C++ONLINE

ROI BARKAN

TALK:

# MORE RANGES PLEASE

Slides

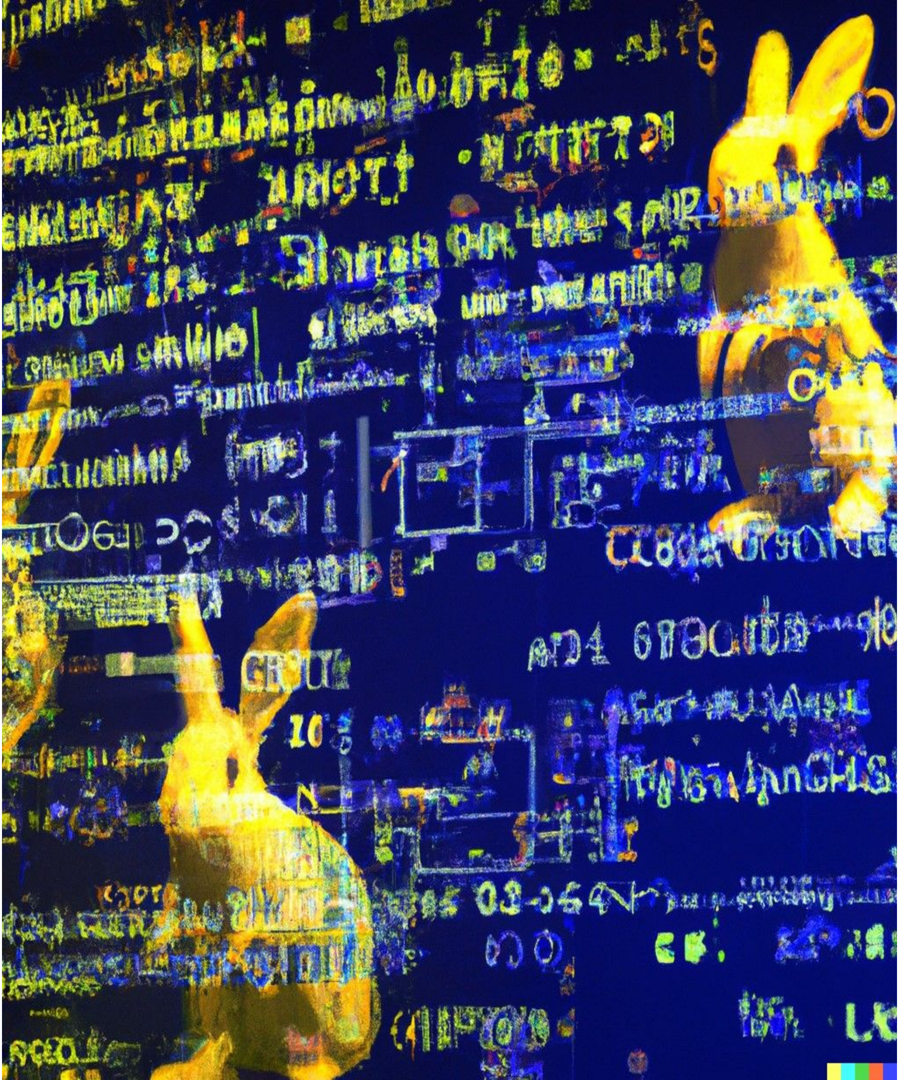
2025

# Hi, I’m Roi

- Roi Barkan (he/him) - רועי ברקן
- I live in Tel Aviv 🎗️
- C++ developer since 2000
- SVP Development & Technologies @ Istra Research
  - Finance, Low Latency, in Israel
  - careers@istraresearch.com
- Always happy to learn and explore
  - Please - ask questions, make comments

Slides

roi@istraresearch.com



# Outline

- Ranges
    - Introduction
    - Strengths, core ideas
    - Review the views
- Libraries
    - What
    - Why
    - How
- Rabbits
- Summary

# C++ Ranges

roi@istraresearch.com

# Ranges is a Breakthrough Library

- One of C++20 big-four features
- Rests on decades of existing libraries and experience
  - C++98 iterator-based algorithms
  - Fundamentals of functional / vectoric languages (APL, BQN, R, Julia, NumPy) [Conor Hoekstra]
  - Libraries of similar languages (D, Rust, Java) [Barry Revzin].
- Main Innovation - Composability
  - Many algorithms take ranges as input and return ranges as output
    - Opposed to in-place or output-iterator nature of C++98 algorithms
  - Range Adaptors - algorithms encalsupated as 'lazy ranges' (views)
    - Algorithms as composable objects - 'expression templates'
  - Projections - unary transformations of the ranges we inspect.

