

MODERN C++ PROGRAMMING COURSE – C++20

Nicolo' Genesio – @Nicogene

OUTLINE

- Concepts
- Ranges and range algorithm
- Modules

CONCEPTS – INTRO

- A mechanism for setting constraints in template types
- Move errors to build time
- Substitutes `static_assert`
- Standard – user defined concepts

Core language concepts

<code>same_as</code> (C++20)	specifies that a type is the same as another type (concept)
<code>derived_from</code> (C++20)	specifies that a type is derived from another type (concept)
<code>convertible_to</code> (C++20)	specifies that a type is implicitly convertible to another type (concept)
<code>common_reference_with</code> (C++20)	specifies that two types share a common reference type (concept)
<code>common_with</code> (C++20)	specifies that two types share a common type (concept)
<code>integral</code> (C++20)	specifies that a type is an integral type (concept)
<code>signed_integral</code> (C++20)	specifies that a type is an integral type that is signed (concept)
<code>unsigned_integral</code> (C++20)	specifies that a type is an integral type that is unsigned (concept)
<code>floating_point</code> (C++20)	specifies that a type is a floating-point type (concept)

CONCEPTS – SINTAX

After the **requires** it is needed a function that can be evaluated at **compile time**. There are 4 possible way to defining a concepts:

```
template <typename T>
requires std::integral<T>
T add (T a, T b) {
    return a+b;
}
```

```
template <std::integral T>
T add (T a, T b) {
    return a+b;
}
```

```
auto add(std::integral auto a,
         std::integral auto b) {
    return a+b;
}
```

```
template <typename T>
T add (T a, T b) requires std::integral<T>{
    return a+b;
}
```

CONCEPTS – USER DEFINED CONCEPT

- It is needed:
 - Template declaration with the type T on which the concept will operate
 - The keyword `concept` for declaring it
- A function can be specified with the `requires` keyword. It is not evaluated, just compiled.

```
template <typename T>
concept my_integral = std::integral<T>;

template <typename T>
concept Multiplicable = requires(T a, T b) {
    a * b; // Only a syntax check
};
```


CONCEPTS – THE REQUIRES CLAUSE

- There are 4 types of requirements


- Simple requirement

- Nested requirement

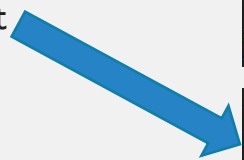
- Compound requirement



```
template <typename T>
concept big_type = requires(T t) {
    sizeof(T) >=4; // Only a syntax check
};
```



```
template <typename T>
concept big_type = requires(T t) {
    sizeof(T) >=4; // Only a syntax check
    requires sizeof(T) >=4; // Nested requirement, the function is evaluated
};
```



```
template <typename T>
concept Subtractable = requires (T a, T b) {
    {a - b} -> std::convertible_to<int>; //compound requirement
    // Check that a-b compile and the result is convertible to int
};
```

- Concepts can be combined with **&&** and **||** operators.

```
template <typename T>
concept big_type_subtractable = Subtractable<T> && big_type<T>;
```

CONCEPTS – AUTO

- Concepts can be used for enforcing the type characteristics when using auto.
- Analogously on template, if auto represent a type that violates the concepts it throws a compiler error.

```
std::integral auto add(std::integral auto a, std::integral auto b){  
    return a+b  
}
```

You, 1 second ago • Uncommitted changes

```
std::integral auto x = add(10,20);
```

RANGES – INTRO

- The C++20 ranges library offers the following features:
 - Range algorithms
 - Projections
 - Views and view adaptors
 - Function composition
 - Range factories

RANGES – RANGE ALGORITHMS

- Different from the legacy algorithms that used to work on iterators, these works directly on containers.
- Each std algorithm come in both versions.
- The range algorithms are in the **ranges** namespace.
- They works also with iterators.
- The ranges algorithms are constrained with **concepts**. → Better error messages

Under the hood
iterators are still
used!

