C++ 20 Language Features

Contents

a) Core Language Features

1. 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**.

```cpp
#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;
}
```

2. 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.

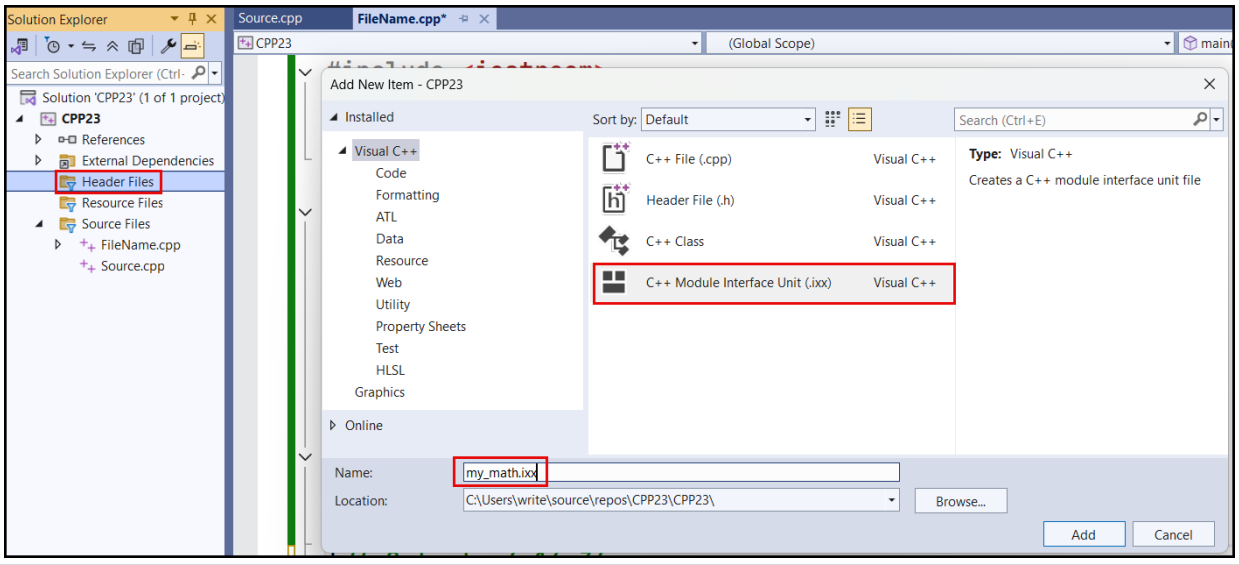
- **Order Dependence**: The order in which you include header files can sometimes matter. If a header file relies on something declared in another header file, you have to include them in the correct order. This is a common source of errors.
- **Name Collisions**: If two header files declare something 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

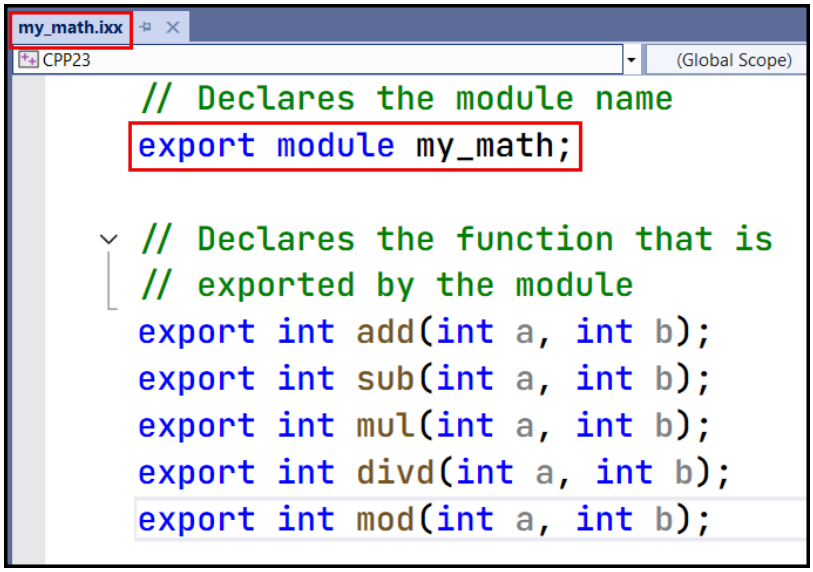
1. **Module Definition**:

- 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.

