C++ 20 Language Features

Contents

[a) Core Language Features 2](#_Toc191067584)

[1. Concepts 2](#_Toc191067585)

[2. Modules – A Modern Alternative to Header Files 4](#_Toc191067586)

[3. Three-way Comparison 8](#_Toc191067587)

[4. Designated Initializers – Explicit member initialization 14](#_Toc191067588)

[5. Lambda Improvements 15](#_Toc191067589)

[6. Coroutines 19](#_Toc191067590)

[7. constexpr Improvements 21](#_Toc191067591)

[8. Immediate Functions (consteval) 25](#_Toc191067592)

[9. constinit Keyword 27](#_Toc191067593)

[10. New Attributes 29](#_Toc191067594)

[11. Class Default Member Initializers for Bit-fields 31](#_Toc191067595)

[b) Library Features 32](#_Toc191067596)

[12. Ranges Library 32](#_Toc191067597)

[13. The std::format 34](#_Toc191067598)

[14. The std::span 37](#_Toc191067599)

# 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

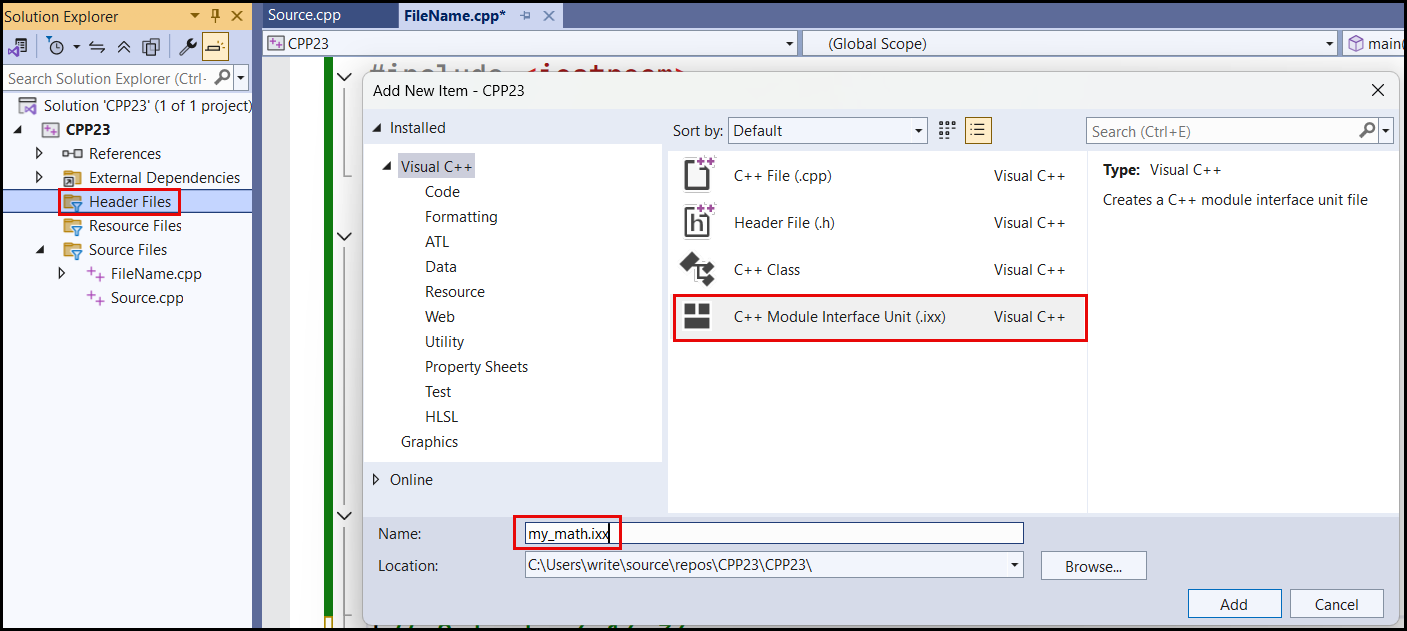
|  |
| --- |
| template<typename T>  concept Printable = requires(T a) {  { std::cout << a };  };  template<Printable T>  void print(T&& a) {  cout << a;  } |

