June 2021 Tech Talk
C++ 20 Enhancements

Rushikesh (R3H)

C++ revisions

1980	1985	1998	2003	2011	2014	2017	2020	2023
C with classes	The C++ programming language	C++98 ISO/IEC 14882:1998	C++03 ISO/IEC 14882:2003	C++11 ISO/IEC 14882:2011	C++14 ISO/IEC 14882:2014	C++17 ISO/IEC 14882:2017	C++20 ISO/IEC 14882:2020	C++23
First imple menta tion			Minor	Major	Minor	Minor	Major	

ISO International Organization for Standardization

IEC International Electrotechnical Commission

C++ 20 Enhancements

Major	Language	Library
❑ Modules ❑ Concepts ❑ Ranges ❑ Coroutines (not discussed)	▪ <=> 3 way comparison operator (spaceship operator) ▪ Designated initializers	▪ std::format ▪ <numbers>

Modules

- Issues with existing C++ build system.
- Modules as a solution to improve build performance.
- An example demonstrating modules.
- Ways to manage larger modules:
  - Sub modules.
  - Module partitions.
- Transition from present day `#include` to new way of modules.

Modules – Issues with present day build system

```cpp
#include <iostream> // thousands of lines of code are included here.

int main()
{
   std::cout << "Hello Delmia Programmers!";
}
```

- Nothing can get as simple as this program. But yet compiler has to compile over 50k lines of code!
- I got 1891 ms clcompile time for this program using Microsoft Visual Studio compiler.
- `#include` is a pre-processor directive. It includes all the contents of header in current compilation unit.
- This build system in C++ is inherited from C language build system from 1970s.

Modules – Issues with present day build system

```
#include "CATMfgResourceUpgradeExt.h"

#include "ManufacturingResourceConditions.h"
#include "CATMfgToolConstant.h"
#include "CATMfgToolAssemblyConstant.h"
#include "CATMfgResourceUtility2.h"
#include "CATIMfgToolExtensionManagement.h"
#include "CATIMfgPLMResource.h"
#include "CATIMfgResource.h"
#include "CATMfgUtilities.h"
#include "CATMfgModelUtilities2.h"
#include "CATIMfgResourceFactory.h"
#include "DELIPLMRMResourceRepresentations.h"
#include "DELIPLMRefReps.h"
#include "CATIMfgMillAndDrillToolAssembly.h"
#include "CATIMfgLatheToolAssembly.h"
#include "CATMfgModelUtilities.h"
#include "CATIMfgPLMToolAssembly.h"
#include "CATIMfgSubAssembly.h"
#include "CATIMfgPLMTool.h"
#include "CATIMfgPLMUserRepresentation.h"
#include "CATIPrdIterator.h"
#include "CATIPLMRepInstances.h"
```

CATMfgResourceUtility2.h

```
#include "CATUnicodeString.h"
#include "CATListOfCATUnicodeString.h"
#include "CATBaseUnknown.h"
#include "CATListOfCATBaseUnknown.h"

#include "MfgResourceEnv.h"
#include "CATIContainer.h"
#include "CATIPLMProducts.h"
#include "CATIPsiRepresentationReference.h"
#include "CATOmbLifeCycleRootsBag.h"
#include "DELIPLMRMResourceAttributes.h"
#include "DELIPLMRMMechanicalPortS.h"
#include "CATIMfgFSConfiguration.h"
#include "CATIMf3DAxisSystem.h"
#include "CATLISTP_CATMathTransformation.h"
#include "CATIProduct.h"
#include "CATRep.h"
#include "CATViewer.h"
#include <CATLISTP_Declare.h>
```

- In real life software there are lots of `#include`.
- Each header again has lots of `#include`. This results in recursive `#include` and hence longer build times.
- Longer build times has been a constant complaint in C++ community.

Modules – Issues with present day build system

CATMfgResourceUtility2.h

```
#ifndef CATMfgResourceUtility2_h
#define CATMfgResourceUtility2_h
```

- Include guards are needed in each header to protect against multiple inclusion.
- It needs to be unique.
- Trivial changes in header such as adding comment, spaces etc. causes re-building of all the source files including this header directly or indirectly.
- Order of inclusion matters when header contains macros. Macro definition may get overridden if same macro is defined in multiple headers.