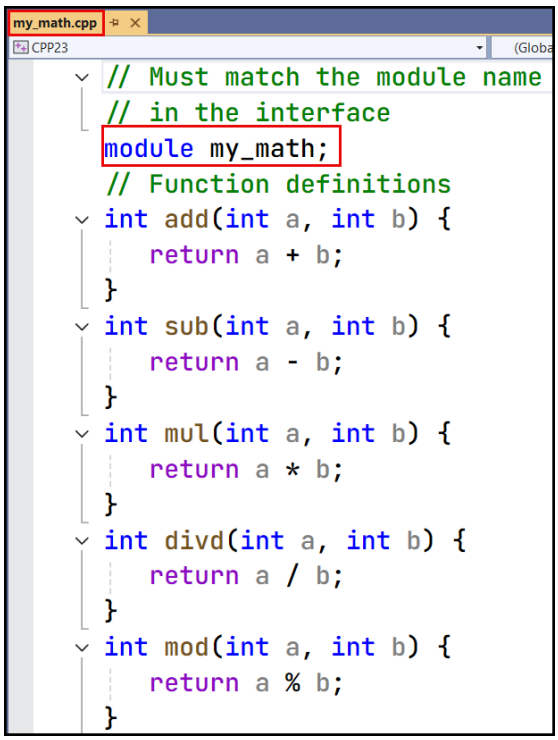


- Example:

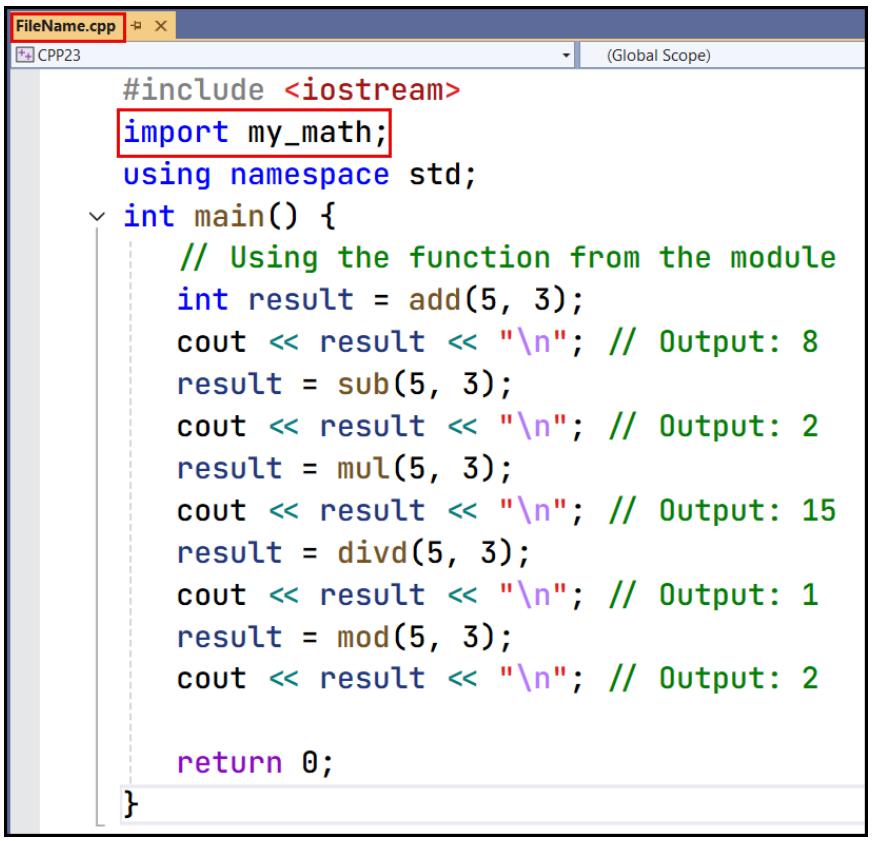
i. Module Interface - `my_math.ixx`



ii. Module Implementation – `my_mapth.cpp`



iii. 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

3. 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 `return`s 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 `return`s 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`).

4. 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;
};
```

```cpp
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).