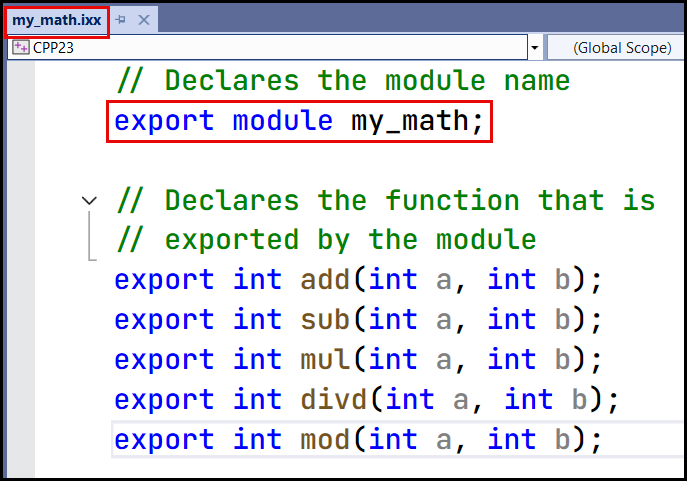
### Modules – A Modern Alternative to Header Files

#### The Problem with Headers

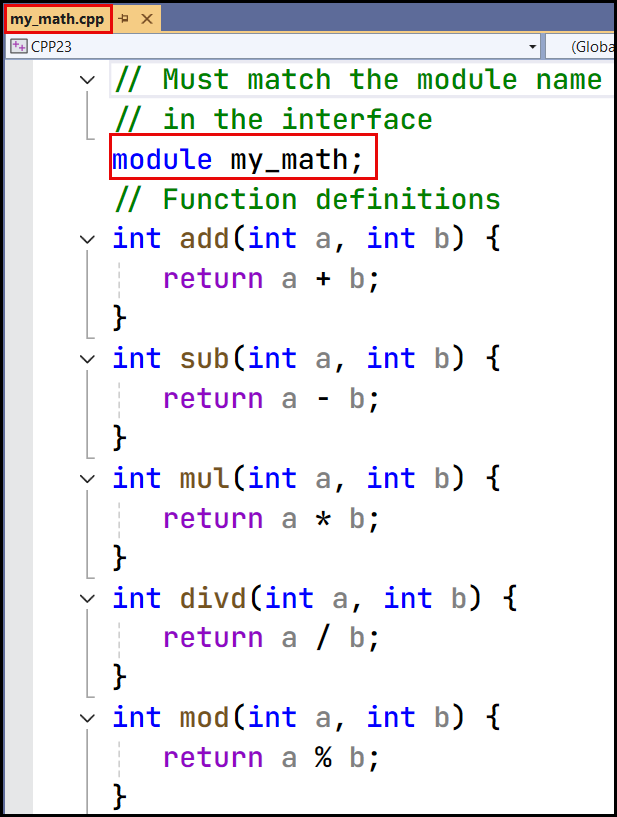
* In traditional C++, we include header files using #include. These header files contain **declarations** of **functions**, **classes**, and other things we want to use in our code. While this works, it has some drawbacks, especially in large projects:
  + **Redundancy**: The same header file might be 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

