# C++20 Ranges. Practical examples.

Semen Lebedev
GSI

# Objectives of the Talk

- It is not meant to ba a full overview of `std::ranges` but rather to give some starting examples for newcomers to modern C++20.
- Understand what `std::ranges` is and its role in modern C++.
- Learn how to utilize `std::ranges` for more readable and efficient code.
- Explore practical examples to grasp `std::ranges` in action.

# What is a range?

- A range is a concept which defines the requirements for a type that allows iteration over its elements.
- A valid range should provide a `begin()` and `end()` (end sentinel).
- Examples: `std::array`, `std::vector`, `std::string_view`, `std::span`, C-style array etc.
- `#include <ranges>`
- `namespace std::ranges`

# Ranges vs. Traditional Iterators

- Traditional STL uses pairs of iterators to denote ranges, which can be error-prone and verbose.

```
std::vector<int> numbers{1, 4, 2, 7, 9, 3, 5};
std::sort(numbers.begin(), numbers.end());
```

- `std::ranges` simplifies this by treating the sequence as a unified object.

```
std::vector<int> numbers{1, 4, 2, 7, 9, 3, 5};
std::ranges::sort(numbers);
```

# Range Concepts

Range Concepts categorize ranges based on their capabilities, such as how they can be iterated:

- `std::ranges::range`
- `std::ranges::input_range` requires `std::input_iterator`
- `std::ranges::output_range` requires `std::output_iterator`
- `std::ranges::forward_range` requires `std::forward_iterator`
- `std::ranges::bidirectional_range` requires `std::bidirectional_iterator`
- `std::ranges::random_access_range` requires `std::random_access_iterator`
- `std::ranges::contiguous_range` requires `std::contiguous_iterator`

Concepts ensure that algorithms can operate on ranges in a type-safe manner.

```
std::list<int> numbers{5, 6, 1, 3, 7}; // std::list is bidirectional_range
std::ranges::sort(numbers); // Compilation error random_access_range not satisfied
```

# Views

- Views are lightweight ranges, non-owning references to data.
- Views can iterate over elements of ranges transforming or filtering data.
- Views have constant time complexity to copy, move, or assign view.
- `std::ranges::operation_view { range, arguments... }`

```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
std::ranges::for_each(std::ranges::take_view{numbers, 3},
                      [](int n){ std::cout << n << " "; });
// output: 1 2 3
```

Usually views are not created using their constructors, but instead using range adapters.

# Range Adaptors

- Range adaptors are helper functions that creates views, such as `views::filter`, `views::transform`, and `views::take`.
- It's preferable to use adaptors for creating views since they enable optimizations: `range | std::ranges::views::operation(arguments...)`
- Range adaptors can be composed together to build complex data processing pipelines efficiently.

```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
auto results = numbers | std::views::filter([](int n){ return n % 2 == 0; })
                       | std::views::transform([](int n){ return n * 2; });
for (const auto& n : results) {
    std::cout << n << " ";
}
// output: 4 8 12
```

# Function Composition, Pipelines

Because a view is a range, one can use a view as an argument for another view to enable chaining:

```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::views::reverse(
    std::views::take(
        std::views::transform(
            std::views::filter(numbers, [](const auto& n) { return n % 2 == 0; }),
            [](const auto& n) { return n * n; }),
        4)
    )
```

Excessive nesting can lead to readability and maintenance issues.
There is another way to combine ranges and views by using operator `|`:

```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto result = numbers | std::views::filter([](const auto& n) { return n % 2 == 0; })
                      | std::views::transform([](const auto& n) { return n * n; })
                      | std::views::take(4)
                      | std::views::reverse;
```

# Lazy Evaluation

- A view is just the description of a processing.
- The processing does not happen when one defines the view.
- The actual processing is performed element by element when the next value is asked for.

```
int main() {
    auto lazySquares = std::views::iota(1, 10)
                     | std::views::transform([](int i){ return i * i; });

    for (auto n : lazySquares) {
        std::cout << n << ' '; // Squares are calculated here
    }
}
```

# Bounded vs. Unbounded (Infinite) Range

- `std::views::iota` is a range factory that generates a sequence of elements by repeatedly incrementing an initial value.
- It can be either bounded or unbounded (infinite).
- `std::views::iota(10)` is lazy. It only produces a new value if it is asked for.

```
std::vector<int> vec1, vec2, vec3;
// Use bounded std::views::iota with begin and end value
for (int i : std::views::iota(0, 10)) vec1.push_back(i);
// Use infinite std::views::iota in combination with std::views::take
for (int i : std::views::iota(0) | std::views::take(10)) vec2.push_back(i);
// Use infinite std::views::iota in combination with std::views::take_while
for (int i : std::views::iota(0) | std::views::take_while([](int y) { return y < 10; })) vec3.push_back(i);

//All 3 vectors are equal: 0 1 2 3 4 5 6 7 8 9
```

