C++ 23 Language Features

Contents

a) Core Language Features

1. Compile-time conditional execution - `if consteval`

- C++ already has constexpr and consteval to enforce compile-time evaluation:
    - constexpr allows a function to be evaluated at compile-time or runtime.
    - consteval forces a function to be evaluated only at compile-time.
- Sometimes, you want to do optimizations or use different algorithms depending on whether a value is known at compile time. For example, if you know a value is constant, you might be able to precalculate something during compilation, making your program run faster.

Enter `if consteval`

- `if consteval` is a conditional statement that is evaluated *at compile time*. It's like a regular `if` statement, but the compiler checks the condition *while compiling the code*, not while the program is running.

```
#include <iostream>
using namespace std;
constexpr int process(int x) {
   if consteval {
      // Compile-time calculation
      return x * x;
   }
   else {
      // Runtime calculation
      return x * 2;
   }
}
// https://godbolt.org/
int main() {
   // Computed at compile-time
   constexpr int result1 = process(5);
   cout << "Compile-time Result1: " << result1 << "\n";

   int y = 6;
   // Computed at runtime
   int result2 = process(y);
   cout << "Runtime Result2: " << result2 << "\n";

   return 0;
}
```

2. Deducing this

The ‘this’ pointer

- In C++, when you write a **member function** (a function that belongs to a `class`), the ‘`this`’ pointer is implicitly passed to the function. This ‘`this`’ pointer points to the current object instance, allowing you to access its members.

```
class MyClass {
public:
   void print() {
      std::cout << "Hello from MyClass!\n";
   }
};

MyClass obj;
obj.print();
```

- When you call `obj.print()`, the `this` pointer inside `print()` points to `obj`.

The "Deducing this" feature

- The "Deducing this" feature allows you to explicitly capture the ‘`this`’ pointer as a parameter in a member function `template`.
- Before C++23, if you wanted to write a member function `template` that could work with different types of objects (e.g., base `class` or derived `class`), you had to rely on workarounds like Curiously Recurring Template Pattern (CRTP) or manually deducing the type. This could get messy and hard to read.

CRTP

- What is the CRTP?
  - The CRTP is a design pattern in C++ where a class derives from a class template, and the template argument is the class itself. It sounds a bit circular, and that's why it's called "curiously recurring."
- Why is it used?
  - The CRTP allows you to achieve static polymorphism (compile-time polymorphism). This means that the specific behaviour is determined at compile time, rather than at runtime.

- A Simple Example

```
#include <iostream>
using namespace std;
template <typename Derived>
class Base {
public:
   void interface() {
      static_cast<Derived*>(this)->implementation();
   }

   void implementation() {
      cout << "Base implementation\n";
   }
};

class Derived : public Base<Derived> {
public:
   void implementation() {
      cout << "Derived implementation\n";
   }
};

int main() {
   Derived d;
   d.interface(); // Calls Derived::implementation
   return 0;
}

// Derived implementation
```

- Explanation:
  - `template <typename Derived>`: `Base` is a `class template`. The `template` parameter `Derived` is what makes this the **CRTP**.
  - `class Derived:public Base<Derived>`: This is the "**curiously recurring**" part. `Derived` inherits from `Base`, but the `template` argument to `Base` is `Derived` itself. So, `Base` is instantiated with `Derived` as its `template` parameter.
  - `void interface()`: This function is in the base `class`. It's the key to how the CRTP works.

    - `static_cast<Derived*>(this)`: Inside `interface()`, we're casting this (which is a pointer to `Base`) to a pointer to `Derived`. This is safe because, in this specific CRTP usage, we know that `this` is actually pointing to a `Derived` object. This is because `Derived` inherits from `Base<Derived>`.
    - `->implementation()`: After the cast, we call the `implementation()` function. Because of the cast, the compiler knows we are calling the `implementation()` function of the `Derived` `class`.
- How it works:
    - When you create a `Derived` object and call `d.interface()`, the following happens:
        - `d.interface()` is called. This is the `interface()` function in the `Base<Derived>` `class`.
        - Inside `interface()`, `this` points to the `Derived` object `d`.
        - `static_cast<Derived*>(this)` converts the `Base*` to a `Derived*`.
        - `->implementation()` calls the `implementation()` function of the `Derived` `class`.

