C++ 23 Language Features

Contents

[a) Core Language Features 2](#_Toc191244771)

[1. Compile-time conditional execution - if consteval 2](#_Toc191244772)

[2. Deducing this 3](#_Toc191244773)

[b) Library Features 7](#_Toc191244774)

[3. Multidimensional std::span 7](#_Toc191244775)

[4. std::expected – Standardized way to handle errors without exceptions 8](#_Toc191244776)

[5. Monadic operations for std::optional, std::expected 9](#_Toc191244777)

[6. The std::print 14](#_Toc191244778)

[7. Convert ranges into containers 15](#_Toc191244779)

[8. Better support for constexpr in standard library algorithms 16](#_Toc191244780)

[9. Extended Algorithms 17](#_Toc191244781)

[10. Faster Sorted Containers - std::flat\_map and std::flat\_set 19](#_Toc191244782)

# Core Language Features

### 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 square(int x) {  if consteval { // Compile-time calculation  return x \* x;  }  else { // Runtime calculation  return x \* x;  }  }  int main() {  // Computed at compile-time  constexpr int result1 = square(5);  cout << "Compiletime Result1: " << result1 << "\n";  int y = 6;  // Computed at runtime  int result2 = square(y);  cout << "Runtime Result2: " << result2 << "\n";  } |

### 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.

# Library Features

### Multidimensional std::span

#### 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.

|  |
| --- |
| #include <iostream>  #include <span>  #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 Multidimensional std::span?

* In C++23, std::span supports multidimensional arrays using std::mdspan.
* This allows treating a 2D or higher-dimensional array in a structured way without extra copies.
* It improves readability and performance when working with matrices or tensors.

Even though std::mdspan is a C++23 feature, it's still relatively new, and full implementations in compilers are still in progress.

### std::expected – Standardized way to handle errors without exceptions

#### 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.

|  |
| --- |
| #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";  }  } |

### 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

|  |
| --- |
| #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. |

### 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.

|  |
| --- |
| #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.

### 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";  } |

### 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  }  } |

### 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  \*/ |

### 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.