# Lazy Evaluation, prime numbers example

```
// predicate returns true if num is prime
bool isPrime(int num) {
    if (num <= 1) return false;
    if (num % 2 == 0) return false;
    for (int i = 2; i * i <= num; ++i) {
        if (num % i == 0) return false;
    }
    return true;
}
```

```
int main() {
    // Generate an infinite range starting from 2
    auto numbers = std::ranges::iota_view{2};
    // Filter the numbers to keep only prime numbers and take the first 20
    auto primeNumbers = numbers | std::views::filter(isPrime) | std::views::take(20);
    // Print the first 20 prime numbers
    for (int prime : primeNumbers) {
        std::cout << prime << " ";
    }
}
```

# Projections

- Many range-based algorithms have a ***projection parameter***, a callback that transforms each element before it's processed.
- Example: projection parameter of the `std::ranges::sort()` function.

```
template<ranges::random_access_range R,
         class Comp = ranges::less,
         class Proj = std::identity>
requires std::sortable<ranges::iterator_t<R>, Comp, Proj>
constexpr ranges::borrowed_iterator_t<R>
sort( R&& r, Comp comp = {}, Proj proj = {} );
```

https://en.cppreference.com/w/cpp/algorithm/ranges/sort

# Projections, example #1

Sort `std::vector` of integers by absolut value.

Using custom comparator:

```
std::vector<int> numbers = {-8, 4, -3, 2, -5, 10, 7, 1, -9};
std::ranges::sort(numbers, [] (auto lhs, auto rhs) {
                        return std::abs(lhs) < std::abs(rhs);});
```

Using projection:

```
std::ranges::sort(numbers, std::ranges::less{}, [] (auto n) {
                        return std::abs(n);});
```

# Projections, example #2

Sort struct by different fields.

```
struct Person{
    std::string name{};
    std::size age{};
};
```

```
std::vector<Person> persons{ {"A", 10}, {"B", 2},{"C", 30},
                             {"D", 67}, {"E", 23}, {"F", 42} };

rng::sort(persons, {}, &Person::name); // ascending by name
rng::sort(persons, std::ranges::greater(), &Person::name); // descending by name
rng::sort(persons, {}, &Person::age); // ascending by age
rng::sort(persons, std::ranges::greater(),&Person::age); // descending by age
```

# Sentinels

- Sentinels represent the end of a range.
- Examples of sentinels in programming:
  - `EOF` as the end of a file stream
  - The null terminator `\0` as the end of a C-style string
  - `nullptr` as the end of a linked list
  - `-1` as the end of list containing non-negative integers
  - `\n` as the end of a line
- In contrast to traditional STL iterator-based approache where the begin and end of a sequence are denoted by iterators of the same type, a sentinel can have a different type.
- This feature can be useful when the end of the sequence is uncertain until it's reached during iteration.

# Sentinels, example #1

```
struct NegativeNumber {
    // c++20 compiler auto-generates the non-equal operator
    bool operator==(auto n) const {
        return *n < 0;
    }
};

int main() {
    std::vector<int> numbers{1, 2, 3, -4, 5, 6};
    std::ranges::transform(std::begin(numbers), NegativeNumber{},
                           std::begin(numbers), [](auto n) { return n * n; });

    std::ranges::for_each(std::begin(numbers), NegativeNumber{},
                          [](int n) { std::cout << n << " "; });
    // output: 1 4 9

    for (auto n: std::ranges::subrange{ std::begin(numbers), NegativeNumber{}}) {
        std::cout << n << " ";
    }
    // output: 1 4 9
}
```

# Sentinels, example #2

```
struct NullTerm {
    // c++20 compiler auto-generates the non-equal operator
    bool operator==(auto c) const {
        return *c == '\0';
    }
};

int main() {
    const char* rawString = "Hello, World!";

    std::ranges::for_each(rawString, NullTerm{},
                          [] (char c) { std::cout << ' ' << c; });
}
// Output: H e l l o ,   W o r l d !
```

# Dangling iterator #1

- When one passes a temporary range into an algorithm which returns an iterator, the returned iterator is invalid at the end of the statement because the range is destroyed.
- This issue couldn't occur when a range required two arguments (begin and end).
- The ranges library introduced ***borrowed iterators*** concept.
- If the range `R` passed to the algorithm is a temporary object ( `prvalue` ), return value is an object of type `std::ranges::dangling` .

```
template<ranges::forward_range R, class Proj = std::identity,
         std::indirect_strict_weak_order<std::projected<ranges::iterator_t<R>, Proj>> Comp = ranges::less>
constexpr ranges::borrowed_iterator_t<R>
    min_element( R&& r, Comp comp = {}, Proj proj = {} );
```

https://en.cppreference.com/w/cpp/algorithm/ranges/min_element

# Dangling iterator #2

This code compiles:

```
auto iter = std::ranges::min_element(getValues()); // iter will have type `ranges::dangling`
```

