A Tour of Ranges in C++

Tom Peters, June 18, 2019

thomas.d.peters@gmail.com

tdp2110 on slack, github

@__tom_peters__

Outline

1. Introduction to ranges
2. Views
3. Actions
4. Algorithms
5. The dark side

range-v3

https://github.com/ericniebler/range-v3/

Work of Eric Niebler

- C++ Standards committee member
- Author of 4 boost libraries

What is a range?

What is a range? (Version 1)

A *range* is something that can be put in a range-based for-loop:

```
for (auto&& elt : rng) {
  // do stuff with elt
}
```

Examples:

- `std::vector`
- `std::array`
- `std::set`
- `std::string_view`

What is a range (Version 2)

A *range* is something we can call std::begin and std::end, which return *iterators*.

```
for (auto it = std::begin(rng); it != std::end(rng); ++it) {
  // do stuff with it
}
```

Iterators are generalizations of pointers, having:

- operator*()
- operator++(), operator++(int)

Iterators and the STL

```
std::vector<int> v = /* … */;

std::sort(v.begin(), v.end());
auto it = std::find(v.begin(), v.end(),
                    [](int elt) { return elt == 42; });
auto it = std::maximum_element(v.begin(), v.end());
```

Range-based algorithms

// using range-v3 library	*// C++20*

#include <range/v3/all.hpp>
 	
std::vector<int> vec = /*…*/;
 	
ranges::sort(vec);
auto it = ranges::find(
                        vec,
                        predicate);
	#include <ranges>
	
	std::vector<int> vec = /*…*/;
	
	std::ranges::sort(vec)
	auto it = std::ranges::find(
	                        vec,
	                        predicate);

Composing STL algorithms is awkward

```
// given
std::vector<int> v = {0,1,2,3,4,5,6,7,8,9};

// how to produce a vector consisting of the squares
// of the even elements?
```

Composing STL algorithms is awkward

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

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

std::vector<int> evens;
std::copy_if(v.begin(),
             v.end(),
             std::back_inserter(evens),
             [](int elt) { return elt % 2 == 0; });

std::vector<int> res;
std::transform(evens.begin(),
               evens.end(),
               std::back_inserter(res),
               [](int elt) { return elt * elt; });
```

Range version:

```
#include <vector>
#include <range/v3/all.hpp>

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

auto rng = v | ranges::view::filter(
                 [](int elt) { return elt % 2 == 0; })
             | ranges::view::transform(
                 [](int elt) { return elt * elt; });
```

Range concepts

```
template<class T>
concept bool Range =
  requires(T&& t) {
    { std::begin(t) } -> Iterator<T>
    { std::end(t) } -> Iterator<T>
  }
}

// see
https://github.com/CaseyCarter/cmcstl2/
// for proper definitions
```

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


// see
https://en.cppreference.com/w/cpp/experim
ental/ranges/range/View
```

Range concepts

```
template<class T>
concept bool Range =
  requires(T&& t) {
    { std::begin(t) } -> Iterator<T>
    { std::end(t) } -> Iterator<T>
  }
}
template<class T>
concept bool View =
  Range<T> &&
  Semiregular<T> &&
  ViewPredicate<T>;
```

```
template<class T>
  requires Range<T>
void Foo(T&& t) {
  // ...
}
```

What is a range (Version 3)

```
template<class T>
concept bool Range =
  requires(T&& t) {
    { std::begin(t) } -> Iterator<T>
    { std::end(t) } -> Iterator<T>
  }
}
```

A *range* is something which models the Range concept.

A few range adaptors

Filter, transform

- **Filter**. operator++ is interesting: skips until predicate is met
- **Transform**. operator* is interesting: it applies a predicate (only when asked for)

Range adaptors: all

```
auto rng2 = rng1 | view::all;
```

Range adaptors: Concat

Join: traverse first range, then second range, then …

```
std::vector<int> v = {1,2,3,4};
std::set<int> s =    {5,6,7,8,9};

auto rng = ranges::view::concat(v, s);

// traversing yields 1,2,3,4, 5,6,7,8,9
```

Range adaptors: take and slice

```
std::vector<int> v = {1,2,3,4,5,6,7,8,9,10};

auto first_five = v | ranges::view::take(5);

auto slice = v | ranges::view::slice(3,6);
```