# Composability of Ranges

- Chaining algorithms due to range arguments and results

  `ranges::reverse(ranges::search(str,"abc"sv));` godbolt
- Views as composable lazy ranges

  `str | views::split(' ') | views::take(2);` godbolt
- Views have a value/algorithm duality

```
auto square_evens =
     views::filter([](auto x) { return int(x) % 2 == 0; }) |
     views::transform([](auto x) { return x * x; });
```
godbolt

roi@istraresearch.com

# The Views in the Standard (C++20/23*/26**)

- Factories/Generators: **empty**, **single**, **iota**, **repeat***, (**std::generator***)
- Rank preserving: **all**, **filter**, **transform**, **take{_while}**, **drop{_while}**, **reverse**, **stride***, **adjacent_transform***, **cache_latest****, (**counted**)
- Rank preserving - variadic→tuples: **zip***, **cartesian_product***
- Rank decreasing - tuples: **elements**, **keys**, **values**
- Rank decreasing - variadic: **zip_transform***, **concat****
- Rank decreasing - ranges: **join{_with*}**
- Rank increasing - tuples: **enumerate***, **adjacent***
- Rank increasing - ranges: **{lazy_}split**, **slide***, **chunk{_by}***
- Committee plan for C++26 is in P2760

roi@istraresearch.com

# Factories / Generators

```
namespace stdv = std::views;



stdv::empty<char>                          //=> []
stdv::single('+')                          //=> ['+']
stdv::iota(2,5)                            //=> [2, 3, 4]
stdv::repeat(0.3,3)                        //=> [0.3, 0.3, 0.3]
```

[godbolt]

roi@istraresearch.com

# Rank Preserving - 1/2

```
auto not5 = [](int i){return i != 5;};
auto mult2 = [](int i){return i * 2;};
auto iota2_10 = stdv::iota(2,10);

iota2_10 | stdv::all              //=> [2, 3, 4, 5, 6, 7, 8, 9]
iota2_10 | stdv::filter(not5)     //=> [2, 3, 4, 6, 7, 8, 9]
iota2_10 | stdv::transform(mult2) //=> [4, 6, 8, 10, 12, 14, 16, 18]
iota2_10 | stdv::take(6)          //=> [2, 3, 4, 5, 6, 7]
iota2_10 | stdv::drop(6)          //=> [8, 9]
```

godbolt

# Rank Preserving - 2/2

```
auto not5 = [](int i){return i != 5;};
auto iota2_10 = stdv::iota(2,10);

iota2_10 | stdv::take_while(not5)      //=> [2, 3, 4]
iota2_10 | stdv::drop_while(not5)      //=> [5, 6, 7, 8, 9]
iota2_10 | stdv::reverse               //=> [9, 8, 7, 6, 5, 4, 3, 2]
iota2_10 | stdv::stride(3)             //=> [2, 5, 8]
iota2_10 | stdv::adjacent_transform<2>(std::plus{})
                                       //=> [5, 7, 9, 11, 13, 15, 17]
iota2_10 | stdv::cache_latest      //=> [2, 3, 4, 5, 6, 7, 8, 9]
```

godbolt

roi@istraresearch.com

# Rank Preserving - Variadic ⇒ Tuples

```
auto iota2_7 = stdv::iota(2,7);
auto iota2_4 = stdv::iota(2,4);
auto iota6_9 = stdv::iota(6,9);

stdv::zip(iota2_7, iota6_9)                       //=> [(2, 6), (3, 7), (4, 8)]

stdv::cartesian_product(iota2_4, iota6_9)         //=> [(2, 6), (2, 7), (2, 8),
                                                        (3, 6), (3, 7), (3, 8)]
```

