range adaptors!

- We've actually already seen a few *range adaptors*, for example `views::filter`, `views::take` and `views::drop`.
- A *range adaptor* takes a range and applies some kind of operation on it. For example: `views::transform` takes a range and applies a callable object on each element, thus *transforming* each requested value into a new value.

Ranges

Range adaptors!

- There are two ways to apply a range adaptor C to a range called `r`:
- The way we've already seen:
  `C(r, parameters...)`
- Or with `operator|` like this:
  `r | C(parameters...)`
- The advantage of `operator|` is that we can now chain the range adaptors together without having to create intermediate views. Like in the second example above.

Ranges

Let’s not forget that containers are ranges too!

```
1  using namespace std;
2  using namespace std::ranges;
3
4  map<string, int> m { /* ... */ };
5
6  auto fun = [](pair<string, int> p)
7             {
8               return p.first + ": " + to_string(p.second);
9             };
10
11 ranges::copy(m | views::transform(fun) | views::reverse,
12      ostream_iterator<string>{cout, "\n"});
13
14 vector<pair<string, int>> v { begin(m), end(m) };
15
16 // with lambda
17 ranges::sort(v, [](auto a, auto b) { return a.second < b.second; });
18
19 ranges::copy(v | views::transform(fun),
20              ostream_iterator<string>{cout, "\n"});
```

Ranges

Let’s not forget that containers are ranges too!

```
1  using namespace std;
2  using namespace std::ranges;
3
4  map<string, int> m { /* ... */ };
5
6  auto fun = [](pair<string, int> p)
7             {
8               return p.first + ": " + to_string(p.second);
9             };
10
11 ranges::copy(m | views::transform(fun) | views::reverse,
12      ostream_iterator<string>{cout, "\n"});
13
14 vector<pair<string, int>> v { begin(m), end(m) };
15
16 // with std::less and projection
17 ranges::sort(v, less<int>{}, &pair<string, int>::second);
18
19 ranges::copy(v | views::transform(fun),
20              ostream_iterator<string>{cout, "\n"});
```

Ranges

Now we have the tools to understand the previous example!

```
1  struct Point
2  {
3    int x;
4    int y;
5  };
6
7  using namespace std;
8
9  int main()
10 {
11   vector<Point> const points { /* ... */ };
12
13   auto filter = [](Point p) { return p.x >= 0 && p.y >= 0; };
14   auto result = points | ranges::views::filter(filter);
15   for (Point p : result)
16   {
17     cout << p.x << ", " << p.y << endl;
18   }
19
20 }
```

Ranges

Projections

- Some range algorithms, like `ranges::sort`, takes an optional *projection* parameter.
- A projection is pointer to some data member in the passed in class which is used for comparison.
- So instead of creating a lambda which compares based on the entire object, this allows us to use a comparison object that operates on that specific data member directly.

www.liu.se

LINKÖPING UNIVERSITY