Modules – as a solution to improve build performance

- Starting from C++20 module is going to be the new compilation unit.
- Modules do not include the contents like `#include`.
- Modules do not result in re-compilation of source files because of trivial changes such as adding spaces, comments etc. Rebuild happens only when signature of exported functions changes.
- Macros cannot be exported to modules. So order of import of modules does not matter.
- C++ wants to get rid of macros. Modules will help achieve this.
- Modules are expected to improve build times from 10 to 50 times for real world software.
- Debug tools for C++ are expected to improve with modules.

An example of module

Module interface file: mymodule.ixx

```cpp
export module mymodule;
import <string>;

export enum Color {RED, GREEN, BLUE};

export void PrintColor(Color color);

export class Person
{
  std::string name;

  public:
    Person(const std::string& iName);
    std::string GetName();
};

void some_internal_function(); // module local function - not exported.
```

```cpp
export module mymodule;
import <string>;

export
{
  enum Color {RED, GREEN, BLUE};

  void PrintColor(Color color);

  class Person
  {
    std::string name;

    public:
      Person(const std::string& iName);
      std::string GetName();
  };
}

void some_internal_function(); // module local function - not exported.
```

An example of module

Module implementation file: mymodule.cpp

```cpp
module mymodule;
import <iostream>;

void PrintColor(Color color)
{
   std::string colors[] {"Red", "Green", "Blue"};
   std::cout << colors[color];
}

Person::Person(const std::string& iName): name(iName)
{
}

std::string Person::GetName()
{
   return name;
}

void some_internal_function()
{
   // some implementation.
}
```

An example of module

User of mymodule: main.cpp

```cpp
import mymodule;
import <string>;
import <iostream>;

int main()
{
    Color lucky_color = GREEN;
    PrintColor(lucky_color);

    Person Rushikesh("Rushikesh");
    std::cout << Rushikesh.GetName();

    some_internal_function(); // Compilation error.
    return 0;
}
```

An example of Module

How to export import statements in module interface file?

```cpp
export module mymodule;
export import <string>;

export
{
    enum Color {RED, GREEN, BLUE};

    void PrintColor(Color color);

    class Person
    {
        std::string name;

        public:
            Person(const std::string& iName);
            std::string GetName();
    };
}
```

```cpp
import mymodule;
import <string>; // No need to import
import <iostream>;

int main()
{
    Color lucky_color = GREEN;
    PrintColor(lucky_color);

    Person Rushikesh("Rushikesh");
    std::cout << Rushikesh.GetName();

    some_internal_function(); // Compilation error.
    return 0;
}
```

Modules

- Export keyword appears only in module interface file.
- Only exported functions are made available for users of the module.
- Non exported functions/symbols declared in module interface file are not made available to users of module. They are available within the module i.e. module implementation file.
- Macros, static functions are not allowed to be exported.
- Extension of module interface file (.ixx for example) depends on compiler implementation.
- Compiler generates a binary file for module and stores it on disk. This file is used during compilation of files that import the module.
- During build, because compiler need not push everything like #include does, it results in improvement of compile, build time.
- Modules and namespaces are orthogonal concepts.

Modules

C++20 provides following 2 ways to manage larger modules:

1. **Submodules**
2. **Module partitions**

Submodules

```
export module mymath;

export import mymath.point;
export import mymath.vector;
export import mymath.transformation;
```

```
export module mymath.point;

export class Point
{
    // class declaration
};
```

```
export module mymath.vector;

export class Vector
{
    // class declaration
};
```

```
export module mymath.transformation;

export class Transformation
{
    // class declarion
};
```

- For simplicity I omitted module implementation files here.
- Note that the name of submodule for example `mymath.point` is chosen for the sake of readability purposes and "." here does not have any special meaning.
- We could have chosen `point` or `mypoint` as the name of submodule.

Submodules

- If the interface for a module is big with lots of functions, classes etc. and if it can be split into parts then submodules is one option.
- Technically from compiler's point of view submodule is like any other module.
- C++ 20 allows "." character to be used in module names. However it can't be the first character.
- Submodules use "." character in their names.
- Submodules are visible outside the module.
- Submodules facilitate users to import only required pieces of the module. This helps in reduced build impact and improved build time.