godbolt

roi@istraresearch.com

# Rank Decreasing - Tuples

```
auto the_zip = stdv::zip(iota2_7, iota6_9, "abcdef"sv);
                                 //=> [(2, 6, 'a'), (3, 7, 'b'), (4, 8, 'c')]

the_zip | stdv::keys             //=> [2, 3, 4]
the_zip | stdv::values           //=> [6, 7, 8]
the_zip | stdv::elements<2>      //=> ['a', 'b', 'c']
```

godbolt

roi@istraresearch.com

# Rank Decreasing - Variadic

```
auto iota2_7 = stdv::iota(2,7); auto iota6_9 = stdv::iota(6,9);

stdv::zip_transform(iota2_7, iota6_9, std::multiplies{})
                                        //=> [12, 21, 32]
stdv::concat(iota2_7, iota6_9)          //=> [2, 3, 4, 5, 6, 6, 7, 8]
```

godbolt

roi@istraresearch.com

# Rank Decreasing - Ranges

```
vector{"hey"sv, "C++"sv} | stdv::join
                                    //=> ['h', 'e', 'y', 'C', '+', '+']
(vector{"hey"sv, "C++"sv} | stdv::join_with(':'))
                                    //=> ['h', 'e', 'y', ':', 'C', '+', '+]
```

godbolt

roi@istraresearch.com

# Rank Increasing - Tuples

```
"hey"sv | stdv::enumerate
                //=> [(0, 'h'), (1, 'e'), (2, 'y')]
"hello"sv | stdv::adjacent<3>
                //=> [('h', 'e', 'l'), ('e', 'l', 'l'), ('l, 'l', 'o')]
```

godbolt

roi@istraresearch.com

# Rank Increasing - Ranges

```
"hey C++"sv | stdv::split(' ')  //=> [['h', 'e', 'y'], ['C', '+', '+']]
"hey C++"sv | stdv::lazy_split(' ')
                                //=> [['h', 'e', 'y'], ['C', '+', '+']]
"hello"sv | stdv::slide(3)      //=> [['h', 'e', 'l'], ['e', 'l', 'l'],
                                     ['l, 'l', 'o']]
"hey C++"sv | stdv::chunk(3)    //=> [['h', 'e', 'y'], [' ', 'C', '+'],
                                     ['+']]
"hello C++"sv | stdv::chunk_by(equal_to{})
 //=> [['h'], ['e'], ['l', 'l'], ['o'], [' '], ['C'], ['+', '+']]
```

godbolt

roi@istraresearch.com

# Recap: Views in the STL (C++20/23*/26**)

- Factories/Generators: **`empty`**, **`single`**, **`iota`**, **`repeat*`**, (**`std::generator*`**)
- Rank preserving: **`all`**, **`filter`**, **`transform`**, **`take{_while}`**, **`drop{_while}`**, **`reverse`**, **`stride*`**, **`adjacent_transform*`**, **`cache_latest**`**, (**`counted`**)
- Rank preserving - variadic→tuples: **`zip*`**, **`cartesian_product*`**
- Rank decreasing - tuples: **`elements`**, **`keys`**, **`values`**
- Rank decreasing - variadic: **`zip_transform*`**, **`concat**`**
- Rank decreasing - ranges: **`join{_with*}`**
- Rank increasing - tuples: **`enumerate*`**, **`adjacent*`**
- Rank increasing - ranges: **`{lazy_}split`**, **`slide*`**, **`chunk{_by}*`**
- Committee plan for C++26 is in [P2760]()

roi@istraresearch.com

# Libraries

roi@istraresearch.com

# What is a Library

- Code for coders
- Self contained reusable software
- Abstraction layer
- Language within a programming language
    - Domain specific language (DSL)
- Building blocks
- Simple interface (API) with tricky implementation
- Vocabulary
- …