Coming back to 'Deducing this'

```
#include <iostream>
using namespace std;
class Base {
public:
   template <typename Self>
   void print(this Self&& self) {
      std::cout << "Hello from Base!\n";
   }
};

class Derived : public Base {
public:
   void print() {
      std::cout << "Hello from Derived!\n";
   }
};

int main() {
   Base obj;
```

```
    obj.print(); // Calls Base::print

    Derived derivedObj;
    derivedObj.print(); // Calls Derived::print
}
// Hello from Base!
// Hello from Derived!
```

- Key Points
  - `this Self&& self`:
    - This is the new syntax introduced in C++23.
    - `Self` is a template parameter that deduces the type of the object (`Base`, `Derived`, etc.).
    - `self` is the explicit `this` pointer, which can be used to access the object's members.
  - Flexibility:
    - The `print` function template can now work with any type of object, including derived classes.
    - If you call print on a `Derived` object, `Self` will be deduced as `Derived`.
  - No More CRTP:
    - Before C++23, you might have used CRTP (Curiously Recurring Template Pattern) to achieve similar behaviour. Now, you can avoid that complexity.

How This Feature Helps

- Deducing this allows us to write more generic member functions.
- It enables CRTP-like behaviours without explicitly using CRTP.

b) Library Features

3. Standardized way to handle errors without exceptions - `std::expected`

What is `std::expected`?

- `std::expected<T, E>` is a new C++23 feature that provides a standardized way to handle errors without using exceptions.
- It's part of the `<expected>` header and is designed to make error handling more explicit, predictable, and efficient.
- It is useful when a function can either:
  - Return a **valid result** (`T`) OR
  - Return an **error value** (`E`)
- Think of it as a "box" that can hold either:
  - A successful result (the value you wanted), or
  - An error (something went wrong).
- It's like `std::optional<T>`, but instead of just "**value** or **nothing**," it stores an error when something goes wrong.

```cpp
#include <iostream>
#include <expected> // C++23 feature
using namespace std;

// Function that returns either a valid result (double) or
// an error message (string)
expected<double, string> divide(double a, double b) {
   if (b == 0) {
      // Return an error
      return unexpected("Error: Division by zero!");
   }
   return a / b; // Return the valid result
}

int main() {
   auto result = divide(10, 2); // Should return 5.0

   if (result) {  // Check if the operation was successful
      // *result gives the valid value
      cout << "Result: " << *result << "\n";
   }
   else {   // Retrieve the error message
```

```
        cout << result.error() << "\n";
    }

    // Should return an error
    auto result2 = divide(10, 0);

    if (result2) {
        cout << "Result: " << *result2 << "\n";
    }
    else {
        // Prints: Error: Division by zero!
        cout << result2.error() << "\n";
    }
}
```

4. Monadic operations for `std::optional`, `std::expected`

- The meaning of Monad - a single unit; the number one.

What is `std::optional`?

- `std::optional <T>` is a feature from C++17 that represents a value that might be missing.
    - If it contains a value → You can access it.
    - If it doesn't → You must handle the missing case.
- Before C++23, handling `std::optional` required manual `if` checks.
- Now, C++23 introduces monadic operations (`and_then`, `or_else`, and `transform`) that make code more concise and functional.

Why Monadic Operations?

- These functions avoid manual `if` checks and allow you to chain operations smoothly, making code cleaner and more readable.

Monadic Function	**Use Case**
`and_then(f)`	Apply `f` **only if** `optional` **has a value**
`or_else(f)`	Call `f` **only if** `optional` **is empty**
`transform(f)`	Modify the value **if present**

Example 1: `and_then` – Chain Dependent Operations

- If the `std::optional` has a value, `and_then(f)` calls `f` and returns its result.
- If empty, it returns `std::nullopt`.

```
#include <iostream>
#include <optional>
#include <string>
using namespace std;

// Function that returns an optional integer
optional<int> parse_int(const string& str) {
   try {
      return stoi(str);
   }
   catch (...) {
      return nullopt;
   }
}

// Function that returns an optional integer
optional<int> doubleIfPositive(int x) {
   if (x > 0)
      return x * 2;
   return nullopt; // No value if x is negative
}

int main() {
   string number;
   cout << "Enter a number: ";
   cin >> number;


   // Call `doubleIfPositive` only if `number` has a value
   optional<int> result = parse_int(number)
      .and_then(doubleIfPositive);

   if (result) cout << "Doubled: " << *result << "\n";
   else cout << "No value\n";
}
/*
Enter a number: 12
Doubled: 24
Enter a number: ABC
No value
*/
```

- If `number` contains a value (`12`), `doubleIfPositive(12)` is called, and it returns `24`.
- If `number` was `std::nullopt`, it would remain empty.

Example 2: `or_else` – Provide a Default Value

```cpp
#include <iostream>
#include <optional>
#include <string>
using namespace std;

// Function that returns an optional integer
optional<int> parse_int(const string& str) {
   try {
      return stoi(str);
   }
   catch (...) {
      return nullopt;
   }
}

// Function returning a default value
std::optional<int> getDefaultValue() {
   return 10; // Some default value
}

int main() {
   string number;
   cout << "Enter a number: ";
   cin >> number;


   // If parse_int has a value, then assign it
   // to result or else call getDefaultValue
   optional<int> result = parse_int(number)
      .or_else(getDefaultValue);

   if (result)
      cout << "Result: " << *result << "\n";
   else
      cout << "No value\n";
}
/*
Enter a number: 12
Result: 12
Enter a number: ABC
Doubled: 10
*/
```

- If `parse_int` returns a value after parsing, then it will assign it to `result` variable.
- If `parse_int` returns `std::nullopt` then, `getDefaultValue` is called.