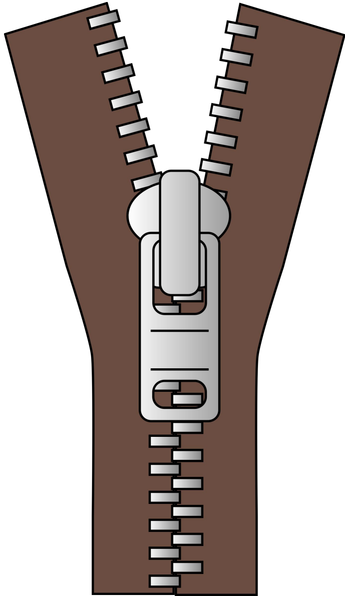
Range Adaptors: Join

Flatten a range of ranges to a single range:

```
std::vector<std::vector<int>> v = {{1,2}, {3,4,5}, {6,7,8,9}};

std::vector<int> flat = v | ranges::view::join;

// flat = {1,2,3,4,5,6,7,8,9};
```

Zip

```
std::vector<int> v = {1,2,3,4};
std::set<int> s =    {5,6,7,8,9};

auto rng = ranges::view::zip(v, s);

// traversal gives pairs of references:
// (1,5), (2,6), (3,7), (4,8)
```



http://ericniebler.com/2015/01/28/to-be-or-not-to-be-an-iterator/
https://ericniebler.github.io/std/wg21/D0022.html

Enumerate

```
// range-v3 version

std::set<std::string> s =
    {"hello", "world"};

for (auto const& [index, str] :
        ranges::view::enumerate(s))
{
    // do stuff
}
```

```
// old style

std::set<std::string> v =
    {"hello", "world"};


int index = 0;
for (auto const& str : s) {
    // do stuff
    ++index;
}
```

Range generators

```
auto nats = ranges::view::ints();

// yields 0,1,2,3, ...

// example, enumerate can be written:

auto enumerator = ranges::view::zip(ranges::view::ints(),
someRng);

auto first_ten_ints = nats | ranges::view::take(10);
```

Range actions

Eager, mutating, composable algorithms

```
std::vector<T> v;

v = move(v) | action::sort | action::unique;
```

Range actions

```
adjacent_remove_if
drop
drop_while
erase
insert
join
remove
remove_if
reverse
shuffle
```

```
slice
sort
split
split_when
stable_sort
stride
take
take_while
transform
unique
```

Projections

```
struct Person {
  int id;
};
std::vector<Person> v;
```

```
std::sort(v.begin(), v.end(), [](auto& p1, auto& p2) {
                                 return p1.id < p2.id; });
```

```
ranges::sort(v, std::less{}, [](auto& p) { return p.id; });
```

```
ranges::sort(v, std::less{}, &Person::id);
```