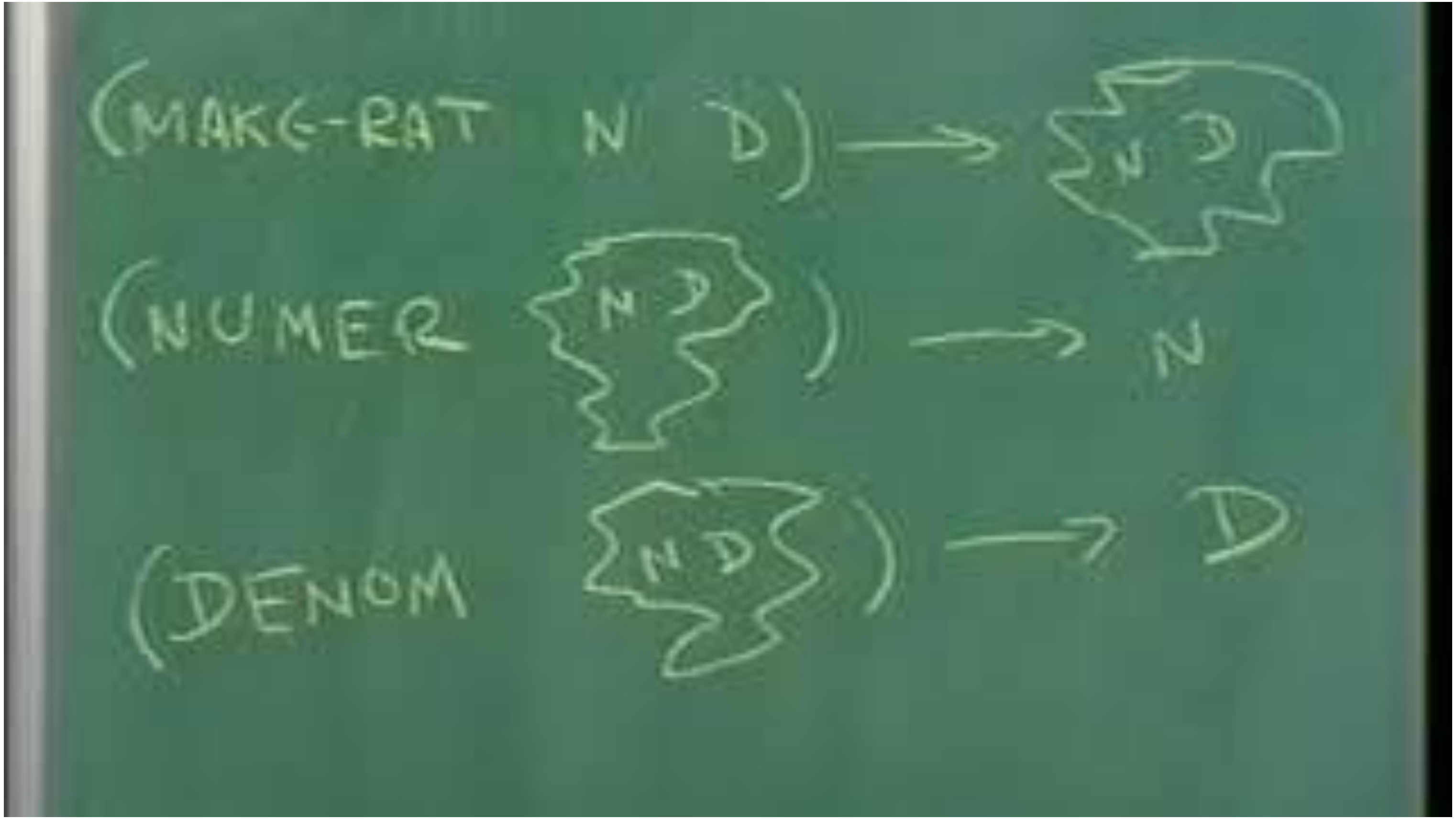
roi@istraresearch.com

# Clip

# SICP

# [link]

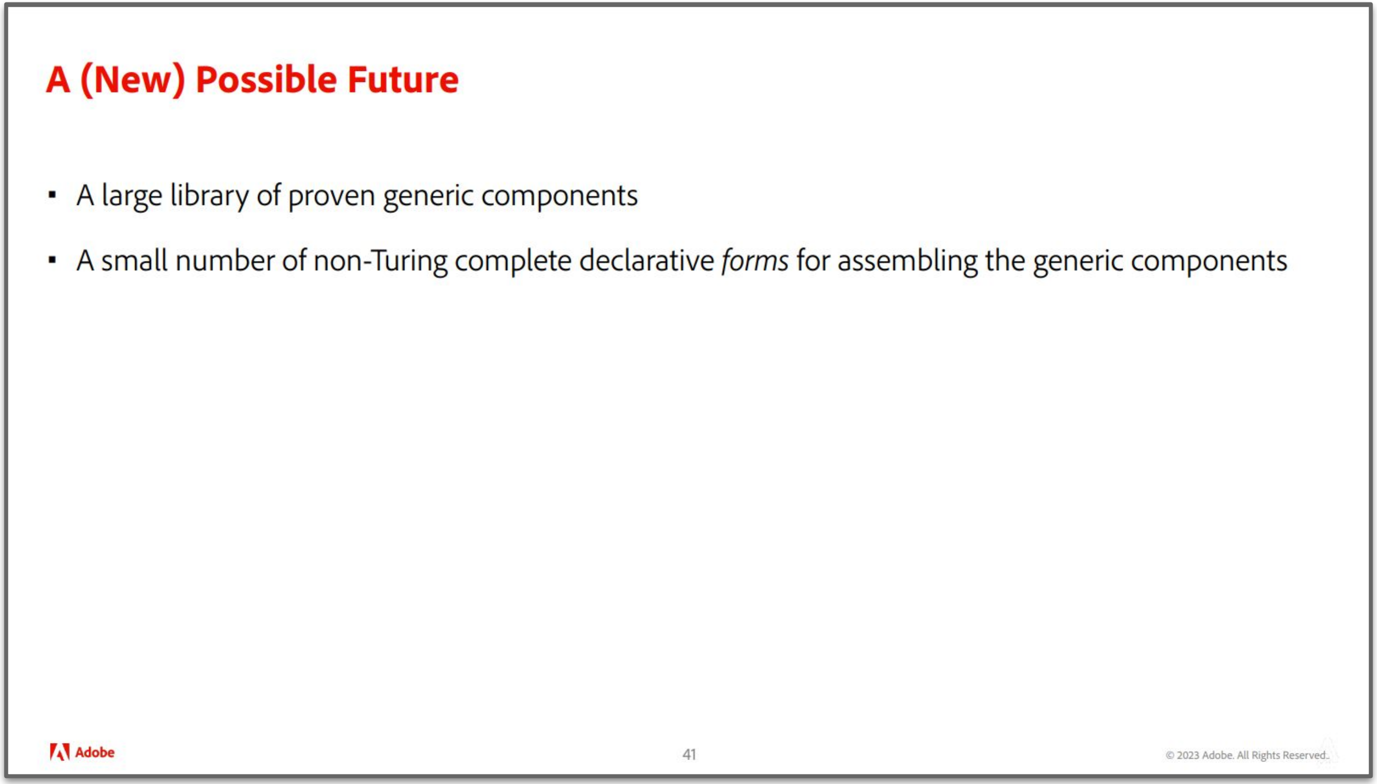
**Abelson, Sussman, Sussman 1984**



# Why Should We Write Libraries

- Reuse: don’t repeat yourself (DRY)
- Vocabulary: raise the level of discussion
- Building blocks:
    - The higher we go - change is more likely (close to users)
    - The lower we go - more stable, potentially robust.
- Abstraction layers:
    - Complexity is safer when it’s well encapsulated
    - Lower level likely better defined (pre/post-conditions)
    - Lower level is typically more testable.
- Libraries can be stacked / composed.

# Sean Parent's Vision




[All the Safeties C++Now 2023]

roi@istraresearch.com

# How to Write Libraries

- Start with algorithms (Christopher Di Bella, Eric Niebler)
  - Write the code that you need
  - Generalize, distill, *simplify*
  - Concepts will emerge from the *generic* attributes you find.
- Pay attention to the API - the abstraction layer
  - Easy to use, hard to misuse (Ben Deane)
  - Consider preconditions and postconditions
  - Return every potentially useful piece of information (Stepanov Law of Useful Return).
- Aim for *regular*, *algebraic* data types
  - Value semantic objects typically behave well in the language
  - Do you have a *group?*, a *monoid?*, a *monad*?