Example 3: transform – Modify the Value if Present

```
#include <iostream>
#include <optional>
#include <string>
using namespace std;

// Function that returns an optional integer
optional<int> parse_int(const string& str) {
   try {
      return stoi(str);
   }
   catch (...) {
      return nullopt;
   }
}
// Function to square a number
int square(int x) {
   return x * x;
}
int main() {
   string number;
   cout << "Enter a number: ";
   cin >> number;

   // If parse_int has a value, then call
   // transform.
   optional<int> result = parse_int(number)
      .transform(square);

   if (result)
      cout << "Result: " << *result << "\n";
   else
      cout << "No value\n";
}
/*
Enter a number: 33
Result: 1089
Enter a number: ABC
No value
*/
```

- If `parse_int` returns a value, then apply '`square`' `transform`.
- If `parse_int` returns `std::nullopt`, then result would be empty.

Summary:

- In C++23, `std::optional` and `std::expected`, received new utility functions—`and_then`, `or_else`, and `transform`—which make handling these types more convenient and expressive.
- Here's a table comparing these functions:

Feature	`and_then`	`or_else`	`transform`
Purpose	Chains another operation if the **value is present** (`std::optional` is engaged, or `std::expected` has a valid value).	Specifies an alternative action when there is **no valid value** (`std::optional` is disengaged, or `std::expected` has an error).	Transforms the contained **value if it's present**.
Works on	`std::optional`, `std::expected`	`std::optional`, `std::expected`	`std::optional`, `std::expected`
When it's executed	If the **value is present**, applies the provided function and returns the result. Otherwise, returns an empty state (`std::nullopt` or **error**).	If the **value is missing**, applies the provided function to handle the alternative case. If the value is present, it remains unchanged.	If the **value is present**, applies the transformation function. Otherwise, it remains unchanged.
Return Type	Can return another `std::optional` or `std::expected` (typically used for function chaining).	Returns the same type as the original but with a potentially modified state.	Returns the same type but with a transformed contained value.
Example Use Case	Used for **chaining operations** when an optional or expected contains a value.	Used to **recover from an error** case or **provide an alternative value.**	Used for **modifying a valid value** without affecting error states.

5. Convert ranges into containers

- A range is anything that you can iterate over using a loop (e.g., a `std::vector`, a `std::array`, a `std::string`, iterators, views, spans or even a custom sequence).
- In C++23, a new feature called `std::ranges::to` was introduced, which makes it **easier and more convenient** to convert ranges (like iterators, views, or spans) into standard containers (like `std::vector`, `std::set`, etc.).

- Before C++23, if we had a range (like a `std::vector` or `std::set`) and wanted to convert it into another container, we had to do it manually using constructors or iterators.
- For example, let's say we have a `std::vector<int>` and want to convert it into a `std::set<int>` (which removes duplicates and sorts elements).

```
#include <vector>
#include <set>
#include <iostream>

int main() {
   std::vector<int> vec = { 3, 1, 4, 1, 5, 9, 2 };

   // Manually converting to std::set
   std::set<int> s(vec.begin(), vec.end());

   for (int num : s) {
      std::cout << num << " ";
   }
} // 1 2 3 4 5 9
```

- C++23 introduces `std::ranges::to`, which makes it much simpler to convert ranges into containers.

```
#include <vector>
#include <set>
#include <deque>
#include <iostream>
#include <ranges>  // Required for ranges::to
using namespace std;

int main() {
   vector<int> vec = { 3, 1, 4, 1, 5, 9, 2 };

   // Easy conversion to set using ranges::to
   set<int> s = ranges::to<set>(vec);
   for (int num : s) {
      cout << num << " ";
   } // 1 2 3 4 5 9
   cout << "\n";
   // Convert to set, but keep only even numbers
   set<int> even_numbers = ranges::to<set>(
      vec | views::filter([](int n) { return n % 2 == 0; })
   );
   for (int num : even_numbers) {
      cout << num << " ";
   } // 2 4
   cout << "\n";
   // Convert to deque
   deque<int> deq = ranges::to<deque>(vec);
   for (int num : deq) {
      cout << num << " ";
   } // 3 1 4 1 5 9 2
   cout << "\n";
}
```

