C++ 20 Language Features

Contents

[a) Core Language Features 2](#_Toc190981576)

[1. Concepts 2](#_Toc190981577)

[2. Ranges Library 4](#_Toc190981578)

[3. Modules – A Modern Alternative to Header Files 7](#_Toc190981579)

[4. Three-way Comparison 10](#_Toc190981580)

[5. Designated Initializers – Explicit member initialization 16](#_Toc190981581)

# Core Language Features

### Concepts

* What are Concepts?
  + Concepts are a way to define **constraints** on template parameters. They specify what requirements a type must meet to be used with a template.
  + Think of them as a set of rules that types must follow to be accepted by a template.
* Why use Concepts?
  + Before Concepts, **template error messages could be very confusing and difficult to understand**. Concepts make templates easier to read and use by providing clear constraints on what types can be passed to them.
  + They improve code readability and give more informative error messages when something goes wrong.
  + Concepts allow us to **enforce constraints on template parameters**, making errors **clearer and more readable**.

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include <concepts> // Include for concepts  // Define a concept "HasSize" that ensures T has a .size() method  template <typename T>  concept HasSize = requires(T t) { t.size(); };  void printSize(const HasSize auto& obj) {  std::cout << "Size: " << obj.size() << '\n';  }  int main() {  std::string str = "Hello";  printSize(str); // Works because std::string has .size()  std::vector<int> vec = { 1, 2, 3, 4 };  printSize(vec); // Works because std::vector has .size()  int num = 42;  // Compilation error: int does not satisfy HasSize  // printSize(num);  } |

#### Breaking It Down

* template <typename T> - **Template for a Concept**
  + Just like a function or class template, concepts can take type parameters.
  + Here, T is a placeholder for any type that might be passed to a template function or class.
* concept HasSize = ...; - **Concept Definition**
  + HasSize is the name of the concept.
  + A concept is like a Boolean condition: it evaluates to true or false depending on whether the type T satisfies the given requirements.
* requires(T t) { ... }; - **Requires Clause**
  + This is where we define what requirements the type T must fulfil.
  + (T t): This introduces a dummy variable t of type T to check expressions on it.
* t.size(); - **Expression Requirement**
  + This checks if the expression t.size() is valid.
  + If the type T has a .size() method, the concept is satisfied (true).
  + If T does not have a .size() method, **compilation fails**.

#### Built-in Concepts in C++20

* C++20 also comes with some pre-defined concepts in the <concepts> header. For example:
  + std::integral<T>: Checks if T is an integer type (e.g., int, long).
  + std::floating\_point<T>: Checks if T is a floating-point type (e.g., float, double).
  + std::same\_as<T, U>: Checks if T and U are the same type.
  + std::convertible\_to<T> : It checks if a type can be implicitly converted to another type T.

#### Some more examples:

* Concept to check whether 2 objects can be added

|  |
| --- |
| template <typename T>  concept Addable = requires(T a, T b) {  // Checks if a + b is valid and returns a T  { a + b } -> std::same\_as<T>;  }; |

* Swappable Types

|  |
| --- |
| template <typename T>  concept Swappable = requires(T a, T b) {  std::swap(a, b);  };  template<Swappable T>  void mySwap(T& a, T& b) {  std::swap(a, b);  } |

* Printable Types

|  |
| --- |
| template<typename T>  concept Printable = requires(T a) {  { std::cout << a };  };  template<Printable T>  void print(T&& a) {  cout << a;  } |

### Ranges Library

* The Ranges Library in C++20 provides a modern way to work with collections (like std::vector, std::list, std::array, etc.).
* It improves how we:
  + Iterate over containers.
  + Filter or transform elements.
  + Make code more readable and efficient.
* Imagine you have a vector of numbers and you want to find even numbers.
* Before C++20:

|  |
| --- |
| #include <algorithm>  int main() {  std::vector nums = { 1, 2, 3, 4, 5, 6 };  std::vector<int> evens;  std::copy\_if(nums.begin(), nums.end(),  std::back\_inserter(evens),  [](auto n) { return n % 2 == 0; });  for (auto n : evens) {  std::cout << n << " ";  }  } |