This code does not compile:

```
auto iter = std::ranges::min_element(getValues()); // iter will have type `ranges::dangling`
std::cout << *iter; // // Compile error: no match for 'operator*' (operand type is 'std::ranges::dangling')
```

Give it a name, create `lvalue`:

```
const auto& values = getValues(); // declare const lvalue reference
auto iter = std::ranges::min_element(values);
std::cout << *iter; // Now it is OK
```

# Keys View and Values View

- Example: get the keys and values of a `map`, `unordered_map` etc.
- Note different ways to write the same code.

```
std::map<std::string, int> numberMap{ {"one", 1}, {"two", 2}, {"three", 3}, {"four", 4}, {"five", 5}};
auto strings = std::views::keys(numberMap);
for (const auto& s : strings){ std::cout << s << " "; }
for (const auto& s : std::views::keys(numberMap)){ std::cout << s << " "; }
for (const auto& s : numberMap | std::views::keys){ std::cout << s << " "; }

auto numbers = std::views::values(numberMap);
for (const auto& n : numbers){ std::cout << n << " "; }
for (const auto& n : std::views::values(numberMap)) { std::cout << n << " ";}
for (const auto& n : numberMap | std::views::values) { std::cout << n << " ";}
```

# Examplle with std::variant

- Create a vector containing numbers and strings
- Filter out the strings using `std::views::filter`

```
int main() {
    std::vector<std::variant<int, std::string>> mixedNumbers = {1, 2, 3, "four", "five"};

    auto stringValues = mixedData | std::views::filter([](const auto& val) {
        return std::holds_alternative<std::string>(val);
    });

    for (const auto& str : stringValues) {
        std::cout << std::get<std::string>(str) << " ";
    }
    // Output: four five
}
```

# Mapping Elements

- Transforming elements of a range doesn't require the resulting range to have elements of the same type.
- One can map elements to another type.

```
int main() {
    std::vector<int> numbers { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    std::map<int, std::string> numberMap{ {1, "one"}, {2, "two"}, {3, "three"}, {4, "four"}, {5, "five"}};

    auto result = numbers
                | std::views::filter([](const auto& n) { return n <= 5; })
                | std::views::transform([&numberMap](const auto& n) { return numberMap[n]; });

    for (const auto& str : result) {
        std::cout << str << " ";
    }
    // Output: one two three four five
}
```

# Filtering and Accumulating Elements

- Combining std::ranges::views and std::accumulate (not range-based algorithm)
- The `std::views::common` creates a view with consistent types of begin and end iterators.
- Similar to `std::views::all`, but it generates a `std::ranges::common_view` if the range has iterators with different types.

```
...
#include <numeric>

int main() {
    //std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    auto numbers = std::views::iota(0) | std::views::take(10);
    auto evenNumbers = numbers
                     | std::views::filter([](int n){ return n % 2 == 0; })
                     | std::views::common;
    int sum = std::accumulate(evenNumbers.begin(), evenNumbers.end(), 0);
    std::cout << sum;
    // Output: 20
}
```

# all_of, any_of, none_of

Given the vector of integers. Check: 1) if any element is negative, 2) if none of the elements are negative, 3) if all elements are even

```
int main() {
    std::vector<int> numbers = {2, 4, 6, 8, 10};
    bool anyNegative = std::ranges::any_of(numbers, [](int x) { return x < 0; }); // false
    bool noneNegative = std::ranges::none_of(numbers, [](int x) { return x < 0; }); // true
    bool allEven = std::ranges::all_of(numbers, [](int x) { return x % 2 == 0; }); //true
}
```

```
int main() {
    std::vector<int> numbers = {2, 4, 6, 8, 10};
    bool anyNegative = false;

    for (std::size_t i = 0; i < numbers.size(); i++) {
        if (numbers[i] < 0) {
            anyNegative = true;
        }
    }
}
```

# Summary

- Usage of `std::ranges` library is highly beneficial.
- Syntax might seem unfamiliar to newcomers.
- It facilitates writing functional-style code.
- Combine operations on ranges using pipelines.
- Pipelines execute lazily.
- Learn and use standard algorithms whenever feasible.

`std::views::filter`, `std::views::transform`, `std::views::values`, `std::views::keys`, `std::ranges::min_element`, `std::ranges::sort`, `std::ranges::for_each`, `std::views::drop`, `std::views::reverse`, `std::views::iota`, `std::views::take`, `std::views::take_while`, `std::ranges::iota_view`, `std::ranges::subrange`, `std::accumulate`, `std::ranges::all_of`, `std::views::common`, `std::ranges::any_of`, `std::ranges::none_of`, `std::views::all`