6. Better support for constexpr in standard library algorithms

- Before C++23, many standard library algorithms (like `std::sort`, `std::find`, `std::transform`, etc.) were not fully `constexpr`-friendly. This limited the amount of computation you could do at compile time.

```
constexpr void sortArray() {
   std::array<int, 3> arr = { 3, 1, 2 };
   // ERROR: std::sort is NOT constexpr before C++23
   std::sort(arr.begin(), arr.end());
}
```

- In C++23, many **standard library algorithms** (like `std::sort`, `std::unique`, `std::remove_if`, etc.) now have `constexpr` **support**. This means you can now **run these algorithms at compile time**, making your code more **efficient** and **faster** at runtime.

```
#include <algorithm>
#include <array>
#include <iostream>

constexpr std::array<int, 3> sortArray() {
   std::array<int, 3> arr = { 3, 1, 2 };
   std::sort(arr.begin(), arr.end());
   return arr;
}
int main() {
   // Computed at compile time!
   constexpr auto sortedArr = sortArray();

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

7. Extended Algorithms

- C++23 introduces several new algorithms that enhance the capabilities of the C++ standard library.

Combine Multiple Ranges - `std::views::zip`

- `std::views::zip` takes **multiple ranges (like vectors, arrays, etc.)** and **pairs corresponding elements** together.

```
#include <iostream>
#include <vector>
#include <ranges>
#include <tuple>

int main() {
   std::vector names = { "Alice", "Bob", "Charlie" };
   std::vector ages = { 25, 30, 35 };


   // Zip the two vectors together
   for (auto [name, age] : std::views::zip(names, ages)) {
      std::cout << name << " is " << age << " years old.\n";
   }
}

/*
Alice is 25 years old.
Bob is 30 years old.
Charlie is 35 years old.
*/
```

Flatten a Range - `std::views::join`

- `std::views::join` **flattens** a range of **nested containers** (e.g., a `vector<vector<int>>`) into **one continuous sequence**.

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

int main() {
   vector<vector<int>> numbers = {
      {1, 2},
        {3, 4},
        {5, 6}
    };

    // Flatten (join) the nested vectors
    for (int num : numbers | views::join) {
        cout << num << " "; // 1 2 3 4 5 6
    }
    cout << "\n";

    vector<string> words = {
        "Hello",
        "Hi",
        "Dear"
    };

    for (auto i : words | views::join) {
        cout << i << " "; // H e l l o H i D e a r
    }
    cout << "\n";
}
```

Splitting a Range - `std::views::split`

- `std::views::split` **splits a range** into subranges based on a delimiter (like `std::string_view::split` but for general ranges).

```
#include <iostream>
#include <ranges>
#include <string_view>

int main() {
    std::string_view text = "Hello world from C++23";

    // Split by space
    for (auto word : text | std::views::split(' ')) {
        std::cout << std::string_view(word) << "\n";
    }
}
/*
Hello
world
from
C++23
*/
```

8. Faster Sorted Containers - `std::flat_map` and `std::flat_set`

- **Associative containers** store elements in a **sorted order**, allowing for efficient lookup, insertion, and deletion of elements based on their keys. `std::map` and `std::set` are the most commonly used associative containers.
- The **Cache Locality** Problem:
  - Traditional associative containers like `std::map` (implemented as a tree) can suffer from poor cache locality. When you access elements in a `std::map`, the elements might be scattered throughout memory, making it more likely that the CPU will have to wait for data to be loaded from slower memory (cache misses). This can slow down your program.
- C++23 introduces `std::flat_map` and `std::flat_set`, which are **sorted associative containers optimized for cache locality**. They work similarly to `std::map` and `std::set` but are more **cache-friendly** and can be faster in certain cases.

Cache Locality

- Cache locality refers to the proximity of data in memory and how it affects the efficiency of cache usage. There are two main types of cache locality:
  - **Temporal Locality**:
    - If a piece of data is accessed once, it's likely to be accessed again soon.
    - Example: A variable used repeatedly in a loop.
    - The CPU keeps this data in the cache to avoid fetching it from RAM multiple times.
  - **Spatial Locality**:
    - If a piece of data is accessed, nearby data is also likely to be accessed soon.
    - Example: Iterating over an array or a contiguous block of memory.
    - The CPU loads a block of memory (called a cache line) into the cache, anticipating that nearby data will be needed.

- Cache Locality in Action:

```
#include <iostream>
#include <vector>

int main() {
   std::vector<int> arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   int sum = 0;
   for (int i = 0; i < arr.size(); ++i) {
      sum += arr[i]; // Accessing elements sequentially
   }

   std::cout << "Sum: " << sum << std::endl;
   return 0;
}
```

- Why it's good: The elements of `arr` are stored contiguously in memory. When the CPU accesses `arr[0]`, it loads a cache line containing nearby elements (`arr[1]`, `arr[2]`, etc.). Subsequent accesses to `arr[1]`, `arr[2]`, etc., are fast because the data is already in the cache.