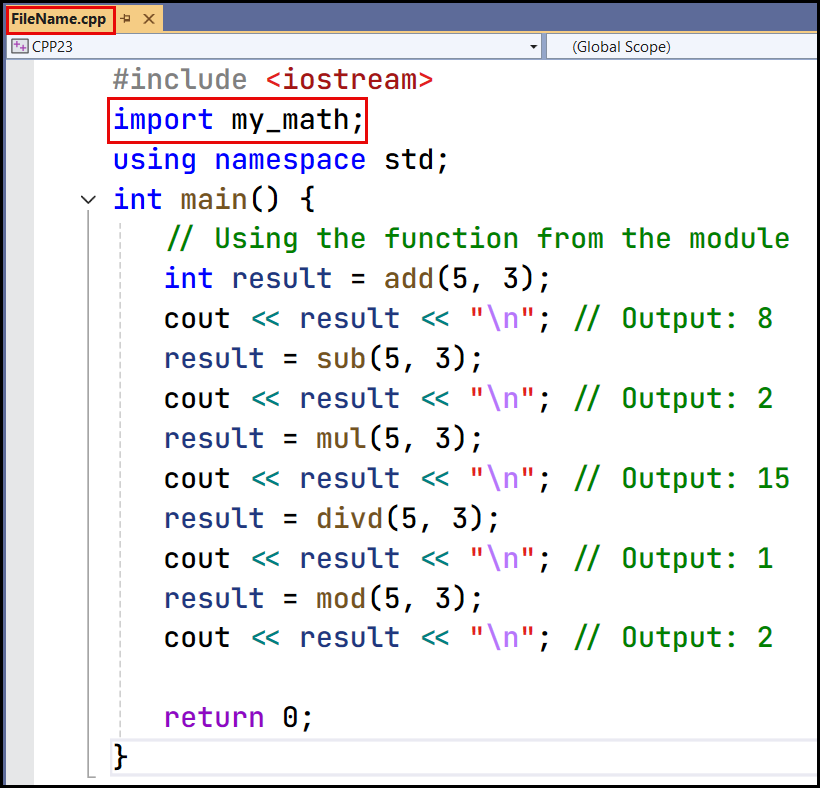
1. **Module Definition**:
   * Instead of a header file, we create a module interface file (typically with a .ixx or .cppm extension).
   * This file declares what the module provides to the outside world (functions, classes, etc.).
   * We then have a corresponding module implementation file (typically 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). |

### Lambda Improvements

* C++20 introduced **several improvements** to **lambdas**, making them **more powerful and flexible**.

#### Problem Before C++20

1. **No template lambdas** – You couldn’t write lambdas that worked with multiple types.
2. **No default-constructible lambdas** – Lambdas had no default constructor.
3. **No capturing this by value** – Capturing this by value required workarounds.

#### Generic Lambdas with Template Parameters

* Before C++20: No Support for Templated Lambdas
  + Before C++20, if you wanted to create a lambda that could handle multiple types, you had to rely on auto parameters. This approach worked, but it had limitations:

|  |
| --- |
| int main() {  // Works in C++14  auto add = [](auto a, auto b) { return a + b; };  std::cout << add(2, 3) << "\n"; // Output: 5  std::cout << add(2.5, 3.1) << "\n"; // Output: 5.6  } |

* The lambda add can accept arguments of any type, but you couldn't explicitly specify the types of a and b.
* This lack of explicit type specification could lead to **less readable code** and **potential type safety issues.**

#### C++20: Templated Lambdas

* C++20 introduces the ability to define **template parameters directly inside a lambda**.
* This allows you to explicitly specify the types the lambda should handle, providing more control and type safety.

|  |
| --- |
| int main() {  auto add = []<typename T>(T a, T b)  {  return a + b;  };  cout << add(2, 3) << "\n"; // Output: 5  cout << add(2.5, 3.1) << "\n"; // Output: 5.6  } |

* Advantages of Templated Lambdas in C++20:
  + **Explicit Type Specification**: In the lambda add, you can see that a and b are of the same type T.
  + **Type Safety**: If you accidentally pass arguments of different types, the compiler will catch it.

|  |
| --- |
| // Error: no matching function for call to 'add'  cout << add(2, 3.1) << "\n"; |

* C++20 Concepts:
  + C++20 also introduces concepts, which allow you to **constrain the types that can be used with templates**.

|  |
| --- |
| template<typename T>  concept Addable = requires(T a, T b) {  { a + b } -> std::same\_as<T>;  };  auto add = []<Addable T>(T a, T b) { return a + b; };  int main() {  cout << add(2, 3) << "\n"; // Output: 5  cout << add(2.5, 3.1) << "\n"; // Output: 5.6  } |

#### Lambdas Are Default-Constructible

* What Does "Default Constructible" Mean?
  + In C++, a type is default constructible if you can **create an instance of that type without providing any arguments**. For example:

|  |
| --- |
| struct MyStruct {  int value;  MyStruct() : value(0) {} // Default constructor  };  MyStruct obj{}; // Default-constructed object |

#### Default Constructibility of Lambdas

* Lambdas in C++ are essentially **objects of an anonymous class type** generated by the compiler. Whether a lambda is default constructible depends on its capture list.
* A **stateless lambda** is a lambda that **doesn't capture any variables** (i.e., its capture list [] is empty). Stateless lambdas have been **default constructible** since C++11.