* Problem:
  + Uses iterators (begin(), end()), std::copy\_if, and a manual insertion mechanism (std::back\_inserter).
  + Harder to Read!
* With C++20 Ranges, the same example becomes much cleaner:

|  |
| --- |
| #include <iostream>  #include <vector>  #include <ranges>  int main() {  std::vector<int> nums = { 1, 2, 3, 4, 5, 6 };  // Use std::views::filter to get only even numbers  auto evens = nums | std::views::filter(  [](int n) { return n % 2 == 0; });  for (int n : evens) {  std::cout << n << " ";  }  } // 2 4 6 |

#### What Changed?

* No need for iterators (begin() / end()).
* Pipes (|) make it readable like a data pipeline.
* std::views::filter directly filters the collection.
* No need for an extra std::vector<int> to store results.
* Lazy evaluation (processed only when needed).

auto evens = nums | std::views::filter(

[](int n) { return n % 2 == 0; });

#### Breakdown

* The object ‘evens’ is a **range**! It's a **filtered view** over the original range nums.
* nums: This is our original range or container holding elements. It could be any range-compatible container like std::vector, std::list, or even another view.
* C++20 introduces **views**, which allow you to **process collections without creating copies**. A view is a non-owning transformation of a range.
* std::views::filter is a **range adaptor that creates a view** that filters the nums range, keeping only the elements that satisfy the given lambda function.

#### What Are Range Adaptors?

* Range adaptors are tools that allow you to create views over ranges. A view is a lightweight, non-owning representation of a range that can be transformed, filtered, or otherwise manipulated without modifying the underlying data. **Range adaptors** are lazy, meaning they don’t perform any work until you iterate over the view.
* C++20 **Range Adaptors** or **View Adaptors**:
  + std::views::filter - Filtering Elements
  + std::views::transform - Transforming Elements
  + std::views::take - Taking the First N Elements
  + std::views::reverse - Reversing a Collection
* The pipe symbol | is used to compose range transformations. It takes the nums range and applies a transformation to it.

#### Combining Multiple Views

|  |
| --- |
| #include <iostream>  #include <vector>  #include <ranges>  int main() {  std::vector<int> nums = { 1, 2, 3, 4, 5, 6 };  auto evens\_squared = nums  | std::views::filter([](int n) { return n % 2 == 0; })  | std::views::transform([](int n) { return n \* n; });  for (int n : evens\_squared) {  std::cout << n << " ";  }  }// Output: 4 16 36 |

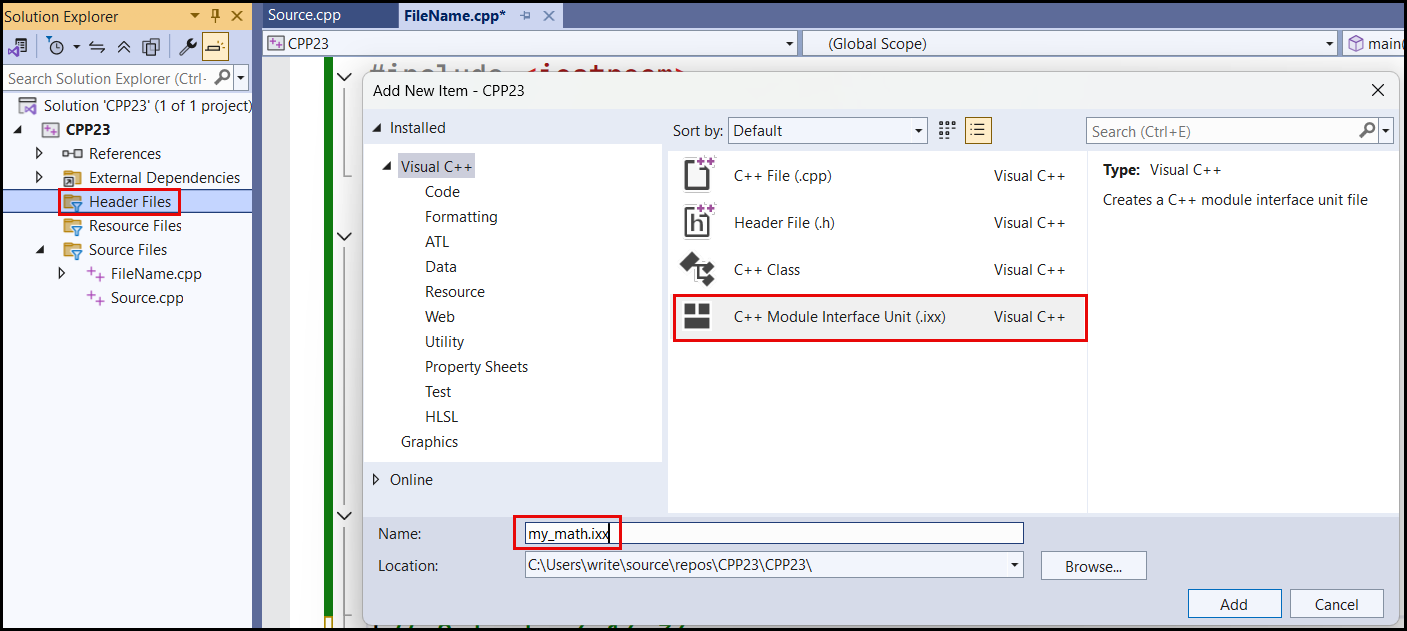
* Filters even numbers → Squares them → Prints them.
* No extra copy of nums is made.
* More readable than the std::algorithm approach.