Module implementation partitions

```cpp
export module car;
import :engine;
import :carburetor;
import :differential_box;

export class Car
{
  public:
    Car();
    ~Car();

    Start();
    Stop();
    Accelerate();

  private:
    Engine engine;
    Carburetor carb;
    DifferentialBox dbox;
};
```

```cpp
module car:engine;

class Engine
{
  // Engine class definition.
};
```

```cpp
module car:carburetor;

class Carburetor
{
  // Carburetor class definition.
};
```

```cpp
module car:differential_box;

class DifferentialBox
{
  // DifferentialBox class definition.
};
```

Module implementation partitions

- They do not contain export keyword under them.
- They do not have module interface associated with them.
- One module implementation partition can exist in only one file. It is not possible to have multiple files with same module partition.
- Partitions are visible only within the module. They can be imported only within the module. It is not possible to import partitions outside the module.
- Users can import the main module only. The main module presents itself as a single module to its users.
- If need be we can entirely change the partitions without affecting the users of module.

Transition from #include to modules

```cpp
module; // global module fragment

#include "header1.h"
#include "header2.h"
// ...

/*export*/ module mymodule;

import <vector>
import <string>
import myothermodule;

// module code
```

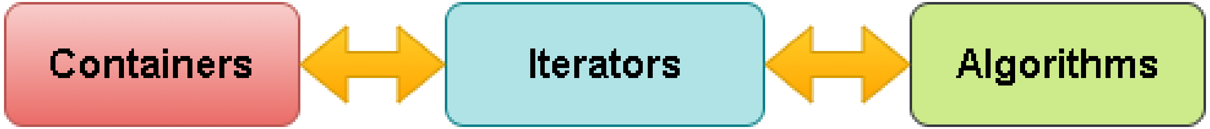
- The section between `module;` and `module mymodule;` is called global module fragment. Only preprocessor directives can appear here. No function or variable declarations are allowed here.
- The section after the statement: `module mymodule;` is called module purview. No preprocessor directives are allowed here.

Ranges

- Containers, iterators, algorithm design in C++ STL.
- Issues with existing algorithm functions.
- Ranges.
- Views.

Ranges

Containers, iterators, algorithm design in C++



vector
set
list
forward_list

Random access += -= ++ --
bidirectional ++ --
forward ++
input
output

sort
find
find_if
min_element
max_element
copy

- The reason that data structures and algorithms work together seamlessly in C++ is because they do not know anything about each other.

Ranges

Existing algorithm functions

- Most of the algorithm functions take two inputs: begin and end iterators:

```
sort(vec.begin(), vec.end())
find(vec.begin(), vec.end(), object)
find_if(vec.begin(), vec.end(), function_object)
min_element(vec.begin(), vec.end())
```

- The interface for these algorithm functions is less intuitive.
- Compiler does not warn even if the two iterators by mistake happen to be from two different container objects. In such cases the behavior is undefined.

Ranges

- C++ 20 introduces new namespace: `std::ranges`
- All the algorithm functions are also available in `std::ranges` namespace that take input as a range, for example:

  ```
  std::ranges::sort(some_range)
  ```

  `some_range` can be `std::vector` object for example.

  Note: Input can't be r-value or temp object. Following code is invalid.

  ```
  std::ranges::sort(create_vector())
  ```

- Range is an abstraction for collection of items that can be iterated.
- Anything that provides iterators and has `begin(), end()` functions is a range.
- So all the containers are ranges because they provide `begin(), end()` functions.

Ranges

```cpp
import <vector>;
import <span>;
import <algorithm>;

std::vector<int> v{4, 2, 3, 1, 8, 5, 7, 6};
std::span<int> s(v.begin()+2, v.begin()+6);
std::ranges::sort(s);
```

- std::span can be used to create a range with desired begin, end.
- std::ranges library functions accept span object as input.

Ranges: filter views

```cpp
std::vector<int> numbers {1, 2, 3, 4, 5, 6, 7, 8};
auto even = [] (int n)->bool { return n % 2 == 0; };
auto even_nums = numbers | std::ranges::views::filter(even);
```

- C++ 20 provides pipe operator “|” on ranges. Its usage is similar to Unix pipe operator.
- `std::ranges::views::filter()` is a range adaptor. It takes range, function object as input and produces view as output.
- Views are light weight objects. Creation, copy, move of views requires constant time independent of the number of elements.
- Views are lazily evaluated. For example, actual evaluation happens only after the call to `even_numbers.begin()` happens.