roi@istraresearch.com

# Finding Algorithms

- Algorithms are everywhere
  - Not just in CS textbooks
- Can be found in many places
  - Other languages
  - Other libraries
  - Research papers
  - Your own codebase

roi@istraresearch.com

# Clip

# Stepanov on Science

[link]

**STL and its Design Principles 2002**

roi@istraresearch.com

# More From Stepanov

link

# More Ranges

# First Inspiration - Hondt Method

- Credit to Tina Ulbrich - [How to Rangify Your Code]
- Hondt method - assigning parliament seats in a party-voting democracy

| | party 1<br>votes: 110 | | party 2<br>votes: 85 | | party 3<br>votes: 35 | |
|---|---|---|---|---|---|---|
| 1 | (1) | 110 / 1 = 110 | (2) | 85 / 1 = 85 | (6) | 35 / 1 = 35 |
| 2 | (3) | 110 / 2 = 55 | (4) | 85 / 2 = 42.5 | | 35 / 2 = 17.5 |
| 3 | (5) | 110 / 3 = 36.66 | (7) | 85 / 3 = 28.33 | | 35 / 3 = 11.66 |
| 4 | | 110 / 4 = 27.5 | | 85 / 4 = 21.25 | | 35 / 4 = 8.75 |
| 5 | | 110 / 5 = 22 | | 85 / 5 = 17 | | 35 / 5 = 7 |
| 6 | | 110 / 6 = 18.33 | | 85 / 6 = 14.16 | | 35 / 6 = 5.83 |
| 7 | | 110 / 7 = 15.71 | | 85 / 7 = 12.14 | | 35 / 7 = 5 |
| | seats: 3 | | seats: 3 | | seats: 1 | |

roi@istraresearch.com

# Basic Hondt Approach

| | party 1 votes: 110 | | party 2 votes: 85 | | party 3 votes: 35 | |
|---|---|---|---|---|---|---|
| 1 | (1) | 110 / 1 = 110 | (2) | 85 / 1 = 85 | (6) | 35 / 1 = 35 |
| 2 | (3) | 110 / 2 = 55 | (4) | 85 / 2 = 42.5 | | 35 / 2 = 17.5 |
| 3 | (5) | 110 / 3 = 36.66 | (7) | 85 / 3 = 28.33 | | 35 / 3 = 11.66 |
| 4 | | 110 / 4 = 27.5 | | 85 / 4 = 21.25 | | 35 / 4 = 8.75 |
| 5 | | 110 / 5 = 22 | | 85 / 5 = 17 | | 35 / 5 = 7 |
| 6 | | 110 / 6 = 18.33 | | 85 / 6 = 14.16 | | 35 / 6 = 5.83 |
| 7 | | 110 / 7 = 15.71 | | 85 / 7 = 12.14 | | 35 / 7 = 5 |
| | seats: 3 | | seats: 3 | | seats: 1 | |

- Calculate cells and sort them

```
std::ranges::sort(proportional_votes, [](const auto& rhs, const auto& lhs)
{
    return rhs.proportion > lhs.proportion;
});
proportional_votes.resize(total_number_of_seats);
```

- Key observations for P parties and S seats:
  - **`sort()`** requires eager evaluation and storage of S*P cells
  - **`resize()`** implies we are sorting too much - **`nth_element()`** requires O(SP) average steps
  - Each column is already sorted - we can **`merge()`**
    - Complexity can go down to O(SlogP)
    - **`merge()`** allow lazy evaluation - a range-view won’t need pre-computation/allocation

roi@istraresearch.com

# Rabbit 1 - Views for Sorted Ranges

- Suggestion - views for `merge`, `set_union`, `set_intersection`, `set_{symmetric_}difference`
  - Most algorithms can benefit from multi-input implementations
  - Heap (`priority_queue`) is needed for efficient `set_union, merge, ....`
- STL contains several algorithms for sorted ranges: `{inplace_}merge`, `includes`, `set_{union,intersection,{symmetric_}difference}`
  - Also search algorithms: `{upper,lower}_bound`, `equal_range`, `(unique)`.
- All the operations are lazy in nature
- Ranges-v3 has views for `set_{union,intersection,{symmetric_}difference}` with 2 input ranges
- D-lang has merge and multiWayMerge.