The `std::flat_map`

- **Like `std::map`**: `std::flat_map` stores key-value pairs, sorted by key.
- **Cache-friendly**: Elements are stored contiguously to improve cache locality.

```
#include <iostream>
#include <flat_map> // Include the <flat_map> header
using namespace std;
int main() {
   flat_map<string, int> ages;

   ages["Alice"] = 30;
   ages["Bob"] = 25;
   ages["Charlie"] = 35;

   for (const auto& [name, age] : ages) {
      cout << name << ": " << age << endl;
   }

   // Accessing elements:
   cout << "Bob's age: " << ages["Bob"] << endl;

   return 0;
}
/*
Alice: 30
Bob: 25
Charlie: 35
Bob's age: 25
*/
```

The std::flat_set

- **Like std::set**: std::flat_set stores unique elements, sorted by their value.
- **Cache-friendly**: Elements are stored contiguously to improve cache locality.

```
#include <iostream>
#include <flat_set>
using namespace std;
int main() {
   flat_set<int> fs = { 3, 1, 2 };
   // Elements are always sorted
   for (int num : fs) {
      cout << num << " ";
   }
   // Fast lookup
   if (fs.contains(2)) {
      cout << "\n2 is in the set!";
   }
}
/*
1 2 3
2 is in the set!
*/
```

When to Use?

- Use std::flat_map/std::flat_set when you do more lookups & iterations than insertions.
- Use std::map/std::set when you frequently insert/delete elements dynamically.

9. Implicit move constructors

- In C++, objects can be copied or moved:
  - Copying (Copy Constructor) → Creates a new object by duplicating the original.
  - Moving (Move Constructor) → Transfers ownership of resources from one object to another, avoiding unnecessary copies.

```
class Data {
public:
   std::vector<int> values;

   // Move Constructor (C++11 and later)
   Data(Data&& other) noexcept :
      values(std::move(other.values)) {}
};
```

  - `std::move(other.values)` moves the resource instead of copying it.
- The Problem Before C++23, the compiler did not generate an implicit move constructor if:
  - A **copy constructor** was **explicitly defined** (even if you didn't need it).
  - A **destructor** was defined, even if it didn't do anything.
- Example:

```
class Data {
public:
   std::vector<int> values;
   Data() = default;
   // Explicitly defined copy constructor
   Data(const Data& other) : values(other.values) {}
   // Explicit destructor (even if empty)
   ~Data() {}
};
Data createData() {
   Data d;
   // Will copy instead of move.
   return d;
}

int main() {
   // Copy constructor is used instead of move constructor.
   Data d1 = createData();
}
```

- Even though we are returning `Data` from a function, **a copy happens instead of a move** because defining a copy constructor prevents the compiler from generating a move constructor.

C++23 Implicit move constructors

- C++23 now allows the compiler to generate an implicit move constructor even if:
    - You define a **copy constructor**.
    - You define a **destructor**.

```
Data createData() {
   Data d;
   // Move happens instead of copy
   return d;
}

int main() {
   // Move happens instead of copy
   Data d1 = createData();
}
```

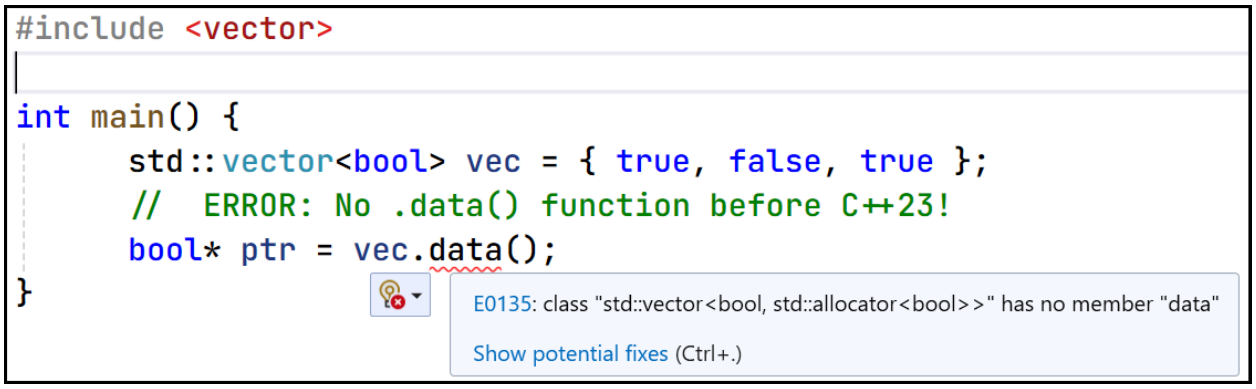
- This will help…
    - If you are working with large objects, reducing unnecessary copies will improve performance.
    - If your `class` defines a copy constructor or destructor, C++23 ensures move semantics still work properly.