5. Lambda Improvements

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

Problem Before C++20

I. **No template lambdas** – You couldn't write lambdas that worked with multiple types.

II. **No default-constructible lambdas** – Lambdas had no default constructor.

III. **No capturing this by value** – Capturing this by value required workarounds.

a. 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:

```cpp
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
}
```

b. 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!
```

c. 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

6. 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.

7. `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.

a. 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;
}
```

b. 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;
}
```

c. The `try-catch` Blocks

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

```
#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;
}
```

d. 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;
}
```

e. 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;
}
```

f. 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;
}
```

8. 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;
}
```

9. `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

10. 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]]`

1. 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]]`.

```cpp
#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;
}
```

2. 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
}
```

11. 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.

b) Library Features

12. 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.

13. 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

14. 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`.

15. The `std::jthread`

The Problem: Managing Threads

- In C++, we can create and manage threads to perform tasks concurrently. However, working with threads directly can be a bit tricky, especially when it comes to **joining them**.
- **What is "Joining"**? When a thread finishes its work, you often need to join it back to the main thread. This means waiting for the thread to complete before your main program continues. If you don't join a thread, your program might terminate before the thread has finished its work, leading to unexpected results or data corruption.
- **The Problem**: Manually joining threads can be error-prone. If you forget to join a thread, or if an exception is thrown before you get a chance to join it, your program could have issues.

Key Features of `std::jthread`

- Automatic Joining:
  - In C++, when you create a thread using `std::thread`, you need to explicitly call `join()` or `detach()` on the `thread` object. If you forget to do this, your program might terminate unexpectedly.
  - `std::jthread` automatically joins the `thread` when the `jthread` object is destroyed. This means you don't have to worry about manually joining the `thread`, which reduces the risk of errors.

```
#include <iostream>
#include <thread>
#include <chrono>

void work() {
   std::this_thread::sleep_for(std::chrono::seconds(2));
   std::cout << "Thread finished work!\n";
}

int main() {
   std::jthread jt(work);  // No need to call join()
   std::cout << "Main thread ends\n";

   return 0;  // `jt` automatically joins before exiting!
}
// Output:
// Main thread ends
// Thread finished work!
```

- Stop Tokens:
  - `std::jthread` introduces the concept of a "**stop token**," which allows you to request the `thread` to stop its execution gracefully.
  - This is useful when you want to stop a `thread` from running without abruptly terminating it.

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
```

```
void work(stop_token st) {
   while (!st.stop_requested()) {
      cout << "Working...\n";
      this_thread::sleep_for(
         chrono::milliseconds(500));
   }
   cout << "Thread stopped!\n";
}

int main() {
   jthread jt(work);

   this_thread::sleep_for(
      chrono::seconds(2));

   jt.request_stop();  // Ask thread to stop

   return 0;
}
```

- How it works?
  - The `thread` checks `stop_requested()` periodically.
  - `request_stop()` tells the `thread` to stop.
  - The `thread` exits cleanly.

16. The std::atomic_ref

The Problem: Atomic Operations on Existing Data

- In concurrent programming, when multiple threads access and modify shared data, you need to use atomic operations to prevent data races and ensure data consistency.
- Atomic operations are indivisible; they happen as a single, uninterruptible unit.
- You typically use types like `std::atomic<int>`, `std::atomic<bool>`, etc., to make variables atomic.
- However, this required you to *store* the data itself as an atomic type. What if you already **had existing data** (like in a `struct` or `class`) that you wanted to access atomically *without* changing the data's type?

The Solution: `std::atomic_ref`

- C++20 introduced `std::atomic_ref` to solve this problem. `std::atomic_ref` allows you to perform `atomic` operations on existing data, even if that data is not stored as an `atomic` type. It provides an `atomic` view into the existing data. It's like **temporarily treating a non-atomic variable as if it were** `atomic` for certain operations.

```
#include <iostream>
#include <atomic>
#include <thread>

int counter = 0;  // Regular non-atomic variable

void increment() {
   std::atomic_ref<int> atomic_counter(counter);  // Wrap it with atomic_ref
   for (int i = 0; i < 1000; ++i) {
      atomic_counter.fetch_add(1, std::memory_order_relaxed);
   }
}

int main() {
   std::thread t1(increment);
   std::thread t2(increment);

   t1.join();
   t2.join();