LEGACY

```
std::vector<int> numbers {11,2,6,4,8,3,17,9};
auto odd = [](int n){
    return n%2 !=0;
};
auto result = std::all_of(numbers.begin(), numbers.end(),odd);
if(result){
    std::cout << "All elements in numbers are odd" << std::endl;
}else{
    std::cout << "Not all elements in numbers are odd" << std::endl;
}
```



RANGE

```
std::vector<int> numbers {11,2,6,4,8,3,17,9};
auto odd = [](int n){
    return n%2 !=0;
};
auto result = std::ranges::all_of(numbers,odd);
if(result){
    std::cout << "All elements in numbers are odd" << std::endl;
}else{
    std::cout << "Not all elements in numbers are odd" << std::endl;
}
```

RANGES – PROJECTIONS

- They allows an algorithm to work on a given aspect of the type of the collection
- Ex: we want to sort a vector of Point (m_x, m_y) comparing m_x.

Just invoke operator<()

```
//Sorting with a projection : The data is passed into the projection before
//it's passed into the comparator. std::less<> is going to compare two doubles
//instead of comparing two Points.
std::cout << std::endl;
std::cout << "projection on Point::m_x : " << std::endl;
print_collection(points);
std::ranges::sort(points, std::less<>{}, [](auto const & p){
    return p.m_x;
});
```

This lambda specifies on what operator<() will be invoked, then it is a comparison between doubles not Point

You can also just pass the **public** member var to be compared

```
std::ranges::sort(points, std::less<>{}[]; &Point::m_x);
```

RANGES – VIEWS AND VIEW ADAPTORS

- A view is a **non-owning** range.
- It is a “view” for checking data without the infrastructure to store it.
- Cheap to copy, they are designed to pass it as parameters.
- E.g.: given a big collection of int (**vi**), “view” only the even numbers.
- The view adaptor is who creates a view, there are several in the standard

View

```
//std::ranges::filter_view
std::cout <<std::endl;
std::cout << "std::ranges::filter_view : " << std::endl;
auto evens = [](int i){
    return (i % 2) == 0;
};
std::cout << "vi : " ;
print(vi);
std::ranges::filter_view v_evens = std::ranges::filter_view(vi, evens);
std::cout << "vi evens : ";
print(v_evens); //Computation happens in the print function
//Print evens on the fly
std::cout << "vi evens : " ;
print(std::ranges::filter_view(vi, evens));
//Print odds on the fly
std::cout << "vi odds : " ;
print(std::ranges::filter_view(vi, [](int i){
    return (i % 2) != 0;
}));
```

You, 1 second ago • Uncommitted changes

View adaptor

```
auto v_evens = std::views::filter(vi, even);
```

We get a
subset without
computation!