10. The `std::vector<bool>`

Before C++23

- The `std::vector<bool>` is a specialized version of `std::vector<T>`, but it does not behave like a normal container.
- Instead of storing `bool` values as separate bytes (like `std::vector<int>` stores integers), `std::vector<bool>` was optimized to store multiple bool values in a compact **bit-packed representation**.
- This optimization caused several problems:

   - No direct access to elements (`operator[]` does not `return bool&`).
   - Iterators behave differently.
   - No `data()` function (so you couldn't get a pointer to the underlying storage).
- Example: Before C++23



```cpp
#include <vector>

int main() {
	std::vector<bool> vec = { true, false, true };
	//  ERROR: No .data() function before C++23!
	bool* ptr = vec.data();
}
```

E0135: class "std::vector<bool, std::allocator<bool>>" has no member "data"

Show potential fixes (Ctrl+.)

What Did C++23 Fix?

- C++23 made `std::vector<bool>` behave more like a normal container:
   - Added `.data()` – You can now access the underlying storage.
   - Improved iterator support – Now behaves more like `std::vector<int>`.
   - Better compatibility with standard algorithms.

As of now, GCC, Clang, and MSVC have not yet implemented the `std::vector<bool>::data()` member function introduced in C++23. This is a notable improvement in the C++23 standard, but compiler support for this specific feature is still pending.

11. Multidimensional `std::mdspan`

What is `std::span`?

- `std::span` (introduced in C++20) is a **view** over a sequence of elements.
- **It does not own the data** - it simply provides access to an existing array or contiguous memory.
- It helps in avoiding raw pointers and makes function parameters safer.

```cpp
#include <iostream>
#include <span>
```

```cpp
#include <array>
#include <vector>
using namespace std;
// Takes any contiguous int collection
void print(span<int> arr) {
   for (int num : arr) {
      cout << num << " ";
   }
   cout << "\n";
}

int main() {
   int num1[] = { 1, 2, 3, 4, 5 }; // C-style array
   print(num1);   // Passes entire array

   std::array num2 = { 1, 2, 3, 4, 5 }; // std::array
   print(num2);   // Passes entire std::array

   vector num3 = { 1, 2, 3, 4, 5 }; // std::vector
   print(num3);   // Passes entire std::vector

   std::span num4 = num3; // std::span
   print(num4);   // Passes entire std::span

   int* ptr = num3.data();
   size_t size = num3.size();
   print({ ptr, size });  // Construct std::span<int>

   print(std::span(num4.begin() + 1,
      num4.begin() + 4));  // Pass a subrange
}
```

What is `std::mdspan`?

- `std::mdspan` is like a ***view* into your grid data**. It **doesn't *own*** the data; it just *refers* to it. Think of it as a **window** through which you can see and access the elements of your grid. The key is that this **window can be configured** to view grids stored in different ways.

Why Do We Need `std::mdspan`?

- Before C++23, handling multi-dimensional arrays was inefficient and cumbersome:
  - **C-style arrays** (`T arr[3][3]`)
    - Hard to use and pass as function arguments.

        - Cannot change size dynamically.
    - `std::vector<std::vector<T>>` (Nested `std::vector`)
        - Memory is not contiguous → slower performance.
        - Extra memory overhead for each `std::vector`.
    - Raw pointers (`T* data`) with manual indexing
        - Hard to manage correctly.
        - No built-in bounds checking → risk of segmentation faults.
- `std::mdspan` fixes these problems by providing a safe, efficient, and flexible way to access multi-dimensional data.

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

int main()
{
   std::vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
   // View data as 3 X 4
   auto threeCrossFour = std::mdspan(v.data(), 3, 4);

   for (std::size_t i = 0; i != threeCrossFour.extent(0); i++)
   {
      for (std::size_t j = 0; j != threeCrossFour.extent(1); j++)
      {
         std::cout << threeCrossFour[i, j] << ' ';
      }
      std::cout << '\n';
   }