|  |
| --- |
| auto f = [] { return 42; }; // Stateless lambda  decltype(f) g{}; // Default-constructed lambda |

* + This works because the compiler-generated class for a stateless lambda has a default constructor.
* Before C++20, **generic lambdas were not default constructible**, even if they were stateless. This was a limitation in the language.

|  |
| --- |
| auto f = [](auto x) { return x \* 2; }; // Generic lambda  decltype(f) g{}; // Error in C++14/C++17! |

* In C++20, this limitation was fixed. Now, **generic stateless lambdas are also default constructible**.

|  |
| --- |
| // Generic lambda with template  auto f = []<typename T>(T x) { return x \* 2; };  decltype(f) g{}; // Works in C++20! |

#### Capturing this by Value

* The Problem Before C++20: Capturing \*this:
  + Before C++20, if you wanted to use members of the current object (this) inside a lambda, you couldn't directly capture this by value. The only way to access members was to capture \*this (the dereferenced this pointer). This meant that the **entire object was copied into the lambda's closure**.
  + Example:

|  |
| --- |
| struct Counter {  int value = 0;  Counter() = default;  // Copy constructor  Counter(const Counter& other) {  value = other.value;  cout << "Counter copied!" << endl;  }  // Method that returns a lambda capturing \*this  auto getLambda() {  return [\*this] { return value; };  }  };  int main() {  Counter c;  c.value = 42;  auto lambda = c.getLambda();// Copy constructor  cout << "Lambda value: " << lambda() << endl;  return 0;  }  // Counter copied!  // Lambda value : 42 |

* Capture this by Value with [=, this]
  + Now, you can **capture** this **by value** while still capturing other variables by reference.

|  |
| --- |
| struct Counter {  int value = 0;  Counter() = default;  // Copy constructor  Counter(const Counter& other) {  value = other.value;  cout << "Counter copied!" << endl;  }  // Method that returns a lambda capturing \*this  auto getLambda() {  return [=, this] { return value; };  }  };  int main() {  Counter c;  c.value = 42;  auto lambda = c.getLambda();  cout << "Lambda value: " << lambda() << endl;  return 0;  }  // Lambda value: 42 |

#### **Summary of C++20 Lambda Improvements**

| **Feature** | **Before C++20** | **C++20 Improvement** |
| --- | --- | --- |
| **Templated Lambdas** | Only auto params | ✅ Supports explicit template params |
| **Default-Constructible Lambdas** | ❌ Not possible | ✅ Allowed if no captures |
| **Capturing** this **by Value** | Workaround using [ \*this ] | ✅ Now [=, this] is supported |

### Coroutines

* Coroutines are a powerful feature in C++20 that allow **functions to be paused and resumed** instead of running from start to finish in one go.
* Unlike regular functions, which execute sequentially, coroutines can:
  + Pause execution at certain points.
  + Return intermediate results.
  + Resume from where they left off.
* Example Analogy
  + Imagine watching a TV series on Netflix.
    - A normal function is like watching a movie – you watch it from start to end in one go.
    - A coroutine is like watching a TV series – you watch one episode, stop, and then resume later from the next episode.

#### Why Use Coroutines?

* Coroutines are useful in scenarios where waiting is involved, such as:
  + **Asynchronous Programming** → Handling network calls, file I/O, or user input without blocking the main thread.
  + **Lazy Evaluation** → Generating values on demand instead of all at once.

#### Key Components of Coroutines

* C++20 coroutines use three new keywords:
  + co\_return → Returns a value and ends the coroutine. It's similar to return, but it's used in the context of coroutines.
  + co\_yield → Pauses the coroutine and returns a value, but can continue later.
  + co\_await → Suspends execution until a task is completed.

### constexpr**Improvements**

* constexpr is a keyword in C++ that allows you to compute values at compile time instead of at runtime. This means the compiler can evaluate expressions and perform calculations while compiling your program, which can make your code faster and more efficient.
* Before C++20, constexpr functions were very limited. They could only perform simple operations, such as:
  + Basic arithmetic.
  + Return statements.
  + Very limited control flow (e.g., no loops or conditionals).
  + No dynamic memory allocation.
  + No try-catch blocks.
  + No virtual functions.
* This made it difficult to write complex compile-time computations using constexpr.