<code>views::all</code> (C++20)	a view that includes all elements of a range
<code>views::all</code> (C++20)	(alias template) (range adaptor object)
<code>ranges::ref_view</code> (C++20)	a view of the elements of some other range
<code>ranges::owning_view</code> (C++20)	a view with unique ownership of some range
<code>ranges::filter_view</code> (C++20)	a view that consists of the elements of a range that satisfies a predicate
<code>views::filter</code> (C++20)	(class template) (range adaptor object)
<code>ranges::transform_view</code> (C++20)	a view of a sequence that applies a transformation function to each element
<code>views::transform</code> (C++20)	(class template) (range adaptor object)
<code>ranges::take_view</code> (C++20)	a view consisting of the first N elements of another view
<code>views::take</code> (C++20)	(class template) (range adaptor object)
<code>ranges::take_while_view</code> (C++20)	a view consisting of the initial elements of another view, until the first element on which a predicate returns false
<code>views::take_while</code> (C++20)	(class template) (range adaptor object)
<code>ranges::drop_view</code> (C++20)	a view consisting of elements of another view, skipping the first N elements
<code>views::drop</code> (C++20)	(class template) (range adaptor object)
<code>ranges::drop_while_view</code> (C++20)	a view consisting of the elements of another view, skipping the initial subsequence of elements until the first element where the predicate returns false
<code>views::drop_while</code> (C++20)	(class template) (range adaptor object)
<code>ranges::join_view</code> (C++20)	a view consisting of the sequence obtained from flattening a view of ranges
<code>views::join</code> (C++20)	(class template) (range adaptor object)
<code>ranges::split_view</code> (C++20)	a view over the subranges obtained from splitting another view using a delimiter
<code>views::split</code> (C++20)	(class template) (range adaptor object)
<code>ranges::lazy_split_view</code> (C++20)	a view over the subranges obtained from splitting another view using a delimiter
<code>views::lazy_split</code> (C++20)	(class template) (range adaptor object)
<code>views::counted</code> (C++20)	creates a subrange from an iterator and a count
<code>views::common_view</code> (C++20)	(customization point object)
<code>views::common</code> (C++20)	converts a view into a common range
<code>views::reverse_view</code> (C++20)	(class template) (range adaptor object)
<code>views::reverse</code> (C++20)	a view that iterates over the elements of another bidirectional view in reverse order
<code>ranges::elements_view</code> (C++20)	takes a view consisting of tuple-like values and a number N and produces a view of Nth element of each tuple
<code>views::elements</code> (C++20)	(class template) (range adaptor object)
<code>ranges::keys_view</code> (C++20)	takes a view consisting of pair-like values and produces a view of the first elements of each pair
<code>views::keys</code> (C++20)	(class template) (range adaptor object)
<code>ranges::values_view</code> (C++20)	takes a view consisting of pair-like values and produces a view of the second elements of each pair
<code>views::values</code> (C++20)	(class template) (range adaptor object)
<code>ranges::enumerate_view</code> (C++23)	a view that maps each element of adapted sequence to a tuple of both the element's position and its value
<code>views::enumerate</code> (C++23)	(class template) (range adaptor object)
<code>ranges::zip_view</code> (C++23)	a view consisting of tuples of references to corresponding elements of the adapted views
<code>views::zip</code> (C++23)	(class template) (customization point object)
<code>ranges::zip_transform_view</code> (C++23)	a view consisting of tuples of results of application of a transformation function to corresponding elements of the adapted views
<code>views::zip_transform</code> (C++23)	(class template) (customization point object)
<code>ranges::adjacent_view</code> (C++23)	a view consisting of tuples of references to adjacent elements of the adapted view
<code>views::adjacent</code> (C++23)	(class template) (range adaptor object)
<code>ranges::adjacent_transform_view</code> (C++23)	a view consisting of tuples of results of application of a transformation function to adjacent elements of the adapted view
<code>views::adjacent_transform</code> (C++23)	(class template) (range adaptor object)
<code>ranges::join_with_view</code> (C++23)	a view consisting of the sequence obtained from flattening a view of ranges, with the delimiter in between elements
<code>views::join_with</code> (C++23)	(class template) (range adaptor object)
<code>ranges::slide_view</code> (C++23)	a view whose Mth element is a view over the Mth through (M + N - 1)th elements of another view
<code>views::slide</code> (C++23)	(class template) (range adaptor object)
<code>ranges::chunk_view</code> (C++23)	a range of views that are N-sized non-overlapping successive chunks of the elements of another view
<code>views::chunk</code> (C++23)	(class template) (range adaptor object)
<code>ranges::chunk_by_view</code> (C++23)	splits the view into subranges between each pair of adjacent elements for which the given predicate returns <code>false</code>
<code>views::chunk_by</code> (C++23)	(class template) (range adaptor object)
<code>ranges::as_const_view</code> (C++23)	converts a view into a constant range
<code>views::as_const</code> (C++23)	(class template) (range adaptor object)
<code>ranges::as_rvalue_view</code> (C++23)	a view of a sequence that casts each element to an rvalue
<code>views::as_rvalue</code> (C++23)	(class template) (range adaptor object)
<code>ranges::stride_view</code> (C++23)	a view consisting of elements of another view, advancing over N elements at a time
<code>views::stride</code> (C++23)	(class template) (range adaptor object)

RANGES – FUNCTION COMPOSITION

- It is possible to **compose** views. E.g. get the square of all even numbers:

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

//Filter out evens and square them out.
std::cout << "vi : " ;
print(vi);
//V1 : Raw function composition
auto even = [](int n){return n%2==0;};
auto my_view = std::views::transform(std::views::filter(vi,even) , [](auto n){return n*=n;});
```

- You can do the same w/ the **pipe operator**

```
auto my_view1 = vi | std::views::filter(even)
                  | std::views::transform([](auto n){return n*=n;});
```

It has the
meaning of
“passed to”

RANGES – RANGES FACTORIES

- They allow to create views out of the blue, without the need of creating a container and apply a view on it.

The numbers
are generated
only when
accessed!