# Thoughts on Sorted Ranges (Rabbit 2)

- `equal_range`, `unique` can also be views - they have the right signature
- Binary search algorithm can become filtering views for random-access ranges
  - `take_until(value)`, `drop_until(value)` with O(logN) cost to cache the cut-off
    - Technically `take_until` mostly needs this to stay random-access.
  - `take_between(min,max)` can be more efficient than separate searches.
- If we'll have many algorithms specifically for sorted ranges, we might want a concept, which can also provide the comparator (`value_comp`)
  - D has it, as well as isSorted and assumeSorted factories.
- Simple combinations can be good for vocabulary:

```
auto histogram =
   views::chunk_by(std::equals{}) |
   views::transform([](const auto& rng) {
      return make_pair(begin(rng),distance(rng));});
```

roi@istraresearch.com

# Thoughts on Sorted Ranges (Rabbit 2)

- `equal_range`, `unique` can also be views - they have the right signature
- Search can also become filtering views for random-access ranges
  - `take_until(value)`, `drop_until(value)` with O(logN) cost to cache the cut-off
    - Technically `take_until` mostly needs this to stay random-access.
  - `take_between(min,max)` can be more efficient than separate searches.
- If we'll have many algorithms specifically for sorted ranges, we might want a concept, which can also provide the comparator (`value_comp`)
  - D has it, as well as isSorted and assumeSorted factories.
- Simple combinations can be good for vocabulary:

```
auto histogram =
    views::chunk_by(std::equals{}) |
    views::transform([](const auto& rng) {
      return make_pair(begin(rng),distance(rng));}
```

roi@istraresearch.com

# Digression - Algorithm Selection

- Sometimes the same goal can be achieved in several ways
  - `ranges::distance` - returns the distance between the beginning and end of a range
  - `ranges::ssize` - returns a signed integer equal to the size of a range
  - `ssize` only works for ranges that have constant-time calculation (opt-out semantic concept); `distance` allows linear calculation. Ben Deane recommends it.
- The library uses concepts to constrain which ranges are applicable for which algorithm/view, and to know the best method of reaching the intended goal
- Before C++20 other mechanisms were used to achieve this goal - and with concepts we have a way to be more precise and more flexible where needed.

# Breaking Sort Apart

- Inspired by the R standard library
- `sort()` is O(NlogN) in two aspects:
  - Comparisons (+projections)
  - `iter_swap` operations.
- Sometimes `iter_swap` is much more expensive than comparison
  - For example - sorting rows of a table by one column
  - Projections stress the potential difference between the objects in question.
- Suggested algorithm: `order()` - generate the sorting *permutation*
  - Requires allocating N indexes, and O(NlogN) comparisons
  - Then, *apply* the permutation with N `iter_swap` operations
  - Implementation is likely a one-liner using `iota()` and projections.

roi@istraresearch.com

# Rabbit 3 - Permutations

- Permutations allow more flexibility, not just intermediary step:
  - Can be reversed, to regain the original order
  - Can be lazily applied (be range views) - no swaps required.
- Iterator adapters that lazily apply permutations exist in boost, thrust.
- The standard library has other algorithms that deal with permutations.
- Potentially, there might be room for a **`permutation`**, or **`permutation_of<Range>`** concept.

# Rabbit 3 - Permutations

- Permutations allow more flexibility, not just intermediary step:
  - Can be reversed, to regain the original order
  - Can be lazily applied (be range views) - no swaps required.
- Iterator adapters that lazily apply permutations exist in boost, thrust.
- The standard library has other algorithms that deal with permutations.
- Potentially, there might be room for a `permutation`, or `permutation_of<Range>` concept.

roi@istraresearch.com

# Summary

- The C++ ranges library is an exemplar of composability
- Libraries can make software better, safer and cleaner
- Potential libraries are all around us - it's not just the STL or written by others
- Even C++ ranges might be improved/enhanced in novel ways
- "Go catch rabbits"

- Thank you !!
  - Questions and comments are welcome
  - Thanks to Bryce Lelbach for review and comments

Slides

roi@istraresearch.com