Ranges: transform views

```cpp
std::vector<int> numbers {1, 2, 3, 4, 5, 6, 7, 8};
auto square = [] (int n)->int { return n*n; };
auto sq_nums = numbers | std::ranges::views::transform(square);
```

- `std::ranges::views::transform()` is a range adaptor that takes a range object and transform function object as input and produces the view as output.
- We can combine filter, transform as follows:

```cpp
std::vector<int> numbers {1, 2, 3, 4, 5, 6, 7, 8};
auto even = [] (int n)->bool { return n % 2 == 0; };
auto square = [] (int n)->int { return n*n; };
auto sq_nums = numbers | std::ranges::views::filter(even) | std::ranges::views::transform(square);
for(int n: sq_nums)
  cout << n << " ";
```

Concepts

- Issues with current templates: long error messages.
- What concepts are and how to define them.
- Defining new concepts in terms of existing concepts.

Concepts

```cpp
#include <vector>

class MyClass
{
};

int main()
{
  MyClass mc1, mc2;
  MyClass& mx = std::max(mc1, mc2);

  return 0;
}
```

- This program does not compile because there is no < operator implemented for MyClass.
- Compiler error message in this case is relatively short and easy to understand.

Concepts

Issue with templates: long error messages

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

class MyClass
{
};

int main()
{
    MyClass mc1, mc2;
    std::vector<MyClass> v{mc1, mc2};
    std::sort(v.begin(), v.end());
}
```

- This program does not compile because there is no < operator implemented for MyClass.
- But in this case compiler reports near about 100 lines of error messages.
- It is not easy to figure out the reason for compilation failure in this case.
- This happens for template functions, classes.

Concepts

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

class MyClass
{
};

int main()
{
  MyClass mc1, mc2;
  std::vector<MyClass> v{mc1, mc2};
  std::ranges::sort(v.begin(), v.end());
}
```

- If we use the sort algorithm function from ranges namespace, then, we will get short and easy to understand compilation failure message.
- This is because the ranges::sort function uses concepts.
- All the algorithm functions from ranges namespace make use of concepts.

Concepts

How to define our own concept

- There are multiple ways in which the constraints can be specified for templates.
- This example shows one way to specify the constraints in templates using concepts.
- C++ standard library has lots of predefined concepts. So before writing our own concepts it is better to check these.

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

class MyClass { };

class Person {
  unsigned int age;
  public:
  bool operator < (const Person& p) {
    return age < p.age;
  }
};

template<class T>
concept Comparable = requires(T a, T b) {
  {a < b} -> std::convertible_to<bool>;
};

bool isLess(Comparable auto& c1, Comparable auto& c2) {
  return c1 < c2;
}

int main() {
  Person p1, p2;
  bool is_less = isLess(p1, p2);

  MyClass mc1, mc2;
  is_less = isLess(mc1, mc2);

  return 0;
}
```

Concepts

- New concepts can be defined using existing concepts.
- This example shows how Sortable container concept is defined in terms of RandomAccessIterator concept, which again is defined using BidirectionalIterator concept.
- Whenever possible define your own concepts in terms of already existing concepts from standard library.

```
template<class Iter>
concept BidirectionalIterator = requires(Iter it)
{
  {++it} -> std::same_as<Iter>;
  {--it} -> std::same_as<Iter>;
};

template<class Iter>
concept RandomAccessIterator = BidirectionalIterator<Iter> &&
requires (T a, T b, const int n)
{
  {a+= n} -> std::same_as<T&>;
  {a-= n} -> std::same_as<T&>;
  {a-b} -> std::same_as<int>;
  {a[n]} -> std::same_as<decltype(*a)>;
};