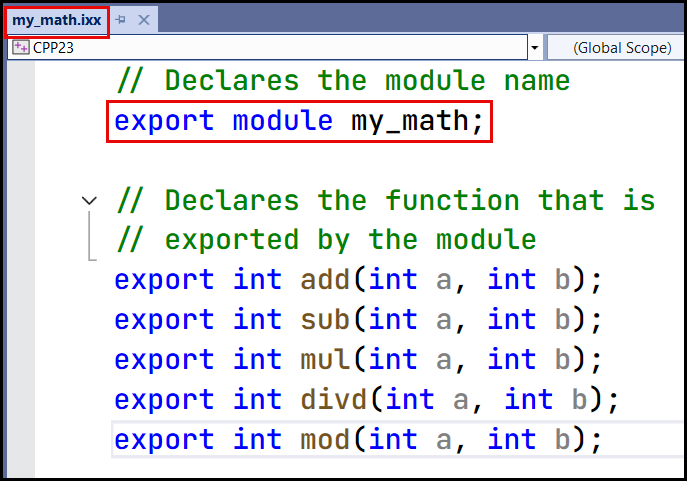
### Modules – A Modern Alternative to Header Files

#### The Problem with Headers

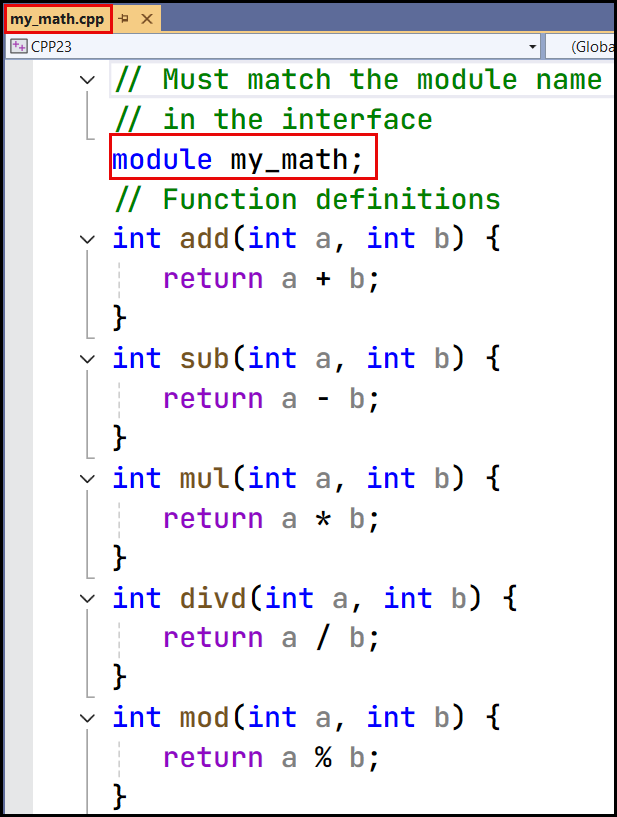
* In traditional C++, we include header files using #include. These header files contain **declarations** of **functions**, **classes**, and other things we want to use in our code. While this works, it has some drawbacks, especially in large projects:
  + **Redundancy**: The same header file might be included in many different source files. The compiler has to process the contents of that header file every single time it's included, even if the contents haven't changed.
  + **Order Dependence**: The order in which you include header files can sometimes matter. If a header file relies on something declared in another header file, you have to include them in the correct order. This is a common source of errors.
  + **Name Collisions**: If two header files declare something with the same name, you can get compiler errors (name clashes).
  + **Slow Compilation**: Because of the redundancy and order dependence, compiling large projects can take a long time.
* C++20 introduced **modules**, which replace header files and improve compilation speed significantly.
  + With **modules**, we can replace header files with **compiled modules**, which the compiler processes **only once**!