   std::cout << "Final counter value: " << counter << '\n';  // Output: 2000

   return 0;
}
```

17. Easier Removal from Containers: `std::erase` and `std::erase_if`

The Problem: Removing Elements from Containers

- In C++, we often need to remove elements from containers like `std::vector`, `std::list`, `std::string`, etc. Before C++20, the standard way to do this was a bit cumbersome, especially when you wanted to remove elements based on a condition. The typical approach involved a combination of `std::remove_if` (or `std::remove`) along with the container's `erase` method.

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

int main() {
   std::vector<int> vec = { 1, 2, 3, 4, 2, 5, 3, 2, 3 };

   // "Remove" elements with value 2,
   // but this doesn't actually change the size of the vector
   auto it = std::remove(vec.begin(), vec.end(), 2);

   // "Erase" the unused part of the vector
   vec.erase(it, vec.end());

   // One Liner: To remove all 3's
   vec.erase(std::remove(vec.begin(), vec.end(), 3), vec.end());
   // Output the vector
   for (int num : vec) {
      std::cout << num << " ";  // Output: 1 4 5
   }

   return 0;
}
```

- Problems with erase-remove:
    - Complicated syntax – Need both `std::remove` and `erase`.
    - Error-prone – Forgetting `.erase()` can cause issues.
    - More typing – Redundant `vec.begin()`, `vec.end()`.

The Solution: `std::erase` and `std::erase_if`

- C++20 introduced `std::erase` and `std::erase_if` as more convenient and safer ways to remove elements from containers. These functions are **associated with the container** (previous method were associated with iterators) itself, making the code cleaner and easier to understand.

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

int main() {
   std::vector<int> vec = { 1, 2, 3, 4, 2, 5, 3, 2, 3 };
```

```
    // Remove all 3's
    std::erase(vec, 3);
    // Remove even numbers
    std::erase_if(vec, [](int x) { return x % 2 == 0; });

    for (int num : vec)
        std::cout << num << " ";  // Output: 1 5

    return 0;
}
```

Containers That Support `std::erase` and `std::erase_if`

- These functions work with most standard containers, such as:
  - `std::vector`
  - `std::string`
  - `std::deque`
  - `std::list`
  - `std::forward_list`
  - `std::set` (for `std::erase_if`)
  - `std::unordered_set` (for `std::erase_if`)
  - `std::map` (for `std::erase_if`)
  - `std::unordered_map` (for `std::erase_if`)

```
#include <iostream>
#include <vector>
#include <string>
#include <deque>
#include <algorithm>
#include <list>
#include <forward_list>
#include <set>
#include <unordered_set>
#include <map>
#include <unordered_map>
#include <format>
using namespace std;
```

```cpp
template <typename Container>
void processContainer(Container& nums)
{
   // Remove all 3's
   erase(nums, 3);
   // Remove even numbers
   erase_if(nums, [](int num) { return num % 2 == 0; });

   // Print the remaining elements
   for (int num : nums) {
      cout << num << " ";
   }
   cout << "\n";
}

int main() {
   {  // string
      string text = "Hello, World! Hello, C++!";
      // Remove all l's
      erase(text, 'l');
      // Remove all punctuation characters
      erase_if(text, [](char ch) { return ispunct(ch); });
      cout << text << "\n";   // Output: Heo Word Heo C
   }
   // Vector
   vector vec = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   processContainer(vec); // Output: 1 5 7 9

   // Deque
   deque deq = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   processContainer(deq); // Output: 1 5 7 9

   // List
   list lst = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   processContainer(lst); // Output: 1 5 7 9

   // Forward List
   forward_list flst = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   processContainer(flst); // Output: 1 5 7 9

   {  // Set
      set st = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
      // erase(nums, 3); Error

      // Remove even numbers
      erase_if(st, [](int num) { return num % 2 == 0; });
```

```
        // Print the remaining elements
        for (int num : st) {
            cout << num << " ";
        }
        cout << "\n";
    }
    {   // Unordered Set
        unordered_set ust = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // erase(nums, 3); Error

        // Remove even numbers
        erase_if(ust, [](int num) { return num % 2 == 0; });

        // Print the remaining elements
        for (int num : ust) {
            cout << num << " ";
        }
        cout << "\n";
    }

    {   // Map
        map<int, string> myMap = {
          {1, "one"}, {2, "two"}, {3, "three"}, {4, "four"}
        };

        erase_if(myMap, [](auto p) { return p.first % 2 == 0; });
        for (const auto& [key, value] : myMap) {
            auto msg = format("Key: {}, Value: {}\n", key, value);
            cout << msg;
        }
    }