template<class Container>
concept Sortable = requires (Container c, const int i, const int j)
{
  {c.begin()} -> RandomAccessIterator;
  {c[i] < c[j]} -> std::convertible_to<bool>;
};
```

<=> Three way comparison operator

- In C language, the signature of comparison function for qsort is as follows:

  int compare(void* left, void* right)

  return value 0 indicates left == right

  1 means left > right

  -1 means left < right

- In C++ we have operator overloading functions.
- Once we implement one operator for our class, users expect all other operators, otherwise it won't be intuitive.
- `<, >, ==, <=, >=, !=` these are the comparison operators. If one of these is available, then, it is better to have all of these.
- It is possible to implement `>, <=, >=, !=` in terms of `<` and `==` operators. Even though code reuse is possible, but this bloats the class with number of operator functions.
- C++ 20 introduces `<=>` three way comparison operator also referred to as spaceship operator.

<=> Three way comparison operator

- The result of <=> operator can be one among following 3:
  - `std::strong_ordering`
  - `std::weak_ordering`
  - `std::partial_ordering`
- Some examples for int built in type:

  `3 <=> 3` evaluates to `std::strong_ordering::equal`

  `3 <=> 4` evaluates to `std::strong_ordering::less`

  `3 <=> -1` evaluates to `std::strong_ordering::greater`
- `int a = 3, b = 4;`

  `if(a <=> b < 0)` // same as `a < b` and it evaluates to true in this case.

  `if(a <=> b > 0)` // same as `a > b` and it evaluates to false in this case.

<=> Three way comparison operator

- If all the members of a class have `<=>` operator implemented, then, the compiler can provide the default implementation for `<=>` operator for class.
- All the comparison operators (`<`, `>`, `<=`, `>=`, `==`, `!=`) will also be implemented automatically by compiler in terms of `<=>` operator.

```cpp
class Record
{
  public:

    auto operator<=>(const Record&) const = default;

  private:
    std::string message;
    int id;
    double weight;
};
```

Designated initiazers

- Limitations:
  - Out of order member initialization not possible.
  - Initialization of nested aggregates not possible.

```
struct Point2D
{
    int x;
    int y;
};

class Point3D
{
public:
    int x;
    int y;
    int z;
};

int main()
{
    Point2D point2D{.x = 1, .y = 2.5};
    Point3D point3D{.x = 1, .y = 2, .z = 3.5f};
}
```

format

- `std::cout` from C++ is good at type safety, concise syntax. But it is difficult to format text with `cout`.
- `printf` from C is good at formatting the output. But it lacks type safety.
- C++ 20 introduces `std::format` on similar lines of python language.

```cpp
import <iostream>;
import <format>;

int main()
{
    int i1 = 1, i2 = 2;

    std::cout << std::format("Parameters without indices = {}, {}", i1, i2); // outupts 1, 2
    std::cout << std::format("Parameters with indices specified: {0}, {1}", i1, i2); // outupts 1, 2
    std::cout << std::format("Parameters in reverse order: {1}, {0}", i1, i2); // outupts 2, 1
    std::cout << std::format("Same parameter appearing multiple times: {0}, {0}", i1); // outupts 1, 1

    return 0;
}
```

format

```cpp
import <iostream>;
import <format>;

int main()
{
  int i = 196;
  std::cout << std::format("Binary representation of {0} = {0:b}\n", i); // 0b11000100
  std::cout << std::format("Hexadecial representation of {0} = {0:x}\n", i); // 0xc4

  const double pi = std::numbers::pi;
  std::cout << std::format("pi value with 2 digits after the fraction = {:.2f}", pi); // 3.14

  return 0;
}
```

<numbers>

- C++ 20 provides new header `<numbers>` with some mathematical constants defined under `std::numbers` namespace.
- These are variable templates (introduced in C++14) with default type as double.
- If someone wants the values of these constants in float for example, then, one can use `pi_v<float>` for example.

```
const float pif = std::numbers::pi_v<float>;
```

```
namespace std::numbers
{
    inline constexpr double pi = pi_v<double>;
    inline constexpr double inv_pi = inv_pi_v<double>;

    inline constexpr double e = e_v<double>;
    inline constexpr double log2e = log2e_v<double>;
    inline constexpr double log10e = log10e_v<double>;

    inline constexpr double ln2 = ln2_v<double>;
    inline constexpr double ln10 = ln10_v<double>;

    inline constexpr double sqrt2 = sqrt2_v<double>;
    inline constexpr double sqrt3 = sqrt3_v<double>;
}
```

References

- C++ 20

https://www.youtube.com/watch?v=FRkJCvHWdwQ

https://en.cppreference.com/w/cpp/20

https://en.wikipedia.org/wiki/C%2B%2B20

- Modules

https://isocpp.org/files/papers/n4214.pdf

https://www.youtube.com/watch?v=Kqo-jIq4V3I

https://www.youtube.com/watch?v=tjSuKOz5HK4

- Ranges

https://www.youtube.com/watch?v=d_E-VLyUnzc

- Coroutines

https://meetingcpp.com/mcpp/slides/2019/C20%20Coroutines.pdf

IF WE™