#### How Modules Work

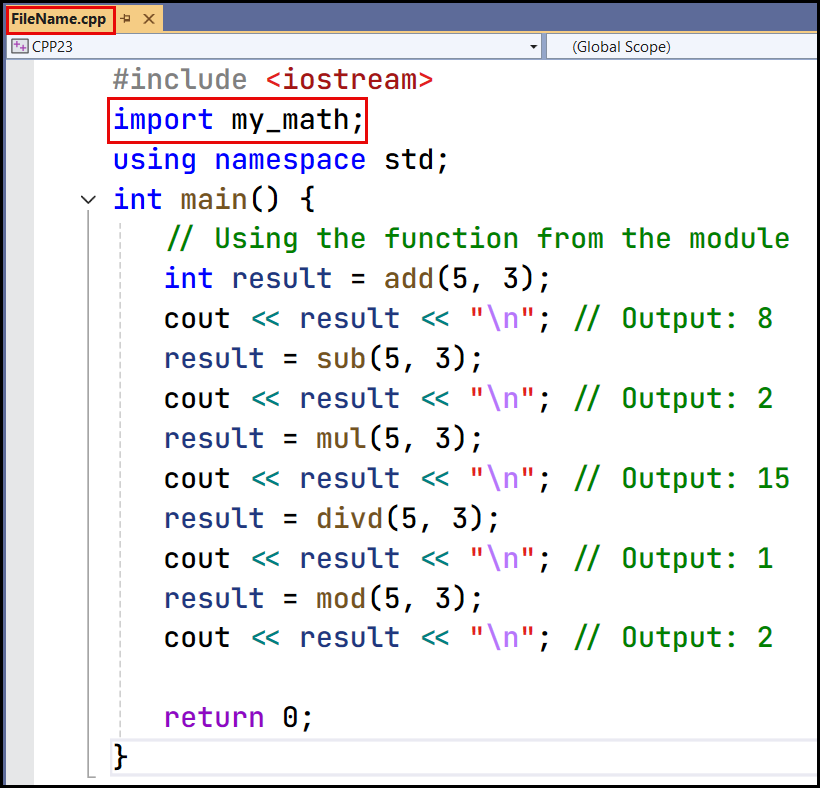
1. **Module Definition**:
   * Instead of a header file, we create a module interface file (typically with a .ixx or .cppm extension).
   * This file declares what the module provides to the outside world (functions, classes, etc.).
   * We then have a corresponding module implementation file (typically a .cpp file) that contains the actual **definitions** of those things.
   * Example:
     1. Module Interface - my\_math.ixx



* + 1. Module Implementation – my\_mapth.cpp



* + 1. Using the Module – FileName.cpp



#### **Summary**

| **Feature** | **Header Files** | **C++20 Modules** |
| --- | --- | --- |
| Uses #include? | ✅ Yes | ❌ No |
| Uses import? | ❌ No | ✅ Yes |
| Needs #ifndef guards? | ✅ Yes | ❌ No |
| Compilation speed | 🐢 Slow | 🚀 Fast |
| Multiple definitions issue | ❌ Yes | ✅ No |
| Code organization | ❌ Messy | ✅ Clean |

### Three-way Comparison

#### The Problem

* Before C++20, if we wanted to make our custom classes comparable, we had to define all six comparison operators manually:
  + == (equal to)
  + != (not equal to)
  + < (less than)
  + > (greater than)
  + <= (less than or equal to)
  + >= (greater than or equal to)
* For example: Let’s say we have a class Point

|  |
| --- |
| class Point {  int x, y;  public:  Point(int x, int y) : x(x), y(y) {}  bool operator==(const Point& other) const {  return x == other.x && y == other.y;  }  bool operator!=(const Point& other) const {  return !(\*this == other);  }  bool operator<(const Point& other) const {  return (x < other.x) || (x == other.x && y < other.y);  }  bool operator>(const Point& other) const {  return other < \*this;  }  bool operator<=(const Point& other) const {  return !(\*this > other);  }  bool operator>=(const Point& other) const {  return !(\*this < other);  }  }; |

* Problems With This Approach:
  + We have to manually write all six operators.
  + Code duplication (some operators are just the negation of others).
  + If we add a new member variable, we need to update all six functions.