```
// Generate an infinite sequence of numbers
auto infinite_view = std::views::iota(1);
// Number initialized lazily at each iteration
for(auto i : infinite_view){
    std::cout << i << std::endl; // ENDLESS LOOP!
}
// Provide an upper limit
auto finite_view = std::views::iota(1,10);
for(auto i : finite_view){
    std::cout << i << std::endl;
}
// Same as before but fancier
for (auto i : infinite_view | std::views::take(10)){
    std::cout << i << std::endl;
}
```

Range factories

Defined in header `<ranges>`
Defined in namespace `std::ranges`

`ranges::empty_view` (C++20)
`views::empty`

`ranges::single_view` (C++20)
`views::single`

`ranges::iota_view` (C++20)
`views::iota`

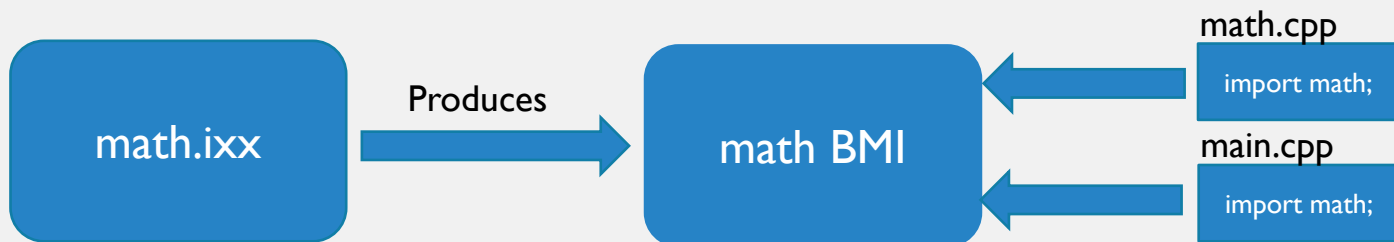
`ranges::basic_istream_view` (C++20)
`views::istream`

`ranges::repeat_view` (C++23)
`views::repeat`

`ranges::cartesian_product_view` (C++23)
`views::cartesian_product`

MODULES – INTRO

- C++20 features that tries to solve some “headers” problems
- Including a header basically means copy the code you are including
- Here are the problems modules address:
 - Compilation speed
 - One Definition Rule violations
 - Include order
- The modules are defined by Module interface files (.ixx for VS, .cc for GCC, .cppm for Clang) and Binary Module Interface (BMI). Not included anymore as file but **imported as binary**.



MODULES – IXX STRUCTURE

Module declaration ←

Name of the
module ←

Header importation:

- Can be done on std c++ headers, not C.
- The compiler transform the header into a BMI, adding the correct module declaration.

The implementation of the functions can be defined in the module implementation file (.cpp). It is the same of .iix except it does not have the export statements, and contains the definitions

} Global Module fragment, preprocessor instructions can be ONLY here!

} Module preamble, import other modules

} Module purview, it can contain multiple functions, classes. **export** make the function visible from outside, otherwise, it remains visible only inside the module. Alternatively export block can be used

```
export {  
    int one() {  
        return 1;  
    }  
    int two() {  
        return 2;  
    }  
}
```

MODULES – EXPORT IMPORT

- This clause is used for make the importers of the module, to import a certain module.
- If someone import `MyFirstModule`, will also access to `string_view` module.
- Otherwise, the visibility of the imported modules is in the module itself.

MODULES – SUBMODULES

- Divide the modules in sub-parts that can be imported and used independently.

```
module;  
  
export module math;  
  
export import math.add_sub;  
export import math.mult_div;
```

math.ixx

```
module;  
  
export module math.add_sub;  
  
export{  
    double add(double a, double b) {  
        return a + b;  
    }  
  
    double sub(double a, double b) {  
        return a - b;  
    }  
}
```

math_add_sub.ixx

```
module;  
  
export module math.mult_div;  
  
export{  
    double mult(double a, double b) {  
        return a * b;  
    }  
  
    double div(double a, double b) {  
        return a / b;  
    }  
}
```

math_mult_div.ixx

MODULES – MODULE INTERFACE PARTITIONS

- It is the dual of submodules, the partitions cannot be imported outside its own module.

```
module;  
  
export module math;  
  
export import :add_sub;  
export import :mult_div;
```

math.ixx

```
module;  
  
export module math:add_sub;  
  
export{  
    double add(double a, double b) {  
        return a + b;  
    }  
  
    double sub(double a, double b) {  
        return a - b;  
    }  
}
```

math_add_sub.ixx

```
module;  
  
export module math:mult_div;  
  
export{  
    double mult(double a, double b) {  
        return a * b;  
    }  
  
    double div(double a, double b) {  
        return a / b;  
    }  
}
```

math_mult_div.ixx