Projections && `std::invoke`

```
struct Person {
  int id;
};
```

What is `&Person::id??`

Pointer to member data: type `int Person::*`

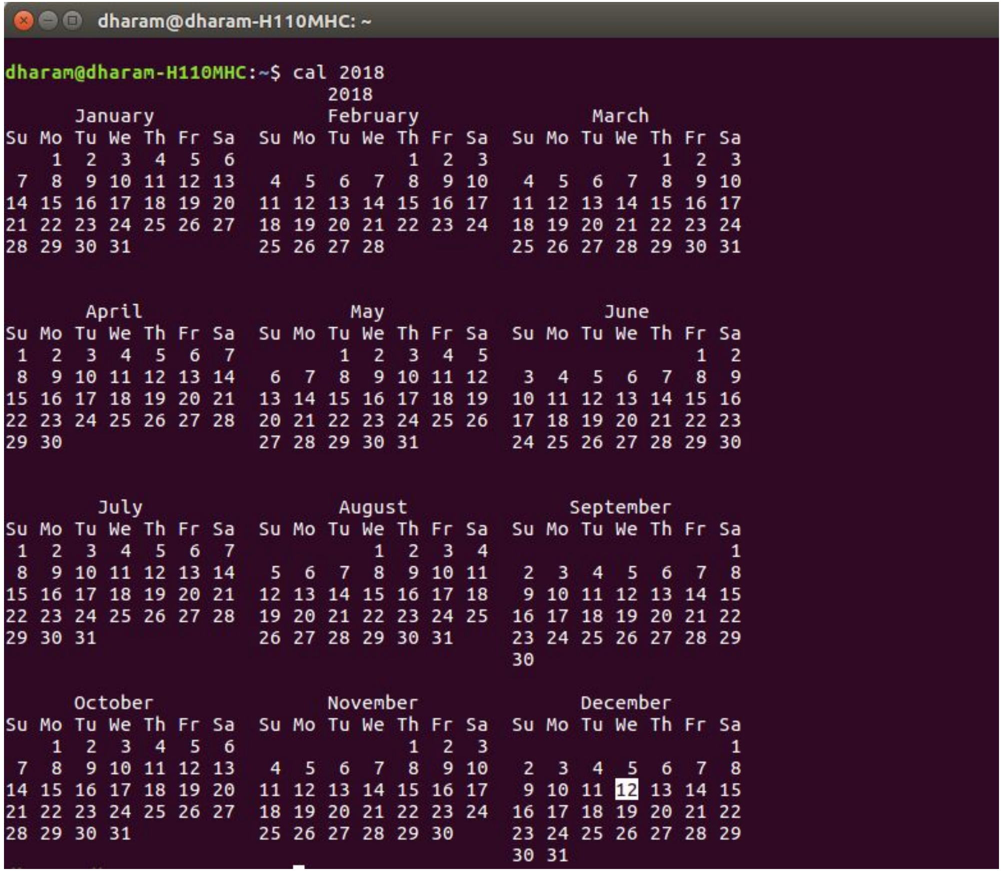
May be called dynamically:

```
  Person p {42};
  int Person::* memberDataPtr = &Person::id;
  assert(p.id == p.*memberDataPtr);
  assert(p.id == std::invoke(memberDataPtr, p));
  assert(p.id == std::invoke([](Person& p) { return p.id; }, p));
```

Instead of `f(args...)`, range-v3 uses the more general `std::invoke(f, args...)`

But wait, there's more!

- Templates for generating views!
- Templates for generating adaptors!
- (Work in progress) asynchronous ranges!



```
dharam@dharam-H110MHC:~$ cal 2018
                            2018
      January               February               March
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6               1  2  3               1  2  3
 7  8  9 10 11 12 13   4  5  6  7  8  9 10   4  5  6  7  8  9 10
14 15 16 17 18 19 20  11 12 13 14 15 16 17  11 12 13 14 15 16 17
21 22 23 24 25 26 27  18 19 20 21 22 23 24  18 19 20 21 22 23 24
28 29 30 31           25 26 27 28           25 26 27 28 29 30 31


       April                  May                   June
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7         1  2  3  4  5                  1  2
 8  9 10 11 12 13 14   6  7  8  9 10 11 12   3  4  5  6  7  8  9
15 16 17 18 19 20 21  13 14 15 16 17 18 19  10 11 12 13 14 15 16
22 23 24 25 26 27 28  20 21 22 23 24 25 26  17 18 19 20 21 22 23
29 30                 27 28 29 30 31        24 25 26 27 28 29 30


        July                 August              September
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7            1  2  3  4                     1
 8  9 10 11 12 13 14   5  6  7  8  9 10 11   2  3  4  5  6  7  8
15 16 17 18 19 20 21  12 13 14 15 16 17 18   9 10 11 12 13 14 15
22 23 24 25 26 27 28  19 20 21 22 23 24 25  16 17 18 19 20 21 22
29 30 31              26 27 28 29 30 31     23 24 25 26 27 28 29
                                            30

      October               November              December
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6               1  2  3                     1
 7  8  9 10 11 12 13   4  5  6  7  8  9 10   2  3  4  5  6  7  8
14 15 16 17 18 19 20  11 12 13 14 15 16 17   9 10 11 12 13 14 15
21 22 23 24 25 26 27  18 19 20 21 22 23 24  16 17 18 19 20 21 22
28 29 30 31           25 26 27 28 29 30     23 24 25 26 27 28 29
                                            30 31
```