#### The C++20 Solution: Relaxing the Restrictions

* C++20 significantly relaxes the restrictions on what you can do inside a constexpr function. Now, you can use:
  + **Loops**: for, while, and **range-based for** loops.
  + **Conditional Statements**: if, switch.
  + try-catch **blocks**: For handling exceptions (though exceptions can't actually be thrown at compile time; the try-catch is useful for compile-time branching based on whether an exception would have been thrown).
  + **Variables**: You can declare variables inside constexpr functions.
  + **More complex expressions**: You can use more complicated expressions.
  + virtual **functions**: constexpr functions can now be virtual. This is a big deal for compile-time polymorphism.

#### Loops

* You can now use for, while, and range-based for loops in constexpr functions.

|  |
| --- |
| #include <array>  using namespace std;  constexpr int sumArray(const array<int, 5>& arr) {  int sum = 0;  // Loop allowed in C++20  for (int i = 0; i < arr.size(); ++i) {  sum += arr[i];  }  return sum;  }  int main() {  constexpr array<int, 5> arr = { 1, 2, 3, 4, 5 };  // Computed at compile time  constexpr int result = sumArray(arr);  // Check at compile time  static\_assert(result == 15);  return 0;  } |

#### Conditional Statements

* You can now use if and switch statements in constexpr functions.

|  |
| --- |
| constexpr int factorial(int n) {  // Conditional allowed in C++20  if (n <= 1) {  return 1;  }  else {  return n \* factorial(n - 1);  }  }  int main() {  // Computed at compile time  constexpr int result = factorial(5);  // Check at compile time  static\_assert(result == 120);  return 0;  } |

#### The try-catch Blocks

* You can now use try-catch blocks in constexpr functions. However, exceptions cannot actually be thrown at compile time. Instead, try-catch is useful for compile-time branching based on whether an exception would have been thrown.

|  |
| --- |
| #include <iostream>  using namespace std;  constexpr int safeDivide(int a, int b) {  try {  if (b == 0) {  // Exception not thrown at compile time  throw "Division by zero!";  }  return a / b;  }  catch (...) {  return -1; // Fallback value  }  }  int main() {  constexpr int result = safeDivide(10, 2);  // Check at compile time  static\_assert(result == 5);  int num = 1, den = 1;  cout << "Enter numerator: "; cin >> num;  cout << "Enter denominator: "; cin >> den;  // The same function works are runtime!  cout << "Result: " << safeDivide(num, den) << endl;  return 0;  } |

#### Variables

* You can now declare and use variables inside constexpr functions.

|  |
| --- |
| constexpr int square(int x) {  // Variable allowed in C++20  int result = x \* x;  return result;  } |

#### More Complex Expressions

* You can now use more complex expressions, including function calls, in constexpr functions.

|  |
| --- |
| constexpr int add(int a, int b) {  return a + b;  }  constexpr int multiply(int a, int b) {  return a \* b;  }  constexpr int compute(int x, int y) {  // Complex expression allowed in C++20  return add(multiply(x, y), 10);  }  int main() {  // Computed at compile time  constexpr int result = compute(3, 4);  // Check at compile time  static\_assert(result == 22);  return 0;  } |

#### Virtual Functions

* The constexpr functions can now be virtual, enabling compile-time polymorphism.

|  |
| --- |
| struct Base {  virtual constexpr int getValue() const {  return 1;  }  };  struct Derived : Base {  constexpr int getValue() const override {  return 2;  }  };  constexpr int computeValue(const Base& obj) {  // Calls the overridden function at compile time  return obj.getValue();  }  int main() {  constexpr Derived d;  // Computed at compile time  constexpr int result = computeValue(d);  // Check at compile time  static\_assert(result == 2);  return 0;  } |

### Immediate Functions (consteval)

* We already know that constexpr tells the compiler that a function can be evaluated at compile time if its arguments are known at compile time.
* However, constexpr **doesn't guarantee compile-time evaluation**. It's possible for a constexpr function to be called at runtime if its arguments are not known until runtime.