    {   // Unordered Map
        unordered_map<int, string> myUnorderedMap = {
          {1, "one"}, {2, "two"}, {3, "three"}, {4, "four"}
        };
        erase_if(myUnorderedMap, [](auto p) { return p.first % 2 == 0; });
        for (const auto& [key, value] : myUnorderedMap) {
            auto msg = format("Key: {}, Value: {}\n", key, value);
            cout << msg;
        }
    }
    return 0;
}
```

18. The `std::to_array`

The Problem: Converting to `std::to_array`

- Before C++20, C-style arrays (`int arr[]`) were commonly used, but they have several drawbacks:
  - They decay into pointers, losing size information.
  - They lack safety (no built-in bounds checking).
  - They can't be easily passed or returned as function arguments.

The Solution: `std::to_array`

- On the other hand, `std::array<T, N>` (introduced in C++11) is a safer alternative that:
  - Preserves size information.
  - Provides STL container-like behaviour.
  - Supports `.size()`, `.at()`, and iterators.
- But before C++20, converting a C-style array into `std::array` was tedious.
- In **C++20**, the new `std::to_array` function was introduced to make it easier to **convert raw arrays (C-style arrays) into `std::array`**. This helps improve **type safety, convenience, and usability**.

```
#include <array>
#include <utility>  // For std::to_array

int main() {
   int arr[] = { 1, 2, 3, 4, 5 };

   // Automatically deduces type and size!
   auto myArray = std::to_array(arr);

   for (int num : myArray) {
      std::cout << num << " ";
   }// Output: 1 2 3 4 5
}
```

19. The `std::ranges` Algorithms

The Problem: Algorithms and Iterators

- In C++, algorithms like `std::sort`, `std::find`, and `std::transform` are used to perform operations on collections of data (like `array`s or `vector`s). Before C++20, these algorithms required you to pass iterators (e.g., `begin()` and `end()`) to specify the range of elements to operate on.
- While powerful, this approach can be a bit cumbersome and error-prone, especially when dealing with complex ranges or when you need to chain multiple algorithms together.

The Solution: `std::ranges`

- C++20 introduced **ranges-based algorithms** (`std::ranges::`), which are improved versions of traditional STL algorithms (like `std::sort`, `, etc.). They make code easier to read, safer, and more flexible.

Key Concepts

- **Range**:
  - A range is **anything that you can iterate over**, like a `std::vector`, `std::array`, or even a plain C-style array.
- **View**:
  - A view is a lightweight, **non-owning range** that can transform or filter data on the fly. Views are lazy, meaning they don't compute anything until you iterate over them.
- **Algorithm**:
  - An algorithm is a function that **performs an operation** on a range, like sorting, searching, or transforming.

Example:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ranges>
using namespace std;
```

```cpp
int main() {
    vector vec1 = { 3, 1, 4, 1, 5, 9 };
    // Before C++20: Need to pass begin() and end()
    sort(vec1.begin(), vec1.end());
    for (int n : vec1) cout << n << " ";

    cout << "\n";

    vector vec2 = { 3, 1, 4, 1, 5, 9 };
    ranges::sort(vec2);  // No begin()/end()
    for (int n : vec2) cout << n << " ";
    cout << "\n";

    // Before C++20: Need to pass begin() and end()
    auto it1 = std::find(vec1.begin(), vec1.end(), 4);
    if (it1 != vec1.end()) {
        std::cout << "Found: " << *it1 << "\n"; // Output: Found: 4
    }
    else {
        std::cout << "Not found\n";
    }

    auto it2 = std::ranges::find(vec2, 4); // No begin()/end()
    if (it2 != vec2.end()) {
        std::cout << "Found: " << *it2 << "\n"; // Output: Found: 4
    }
    else {
        std::cout << "Not found\n";
    }

    vector<int> squares;
    squares.reserve(vec1.size());
    // Before C++20: Need to pass begin() and end()
    std::transform(vec1.begin(), vec1.end(),
        std::back_inserter(squares), [](int n) { return n * n; });

    for (int n : squares) cout << n << " ";
    cout << "\n";

    // We need to pass Output iterator: squares.begin()
    std::ranges::transform(vec2, squares.begin(),
        [](int n) { return n * n; });

    for (int n : squares) cout << n << " ";
    cout << "\n";
}
```

