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

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

ISTITUTO ITALIANO
DI TECNOLOGIA

- 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 trows 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

ISTITUTO ITALIANO
DI TECNOLOGIA

- 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){       You, 1 second ago • Uncommitted changes
    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

```cpp
//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

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

We get a subset without computation!

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

RANGES – FUNCTION COMPOSITION

ISTITUTO ITALIANO DI TECNOLOGIA

- 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”

ISTITUTO ITALIANO
DI TECNOLOGIA

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;
}
```

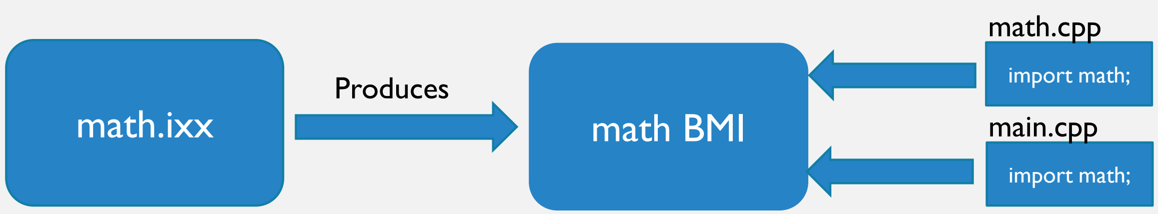
Range factories

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

ranges::empty_view views::empty	(C++20)
ranges::single_view views::single	(C++20)
ranges::iota_view views::iota	(C++20)
ranges::basic_istream_view views::istream	(C++20)
ranges::repeat_view views::repeat	(C++23)
ranges::cartesian_product_view views::cartesian_product	(C++23)

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

ISTITUTO ITALIANO
DI TECNOLOGIA

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.

```
module;

#include <stdlib.h>

export module MyFirstModule;

import <ctime>;        You, 1 second ago • Uncommitted
import <iostream>;

export void print_info() {
    std::cout<<"This is my first Module!"<<std::endl;
}
```

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;
    }
}
```

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, an contains the definitions

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.

```
module;

#include <stdlib.h>

 export module MyFirstModule;

import <ctime>;
import <iostream>;
export import <string_view>;

export void print_message(std::string_view message) {
    std::cout<<message<<std::endl;
}
```

MODULES – SUBMODULES

iCub Tech
ISTITUTO ITALIANO
DI TECNOLOGIA

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

```cpp
module;

export module math;

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

math.ixx

```cpp
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

```cpp
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.

```cpp
module;

export module math;

export import :add_sub;
export import :mult_div;
```

math.ixx

```cpp
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

```cpp
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