#### The Problem: Ensuring Compile-Time Evaluation

* Sometimes, we need a function to be executed at compile time. We might be doing metaprogramming, generating code at compile time, or performing calculations that must be done at compile time for our program to work correctly.
* The constexpr alone doesn't give we that guarantee.

#### The Solution: consteval (Immediate Functions)

* C++20 introduces consteval. consteval declares a function as an **immediate function**. This means that the function **must be evaluated at compile time**. If the compiler cannot evaluate a consteval function at compile time, it will produce an error.

|  |
| --- |
| // Must be evaluated at compile time  consteval int factorial(int n) {  if (n <= 1) {  return 1;  }  else {  return n \* factorial(n - 1);  }  }  int main() {  // Evaluated at compile time (no choice!)  constexpr int result = factorial(5);  static\_assert(result == 120); // OK  int runtime\_val = 10;  // Error: factorial is not a constant expression  //int runtime\_result = factorial(runtime\_val);  // ...  } |

#### **Difference Between** constexpr **and** consteval

| **Feature** | constexpr | consteval |
| --- | --- | --- |
| **Evaluation** | Can be evaluated at compile time or runtime | Must be evaluated at compile time |
| **Compile-time execution** | Optional (can run at runtime) | Mandatory (must run at compile-time) |
| **Use in runtime code** | Allowed if not used in a constexpr context | Not allowed in runtime code |
| **Flexibility** | More flexible | More restrictive |

|  |
| --- |
| #include <iostream>  using namespace std;  // `constexpr` function: Can be evaluated at  // compile-time OR runtime  constexpr int add(int a, int b) {  return a + b;  }  // `consteval` function: MUST be evaluated  // at compile-time  consteval int multiply(int a, int b) {  return a \* b;  }  int main() {  int x = 3, y = 4;  // Compile-time evaluation  cout << add(2, 3) << endl;  // Allowed (evaluated at runtime)  cout << add(x, y) << endl;  // ERROR: must be compile-time (consteval)  // cout << multiply(x, y) << endl;  return 0;  } |

### constinit Keyword

* C++20 introduced the constinit keyword, which **ensures** that a variable is **initialized at compile time** but **can be modified at runtime** (unlike constexpr, which enforces immutability).
* Before C++20, **static storage duration variables** (like **global** and static variables) might be initialized at **runtime**, leading to potential performance issues and **static initialization order fiasco**.

#### What Are Static Storage Duration Variables?

* Static storage duration variables are variables that exist for the **entire lifetime of the program**. They include:
  + **Global variables**: Variables declared outside of any function.
  + **Static variables**: Variables declared with the static keyword inside a function or class.

#### The Problem: Static Initialization Order Fiasco

* The **static initialization order** **fiasco** (fiasco - an event that does not succeed) refers to the problem where the order of initialization of static storage duration variables across different translation units (source files) is undefined. This can lead to bugs and runtime issues.
* Why Does This Happen?
  + The C++ standard does not define the order in which global/static variables in different translation units are initialized.
  + If one global variable depends on another global variable being initialized first, the program may behave unpredictably.

#### How C++20’s constinit Solves This Problem

* C++20 introduced the constinit keyword to ensure that a variable is statically initialized (i.e., initialized at compile time). This avoids the **static initialization order fiasco** and runtime overhead.

|  |
| --- |
| // Must be evaluated at compile time  consteval int getValue() { return 42; }  // Ensures compile-time initialization  constinit int globalVar = getValue();  int main() {  cout << "globalVar: " << globalVar << endl;  globalVar = 100; // Allowed, unlike constexpr  cout << "Updated globalVar: " << globalVar << endl;  return 0;  } |

* + globalVar must be initialized at compile-time.
  + Unlike constexpr, it can be modified at runtime.