Compiler support

From https://ericniebler.github.io/range-v3/

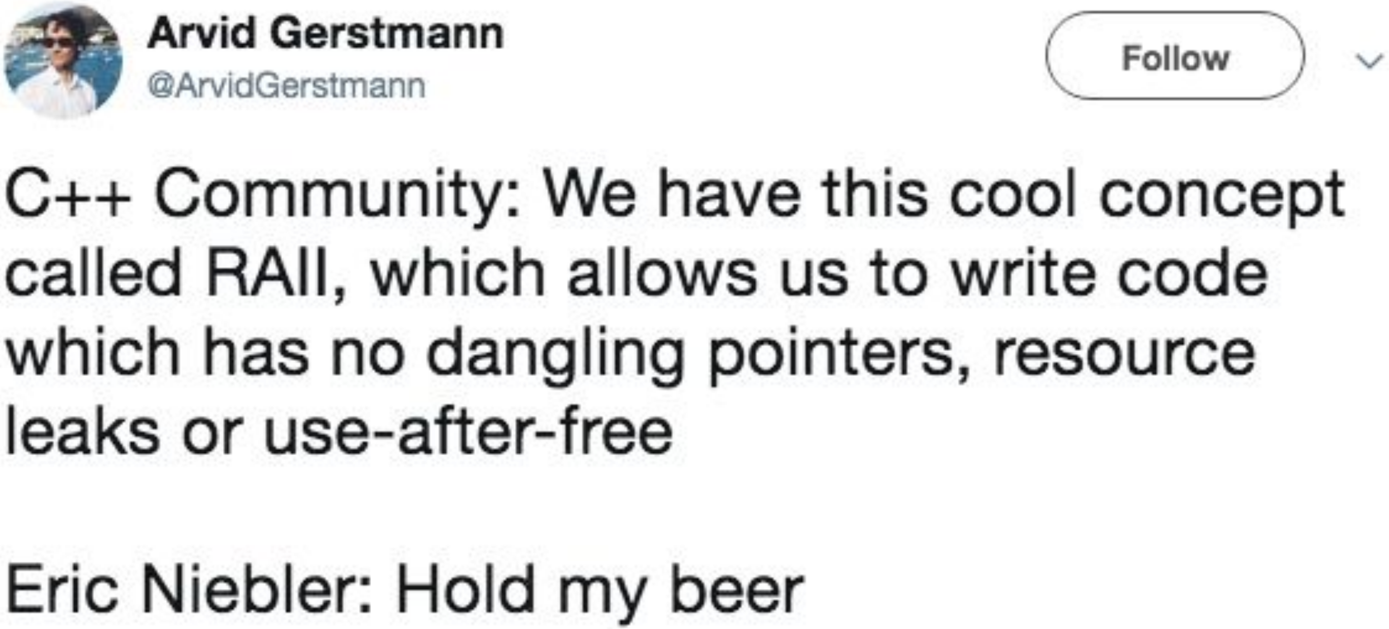
- clang 3.6.2
- GCC 4.9.1
- MSVC VS2017 15.9 (_MSC_VER >= 1916), with /std:c++17 /permissive-

Requirements

- Requires C++11/14/17
- pre-C++17, can't use range-based for on all ranges, need `RANGES_FOR` macro.

Downsides of ranges

Ownership



Arvid Gerstmann
@ArvidGerstmann

C++ Community: We have this cool concept called RAII, which allows us to write code which has no dangling pointers, resource leaks or use-after-free

Eric Niebler: Hold my beer

Ownership 2

```
auto rng = std::vector<int>{1,2,3} | ranges::view::reverse;
```

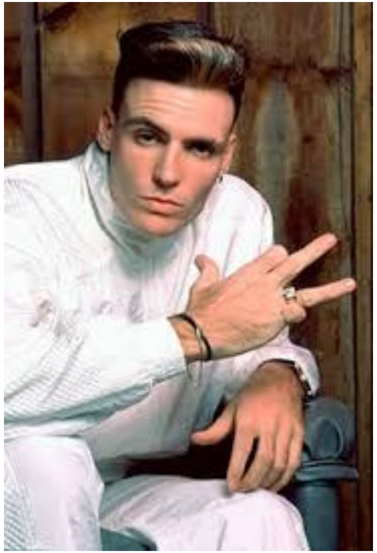
```
error: static assertion failed: You can't pipe an rvalue container into a view. First, save the container into a
named variable, and then pipe it to the view.

               static_assert(ranges::View<Rng>() || std::is_lvalue_reference<Rng>(),

                                    ~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~
```