   std::cout << "Row Size:" << threeCrossFour.extent(0) << '\n';
   std::cout << "Column Size:" << threeCrossFour.extent(1) << '\n';
   return 0;
}
/*
1 2 3 4
5 6 7 8
9 10 11 12
Row Size:3
Column Size:4
*/
```

- Here `std::mdspan` creates a 3×4 view over a 1D `std::vector<int>` 'v'.
- `extent(N)` returns the size of dimension N in `std::mdspan`.
    - `extent(0)`: Number of rows.
    - `extent(1)`: Number of columns.

12. Standard Library Module

- In older versions of C++, we included standard library headers using `#include`. For example, to use `std::vector`, we would write `#include <vector>`.
- This `#include` mechanism is a text-based inclusion. The compiler literally copies and pastes the contents of the `<vector>` header file into your code.
- This has several drawbacks:
    - **Slow Compilation**: Including many headers can make compilation slow because the compiler has to process a lot of code, even if you only use a small part of it.
    - **Name Collisions**: Headers might define the same names, leading to conflicts.
    - **Order Dependence**: The order of `#include` directives can sometimes matter, which can be confusing.
- Example: Before `import std;`

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

int main() {
   vector<int> vec = { 3, 1, 4, 1, 5, 9, 2 };

   // Easy conversion to set using ranges::to
   set<int> s = ranges::to<set>(vec);
   for (int num : s) {
      cout << num << " ";
   } // 1 2 3 4 5 9
   cout << "\n";
   // Convert to set, but keep only even numbers
   set<int> even_numbers = ranges::to<set>(
      vec | views::filter([](int n) { return n % 2 == 0; })
   );
   for (int num : even_numbers) {
      cout << num << " ";
   } // 2 4
   cout << "\n";
   // Convert to deque
   deque<int> deq = ranges::to<deque>(vec);
   for (int num : deq) {
      cout << num << " ";
   } // 3 1 4 1 5 9 2
   cout << "\n";
}
```

Using `import std;`

- Now, you can **replace multiple includes** with a single line: `import std;`

```
import std;
using namespace std;

int main() {
    vector<int> vec = { 3, 1, 4, 1, 5, 9, 2 };

    // Easy conversion to set using ranges::to
    set<int> s = ranges::to<set>(vec);
    for (int num : s) {
        cout << num << " ";
    } // 1 2 3 4 5 9
    cout << "\n";
    // Convert to set, but keep only even numbers
    set<int> even_numbers = ranges::to<set>(
        vec | views::filter([](int n) { return n % 2 == 0; })
    );
    for (int num : even_numbers) {
        cout << num << " ";
    } // 2 4
    cout << "\n";
    // Convert to deque
    deque<int> deq = ranges::to<deque>(vec);
    for (int num : deq) {
        cout << num << " ";
    } // 3 1 4 1 5 9 2
    cout << "\n";
}
```

Why Is `import std;` Better?

Feature	`#include` (C++98 - C++20)	`import std;` (C++23)
Compilation Speed	Slower (text inclusion)	Faster (precompiled module)
Code Simplicity	Many `#include` lines	Just `import std;`
Redundancy	Same headers parsed repeatedly	Uses precompiled interface
Scalability	More includes slow builds	Efficient for large projects

13. The `std::print`

Why `std::print`?

- While **`std::cout`** is a C++ essential, it can be a bit complex for beginners (and sometimes even for experienced programmers). Formatting output with **`std::cout`** often involves using manipulators (like **`std::setw`**, **`std::fixed`**, **`std::setprecision`**), which can be a bit cumbersome.
- In C++23, a new feature called **`std::print`** was introduced, which provides a **simpler and more efficient** alternative to **`std::cout`** for printing output to the console.
- **It is similar** to **`std::format`** which creates a formatted string.
- The formatting rules and syntax used within the format string (the first argument) of **`std::print`** are *exactly* the same as those used with **`std::format`**.

```cpp
#include <print>

int main() {
   int age = 30;
   std::string name = "Alice";
   double height = 5.8;

   // Basic usage:
   std::print("{}! You are {} years old and {} feet tall.\n",
      name, age, height);

   // Formatting:
   std::print("{:10} {:5d} {:6.2f}\n",
      name, age, height);
```

```
    return 0;
}
```

- `{:10}`: Specifies that the `name` should be printed in a field of width `10`.
- `{:5d}`: Specifies that the `age` should be printed as a decimal integer in a field of width 5.
- `{:6.2f}`: Specifies that the `height` should be printed as a floating-point number with `2` decimal places, in a field of width `6`.

Custom Formatter for a `class`

- By default, `std::print` **only supports built-in types** like `int`, `double`, and `std::string`.
- In C++20, `std::format` was introduced as a modern, type-safe way to format strings, and C++23 allows you to **customize how your own types are formatted** by writing a **custom formatter**.

```
import std;
using namespace std;

class Student {
private:
    string name;
    int age;
    vector<int> grades;
public:
    Student(string name, int age, vector<int> grades)
        : name{ move(name) }, age{ age }, grades{ move(grades) } {
    }
    const string& getName() const { return name; }
    int getAge() const { return age; }
    const vector<int>& getGrades() const { return grades; }
};

template <> // Full specialization of formatter for Student
struct formatter<Student> : std::formatter<std::string> {
    auto format(Student h, format_context& ctx) const {
        return formatter<string>::format(
            std::format("Student{{name = {}, age = {}, grades ={}}}",
                h.getName(), h.getAge(), h.getGrades()), ctx);
    }
};
```

```
int main() {
   Student alice{ "Alice", 20, {85, 92, 78} };
   Student bob{ "Bob", 22, {90, 88, 95, 92} };

   println("{}", alice);
   println("{}", bob);

   return 0;
}
// Output: Student{name = Alice, age = 20, grades =[85, 92, 78]}
// Output: Student{name = Bob, age = 22, grades =[90, 88, 95, 92]}
```