#### constinit **vs** constexpr **vs** const

| **Feature** | constinit | constexpr | const |
| --- | --- | --- | --- |
| **Compile-time initialization required?** | ✅ Yes | ✅ Yes | ❌ No |
| **Value can change at runtime?** | ✅ Yes | ❌ No | ❌ No |
| **Valid for function return values?** | ❌ No | ✅ Yes | ✅ Yes |
| **Best used for** | Global/static variables | Compile-time constants | Read-only variables |

#### **Summary**

| **Feature** | constinit |
| --- | --- |
| **Ensures compile-time initialization** | ✅ Yes |
| **Allows modification at runtime** | ✅ Yes |
| **Prevents static initialization order issues** | ✅ Yes |
| **Cannot be used on function return values** | ✅ Yes |

### New Attributes

* C++20 introduced new attributes to improve performance and memory efficiency. These attributes provide hints to the compiler but do not change program correctness.
* New attributes introduced in C++20 are…
  + [[no\_unique\_address]]
  + [[likely]]
  + [[unlikely]]

#### The [[no\_unique\_address]] attribute

* **The Problem:**
  + Sometimes, you might have empty members in a struct or class. The compiler is required to give each member a unique address, even if they don't store any data. This can sometimes lead to unnecessary memory usage (especially if you have many empty members).
  + When you have a struct/class with **empty members** that would otherwise take up extra memory due to padding, use [[no\_unique\_address]].

|  |
| --- |
| #include <iostream>  using namespace std;  struct Empty {};  struct MyStruct {  int data;  [[no\_unique\_address]] Empty e;  };  int main() {  MyStruct s;  cout << "Size of MyStruct: "  << sizeof(s) << endl; // Optimized size  return 0;  } |

#### Branch Prediction Hints: [[likely]] and [[unlikely]]

* **The Problem**:
  + When your code has branches (using if statements or switch statements), the compiler often has to make guesses about which branch is more likely to be taken. If it guesses wrong, it can lead to less efficient code.
* **The Solution**:
  + [[likely]] and [[unlikely]] are hints you give to the compiler about the probability of a branch being taken. [[likely]] tells the compiler that a branch is likely to be taken, and [[unlikely]] tells it that a branch is unlikely to be taken.

|  |
| --- |
| #include <iostream>  using namespace std;  int process(int x) {  if (x > 0) [[likely]] { // Most cases  return x \* 2;  }  else [[unlikely]] { // Rare case  return -1;  }  }  int main() {  cout << process(10) << '\n'; // Fast path  cout << process(-5) << '\n'; // Unlikely case  } |

### Class Default Member Initializers for Bit-fields

* Bit-fields are a way to store multiple values in a single integer-type variable, using a specific number of bits for each value. This helps save memory when storing small values.
* Before C++20, **you couldn't initialize bit-fields directly inside the** class.

|  |
| --- |
| struct Flags {  unsigned int is\_enabled : 1; // 1-bit field  unsigned int mode : 2; // 2-bit field  };  int main() {  Flags f;  // Undefined behaviour! Values are uninitialized.  std::cout << f.is\_enabled << " "  << f.mode << '\n';  } |

* + The bit-fields are **uninitialized**, leading to **undefined behaviour** if accessed before assignment.

#### C++20: Default Initializers for Bit-fields

|  |
| --- |
| struct Flags {  unsigned int is\_enabled : 1 {1}; // Default to 1  unsigned int mode : 2 {2}; // Default to 2  };  int main() {  Flags f;  std::cout << f.is\_enabled  << " " << f.mode << '\n'; // Output: 1 2  } |

* + No need for constructors just to initialize bit-fields.
  + Prevents uninitialized values, avoiding undefined behaviour.
  + Code becomes more readable and maintainable.

# Library Features

### 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.

### The std::format

* std::format is a **new way to format strings** in C++20, similar to Python’s str.format(). It is more **readable, safe, and flexible** compared to printf and std::cout.

#### Why std::format?

* Before C++20, formatting strings could be done using:
  + std::cout for output streaming (clunky for complex formatting).
  + printf (from C) but it’s not type-safe, meaning you could mix types and get unexpected results.
  + Manual string concatenation (std::string + std::to\_string()), which is tedious and error-prone.
* std::format simplifies this by combining type safety, flexibility, and modern syntax.