#### Solution: The <=> Operator (Three-Way Comparison)

* C++20 introduced the **spaceship operator (**<=>**)**, which **automatically generates all six comparison operators** for us!

|  |
| --- |
| #include <iostream>  #include <compare> // Required for <=> operator  class Point {  int x, y;  public:  Point(int x, int y) : x(x), y(y) {}  // Three-way comparison  auto operator<=>(const Point& other) const = default;  };  int main() {  Point p1(3, 4), p2(5, 4);  std::cout << std::boolalpha;  std::cout << (p1 == p2) << "\n"; // false  std::cout << (p1 != p2) << "\n"; // true  std::cout << (p1 < p2) << "\n"; // true  std::cout << (p1 > p2) << "\n"; // false  std::cout << (p1 <= p2) << "\n"; // true  std::cout << (p1 >= p2) << "\n"; // false  } |

#### How Does <=> Work?

* The <=> operator **returns a special comparison result** that tells whether the left-hand side (lhs) is:
  + less than (<)
  + equal to (==)
  + greater than (>)
* It returns a **value of type** std::strong\_ordering, std::weak\_ordering, or std::partial\_ordering from <compare>.

#### Different Types of Ordering in <=>

* std::strong\_ordering, std::weak\_ordering, and std::partial\_ordering are collectively called **comparison category types** (or sometimes just **ordering types**). They are defined in the <compare> header.

##### std::strong\_ordering (For Things That Are Clearly Ordered)

* Imagine you have a bunch of apples. You can easily say if one apple is bigger than another, smaller, or exactly the same size. There's no ambiguity. This is what std::strong\_ordering is for.
  + **Meaning**: Every value is clearly less than, greater than, or equal to every other value. There are no "in-between" or "uncomparable" cases.
  + **Example**: Integers, characters, and pointers are typically strongly ordered.

|  |
| --- |
| #include <compare>  int main() {  int a = 5;  int b = 10;  auto result = a <=> b; // result is a strong\_ordering  if (result == strong\_ordering::less) {  cout << "a is less than b" << endl;  }  else if (result == strong\_ordering::greater) {  cout << "a is greater than b" << endl;  }  else if (result == strong\_ordering::equal) {  cout << "a is equal to b" << endl;  }  }// Output: a is less than b |

* In this example, a <=> b returns strong\_ordering::less because 5 is clearly less than 10. There's no other possibility.

#### std::weak\_ordering (Equality is Clear, But Ordering Might Be Fuzzy)

* Imagine you have a bunch of shirts. You can say if two shirts are the same size, but the style or color might make it hard to say definitively if one is "greater" than the other in some overall sense. You can compare the sizes, but other features are not easily comparable. This is std::weak\_ordering.
  + **Meaning**: Equality is well-defined (you can say if two things are the same), but the ordering might not be consistent or meaningful for all aspects.
  + **Example**: Comparing objects based on a single member variable, while ignoring other members.

|  |
| --- |
| #include <iostream>  #include <compare>  using namespace std;  struct Person {  string name;  int age;  auto operator<=>(const Person& other) const {  return age <=> other.age; // Compare only by age  }  };  int main() {  Person p1{ "Alice", 25 };  Person p2{ "Bob", 25 }; // Same age  Person p3{ "Charlie", 30 };  auto result1 = p1 <=> p2;  auto result2 = p1 <=> p3;  // Output: p1 and p2 have the same age  if (result1 == weak\_ordering::equivalent) {  cout << "p1 and p2 have the same age" << endl;  }  // Output: p1 is younger than p3  if (result2 == weak\_ordering::less) {  cout << "p1 is younger than p3" << endl;  }  } |

Why the operator<=> returns weak\_ordering?

* age <=> other.age returns std::strong\_ordering, since int is strongly ordered.
* However, since operator<=> does not define operator== explicitly, C++ automatically generates operator== for Person.
* The problem is that Person contains a std::string (name), and std::string provides only weak\_ordering.
* This weak ordering of std::string affects the entire struct's ordering!