Projection in `std::ranges`

- A projection in C++20's ranges-based algorithms is a way to **transform** or **extract** a **specific part of an object** before applying an algorithm like **`std::ranges::sort`**.
- It allows you to sort, search, or manipulate a collection based on a specific property of objects, **without needing to write a custom comparator**.
- Before C++20, when sorting a list of objects (like `Person` structs), you had to define a **custom comparator**:

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

struct Person {
   std::string name;
   int age;
};

int main() {
   std::vector<Person> people = {
      {"Alice", 30},
      {"Bob", 25},
      {"Charlie", 35}
   };

   std::sort(people.begin(), people.end(),
      // Custom comparator required
      [](const Person& a, const Person& b)
      {
         return a.age < b.age;
      });

   std::cout << "Sorting based on Age:\n";
   for (const auto& p : people)
      std::cout << p.name << " (" << p.age << ")\n";
}
```

- Issue:
  - You need to manually write the lambda function (`[](const Person& a, const Person& b) { return a.age < b.age; });`) for sorting by `age`.

- Sorting with Projections (`std::ranges::sort`)
    - With **projections**, you **don't need a custom comparator**! You can simply tell the algorithm to use a specific member (`age`):

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

struct Person {
   std::string name;
   int age;
};

int main() {
   std::vector<Person> people = {
      {"Alice", 30},
      {"Bob", 25},
      {"Charlie", 35}
   };

   // Sorting by 'age' using a projection
   std::ranges::sort(people, {}, &Person::age);

   std::cout << "Sorting based on Age:\n";
   for (const auto& p : people)
      std::cout << p.name << " (" << p.age << ")\n";
}
```

```cpp
std::ranges::sort(people, {}, &Person::age);
```

- The first parameter is the collection that you want to sort.
- The second parameter `{}` is empty, meaning the default sorting (`<`) is used.
- `&Person::age` extracts the `age` property of each object before sorting.
- Internally, the algorithm compares objects based on their `age` values without needing a lambda function.

Example 2: Finding the Oldest Person Using Projections

```cpp
auto oldest = std::ranges::max_element(people, {}, &Person::age);
std::cout << "Oldest: " << oldest->name
   << " (" << oldest->age << ")\n";
```

Example: Counting people with age 30 and above

```
int count = std::ranges::count_if(people,
   [](int age) { return age >= 30; },
   &Person::age);
std::cout << "People above 30: " << count << "\n";
```

Example: Copying names of all people into another vector

```
std::vector<std::string> names;
std::ranges::transform(people,
   std::back_inserter(names), &Person::name);
```

- Many `std::ranges` algorithms support **projections**, allowing them to operate on **specific attributes of objects** without requiring a custom comparator or transformation function.

20. Improvements - Class Template Argument Deduction (CTAD)

What is CTAD?

- Before C++17, when using **class templates**, you had to **explicitly** specify template parameters. C++17 introduced **Class Template Argument Deduction (CTAD)**, allowing the compiler to **automatically deduce** template arguments based on constructor parameters.
- Example:

```
std::vector v = { 1, 2, 3 };  // Compiler deduces std::vector<int>
```

C++20 CTAD Improvements

a. CTAD for Aggregates (Structs & Classes Without Explicit Constructors)
   - Before C++20, CTAD did not work with aggregate types (structs with only public data members and no user-defined constructors). Now, CTAD works with them too!

```
struct Point {
   int x, y;
};

int main() {
   Point p = { 10, 20 };  // C++20: Compiler deduces Point<int, int>
```

b. CTAD for Derived Classes

- In C++17, **CTAD didn't work well with derived classes**. C++20 **allows base class CTAD to be inherited by derived classes**.

```
template <typename T>
struct Base {
   T value;
   Base(T v) : value(v) {}
};

struct Derived : Base<int> {
   // Inherit constructors and CTAD
   using Base<int>::Base;
};

Derived d = 42;  // C++20 deduces Derived<int>
```

c. CTAD for Alias Templates

- **C++17 required** `Ptr<int> p = std::make_shared<int>(10);`, but **C++20 deduces** `Ptr<int>` **automatically**.

```
template <typename T>
using Ptr = std::shared_ptr<T>;

Ptr p = std::make_shared<int>(10);  // C++20 deduces Ptr<int>
```