#### How std::format Works

* std::format is a function that allows you to build formatted strings using placeholders (like {}) inside a format string. You pass the values you want to insert, and std::format handles the formatting for you.
* Example:

|  |
| --- |
| #include <format>  int main() {  std::string formattedString =  std::format("Hello, {}! You have {} new messages.", "Alice", 5);  std::cout << formattedString << std::endl;  } // Hello, Alice! You have 5 new messages. |

#### Formatting Numbers, Floats, and Booleans

|  |
| --- |
| #include <format>  using namespace std;  int main() {  int age = 25;  double pi = 3.14159;  bool isHappy = true;  cout << format("Age: {}, Pi: {:.2f}, Happy: {}\n",  age, pi, isHappy);  }// Output: Age: 25, Pi: 3.14, Happy: true |

* {:.2f} → Rounds float to 2 decimal places.
* Boolean prints as "true" or "false" (instead of 1 or 0 in std::cout).

#### Positional Arguments

|  |
| --- |
| #include <format>  using namespace std;  int main() {  string message = format("{1} scored {0} points.", 100, "Bob");  cout << message << endl; // Output: Bob scored 100 points.  } |

* Here, {1} refers to the **second** argument ("Bob") and {0} refers to the **first** (100).

#### Aligning and Padding Text

|  |
| --- |
| int main() {  cout << format("|{:>10}|\n", "Right"); // Right align  cout << format("|{:<10}|\n", "Left"); // Left align  cout << format("|{:^10}|\n", "Center"); // Center align  }  /\* Output  | Right|  |Left |  | Center |  \*/ |

* + {:<10} → Left-align in 10-character-wide space
  + {:>10} → Right-align
  + {:^10} → Center-align

#### Formatting Numbers as Hex, Binary, Octal

|  |
| --- |
| int main() {  int number = 42;  cout << format("Decimal: {}, Hex: {:#x}, Binary: {:#b}, Octal: {:#o}\n",  number, number, number, number);  }  // Decimal: 42, Hex: 0x2a, Binary: 0b101010, Octal: 052 |

* + {:#x} → Hexadecimal
  + {:#b} → Binary
  + {:#o} → Octal

#### **Why Use** std::format**?**

| **Feature** | std::format **(C++20)** | printf **(C-style)** | std::cout **(C++98)** |
| --- | --- | --- | --- |
| Type-Safety | ✅ Yes | ❌ No (Prone to errors) | ✅ Yes |
| Readability | ✅ Easy to read | ❌ Harder (Needs format specifiers) | ❌ Needs std::to\_string() |
| Flexibility | ✅ Works with different types | ✅ Works, but risky | ✅ Works, but verbose |
| Alignment & Formatting | ✅ Simple | ❌ Difficult | ❌ Limited |

### The std::span

* The std::span is a **lightweight view over a contiguous sequence of elements**. It helps you work with arrays, std::vector, or raw pointers **without copying** data.

#### Why std::span?

* No copying → Unlike std::vector, std::span does not own the data.
* Safer than raw pointers → Avoids pointer decay and size mismatches.
* More flexible → Works with C-style arrays, std::vector, and pointers.

#### How Does std::span Work?

|  |
| --- |
| #include <span>  void printNumbers(std::span<int> numbers) {  for (int num : numbers) {  std::cout << num << " ";  }  std::cout << "\n";  }  int main() {  int arr[] = { 1, 2, 3, 4, 5 };  // No decay to pointer, retains size info!  printNumbers(arr);  } // 1 2 3 4 5 |

#### std::span vs. std::vector

|  |
| --- |
| #include <vector>  #include <iostream>  #include <span>  void process(std::span<int> span) {  std::cout << "Size: " << span.size() << "\n";  }  int main() {  std::vector<int> vec = { 10, 20, 30, 40 };  process(vec); // Works with std::vector!  int arr[] = { 50, 60, 70, 80, 90 };  std::span<int> sp(arr + 1, 3);  process(sp);  } |

* std::vector<int> **owns** memory, while std::span<int> **only views** existing memory.
* std::span essentially stores two things:

std::span<int> sp(arr + 1, 3);

* + A pointer to the beginning of the sequence.
  + The number of elements in the sequence.

#### Key Points to Remember:

* std::span is a non-owning view into a contiguous sequence of elements.
* It stores a pointer and a size.
* It provides a safe and convenient way to work with data sequences.
* It's lightweight and efficient.
* It doesn't own the data, so the data must outlive the std::span.