#### std::partial\_ordering (Things That Might Be Uncomparable)

* Imagine comparing the areas of two shapes. If the shapes are simple (like squares), you can easily compare their areas. But what if the shapes are complex and overlapping? You might not be able to definitively say which one has a larger area. This is std::partial\_ordering.
  + **Meaning**: Some values might be completely incomparable.
  + **Example**: Floating-point numbers, especially when dealing with NaN (Not a Number) values.

|  |
| --- |
| #include <iostream>  #include <compare>  #include <cmath> // For std::nan  int main() {  double a = 10.5;  double b = std::nan(""); // NaN represents an undefined value.  auto result = a <=> b; // result is a std::partial\_ordering  if (result == std::partial\_ordering::unordered) {  std::cout << "a and b are unordered" << std::endl;  }  } |

#### How to use Three-Way Comparison <=>?

|  |
| --- |
| #include <iostream>  #include <compare>  class Point {  int x, y;  public:  Point(int x, int y) : x(x), y(y) {}  // Custom three-way comparison  auto operator<=>(const Point& other) const {  if (auto cmp = x <=> other.x; cmp != 0)  return cmp;  return y <=> other.y;  }  bool operator==(const Point& other) const = default;  };  int main() {  Point p1(3, 4), p2(3, 5);  // true, because (3,4) < (3,5)  std::cout << (p1 < p2) << "\n";  } |

* Explanation:
* auto operator<=>(const Point& other) const**:** This line declares the spaceship operator for the Point struct.
* if (auto cmp = x <=> other.x; cmp != 0):
  + auto cmp = x <=> other.x;: This performs a three-way comparison between the x members of the two Point objects. The result of this comparison is stored in the variable cmp. The type of cmp will be the appropriate comparison category type based on the type of x (e.g. if x is an int, cmp will be a std::strong\_ordering).
  + cmp != 0: This checks if cmp is not equal to 0. In the context of comparison category types, a non-zero value means that the x coordinates are different (either less than or greater than). A zero value means the x coordinates are equal.
* return cmp;**:** If cmp is not 0 (meaning the x coordinates are different), this line immediately returns the result of the x comparison. This is because if the x coordinates are different, there's no need to compare the y coordinates. The comparison is already decided.
* return y <=> other.y;**:** If the if condition is false (meaning the x coordinates are equal), this line performs a three-way comparison between the y coordinates and returns the result. This is the tie-breaker: if the x coordinates are the same, the comparison is determined by the y coordinates.
* We have defined bool operator==(const Point& other) const = default; Does operator<=> need operator==?
  + No, operator<=> (the spaceship operator) does not technically need operator== to function. The spaceship operator can be defined independently. However, the C++ standard strongly encourages (and practically requires in many cases) that if you define operator<=>, you also define operator== (even if it's just = default).

### Designated Initializers – Explicit member initialization

#### The Problem: Initializing Structs Can Be Error-Prone

* Before C++20, you would typically initialize structs (or aggregates) using aggregate initialization. This means providing the values in the order the members are declared in the struct:

|  |
| --- |
| struct Person {  std::string name;  int age;  double height;  };  int main() {  Person person1 = { "Alice", 30, 5.8 }; // Aggregate initialization  Person person2 = { "Bob", 25 }; // Aggregate initialization (missing height)  Person person3 = { 20, "Charlie", 6.2 }; // Incorrect order!  // ...  } |

* Problems with this approach:
  + **Order-dependent**: If you **add/remove members**, you must update all initializations.
  + **Less readable**: It’s unclear what each value represents just by looking at {...}.

#### Solution: Designated Initializers

* With **designated initializers**, you can **explicitly specify which members to initialize**, making the code **more readable** and **order-independent**.

|  |
| --- |
| #include <iostream>  struct Person {  std::string name;  int age;  double height;  };  int main() {  // Explicit initialization  Person p1{ .name = "Alice", .age = 25, .height = 5.6 };  std::cout << p1.name << " is " << p1.age << " years old.\n";  } |

#### **Summary**

| **Feature** | **Explanation** |
| --- | --- |
| **Before C++20** | Must initialize struct/class members **in order**, & is maintenance hard. |
| **C++20 Designated Initializers** | Allows **explicitly initializing** members **by name**, in **any order**. |
| **Benefits** | More readable, Order-independent, Allows partial initialization. |
| **Limitations** | Only for **aggregates** (no constructors/inheritance/virtual functions). |