Ownership 3

```
auto Oops()
{
    std::vector<int> v{1,2,3,4};
    return v | ranges::view::reverse;
}

int main() {
  for (auto elt : Oops()) { // use after free!
    // whatever
  }
}
```

ICE

```
fatal error C1001: An internal error has occurred in the compiler.
```



A few more of my complaints:

1. No view to const
2. Creating custom ranges/views is tedious
3. Error messages can be bad.

"Modern" C++ lamentations

https://aras-p.info/blog/2018/12/28/Modern-C-Lamentations/

Pythagorean Triples

A triple of integers (a,b,c) such that $a^2 + b^2 = c^2$.

Examples:

- (3, 4, 5): $3^2 + 4^2 = 5^2$
- (5,12,13): $5^2 + 12^2 = 13^2$

Pythagorean Triples

Printing the first n:

```
void printNTriples(int n)
{
    int i = 0;
    for (int z = 1; ; ++z)
        for (int x = 1; x <= z; ++x)
            for (int y = x; y <= z; ++y)
                if (x*x + y*y == z*z) {
                    printf("%d, %d, %d\n", x, y, z);
                    if (++i == n)
                        return;
                }
}
```

Pythagorean triples, lazily

From http://ericniebler.com/2018/12/05/standard-ranges/

```
auto triples =
  for_each(iota(1), [](int z) {
    return for_each(iota(1, z+1), [=](int x) {
      return for_each(iota(x, z+1), [=](int y) {
        return yield_if(x*x + y*y == z*z,
          make_tuple(x, y, z));
        });
      });
    });
```

Pythagorean triples, lazily

From http://ericniebler.com/2018/12/05/standard-ranges/

```
auto triples =
  for_each(iota(1), [](int z) {
    return for_each(iota(1, z+1), [=](int x) {
      return for_each(iota(x, z+1), [=](int y) {
        return yield_if(x*x + y*y == z*z,
          make_tuple(x, y, z));
        });
      });
    });
```

```
for_each(rng, f) <=> rng | transform(f) | join;
```

Pythagorean triples, lazily

From http://ericniebler.com/2018/12/05/standard-ranges/

```cpp
auto triples =
  for_each(iota(1), [](int z) {
    return for_each(iota(1, z+1), [=](int x) {
      return for_each(iota(x, z+1), [=](int y) {
        return yield_if(x*x + y*y == z*z,
          make_tuple(x, y, z));
        });
      });
    });
```

```haskell
-- Haskell
triples :: [(Int, Int, Int)]
triples = [(x, y, z)
            | z <- [0..]
            , x <- [1..z]
            , y <- [x..z]
            , x * x + y * y == z * z
            ]
```

```cpp
for_each(rng, f) = rng | transform(f) | join;
```

Pythagorean triples

```
auto triples =
  for_each(iota(1), [](int z) {
    return for_each(iota(1, z+1), [=](int x) {
      return for_each(iota(x, z+1), [=](int y) {
        return yield_if(x*x + y*y == z*z,
          make_tuple(x, y, z));
        });
      });
    });
for (auto elt: triples | ranges::view::take(100)) {
  std::cout << elt << "\n";
}
```

```
void printNTriples(int n)
{
    int i = 0;
    for (int z = 1; ; ++z)
        for (int x = 1; x <= z; ++x)
            for (int y = x; y <= z; ++y)
                if (x*x + y*y == z*z) {
                    printf("%d, %d, %d\n", x, y, z);
                    if (++i == n)
                        return;
                }
}

printNTriples(100);
```

Compile time: ~3 sec Runtime (debug): 300ms (0.3s) Runtime (release): 1ms	0.064 sec 2ms 0ms

Coroutines

```
#include <cppcoro/generator.hpp>
#include <tuple>

cppcoro::generator<std::tuple<int,int,int>> triples()
{
    for (int z = 1; ; ++z)
        for (int x = 1; x <= z; ++x)
            for (int y = x; y <= z; ++y)
                if (x*x + y*y == z*z) {
                    co_yield std::make_tuple(x,y,z);
                }
}
```

Conclusions

- Ranges give new primitives to more directly state intent.
- But they come with a cost: compile times and debug perf.