Explanation:

- `template <> struct formatter<Student>`
  - This tells the compiler that you're specializing the `formatter` `template` for the `Student` type.
  - `formatter<Student>` will now handle formatting for any `Student` objects.
- `std::formatter<std::string>`
  - You're inheriting from `std::formatter<std::string>`, which is the standard `formatter` for `string`s.
  - This means you're leveraging the `string` formatting capabilities to format your `Student` object.
- `auto format(Student h, format_context& ctx) const`
  - This function is the core of the custom `formatter`. It tells the compiler how to format a `Student` object.
  - `Student h` is the `Student` object that needs to be formatted.
  - `format_context& ctx` is the context where the formatting happens, but you don't need to worry about its details here.
- `std::format("Student{{name = {}, age = {}, grades ={}}}", ...)`
  - This line constructs a formatted `string` for the `Student`.
  - `std::format` is used to format the output by substituting placeholders `{}` with values (like `name`, `age`, `grades`).

- `formatter<string>::format(..., ctx)`
    - This line calls the base `string` `formatter` to actually format and output the final `string`.
    - The formatted `string` from the previous step is passed into `formatter<string>::format`, which outputs it in the desired format.

14. The `std::generator`

- C++23 introduces `std::generator`, which allows writing **lazy sequences** using coroutines.
- This makes it easier to create **iterators that generate values on demand**, reducing memory usage and improving performance.
- Before C++23, generating sequences required:
    - Precomputing all elements and storing them in a container (e.g., `std::vector`).
    - Manually managing iterators to avoid unnecessary computations.
- With `std::generator`, values are computed only when needed (lazy evaluation).

```
import std;

std::generator<int> fibonacci() {
   int a = 0, b = 1;
   while (true) {  // Infinite lazy sequence
      co_yield a; // Yield the current number
      int next = a + b;
      a = b;
      b = next;
   }
}

int main() {
   int count = 10;
   for (int num : fibonacci()) {
      if (count-- == 0) break;
      std::cout << num << " ";
   }
}// 0 1 1 2 3 5 8 13 21 34
```

- How `std::generator` Works
    - Uses `co_yield` to `return` a value without ending the function.
    - Execution pauses at `co_yield` and resumes from there the next time the `generator` is accessed.
    - Unlike `return`, `co_yield` allows continuing execution later.

15. The `std::stacktrace`

- In **C++23**, the `std::stacktrace` feature was introduced to help developers **capture and print stack traces** easily. This is particularly useful for debugging runtime issues such as crashes or exceptions.

What is a Stack Trace?

- A **stack trace** is a list of function calls that **shows how a program reached a certain point** during execution. It is commonly used to **debug errors** by tracking the sequence of function calls.
- For example, if your program crashes due to an **unhandled exception**, a **stack trace** can help you find which functions were called before the crash.

```
import std;

void functionC() {
   std::cout << "Currently Inside functionC\n";
   // Capture the stack trace here
   std::stacktrace st = std::stacktrace::current();
   // Print the stack trace
   std::cout << st << std::endl;
}

void functionB() {
   std::cout << "Inside functionB\n";
   functionC();
}

void functionA() {
   std::cout << "Inside functionA\n";
   functionB();
}

int main() {
   std::cout << "Inside main\n";
   functionA();
   return 0;
}
/*
```

```
Inside main
Inside functionA
Inside functionB
Currently Inside functionC
0> C:\Users\write\source\repos\CPP23\CPP23\FileName.cpp(6): CPP23!functionC+0x78
1> C:\Users\write\source\repos\CPP23\CPP23\FileName.cpp(13): CPP23!functionB+0x35
2> C:\Users\write\source\repos\CPP23\CPP23\FileName.cpp(18): CPP23!functionA+0x35
3> C:\Users\write\source\repos\CPP23\CPP23\FileName.cpp(23): CPP23!main+0x35
4> D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(79): CPP23!invoke_main+0x39
5> D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(288): CPP23!__scrt_common_main_seh+0x132
6> D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl(331): CPP23!__scrt_common_main+0xE
7> D:\a\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_main.cpp(17): CPP23!mainCRTStartup+0xE
8> KERNEL32!BaseThreadInitThunk+0x17
9> ntdll!RtlUserThreadStart+0x2C
*/
```

Benefits of `std::stacktrace`

- Simplifies debugging with a standard way to get stack traces.
- Helps log function call sequences in error logs.
- Portable across different operating systems.