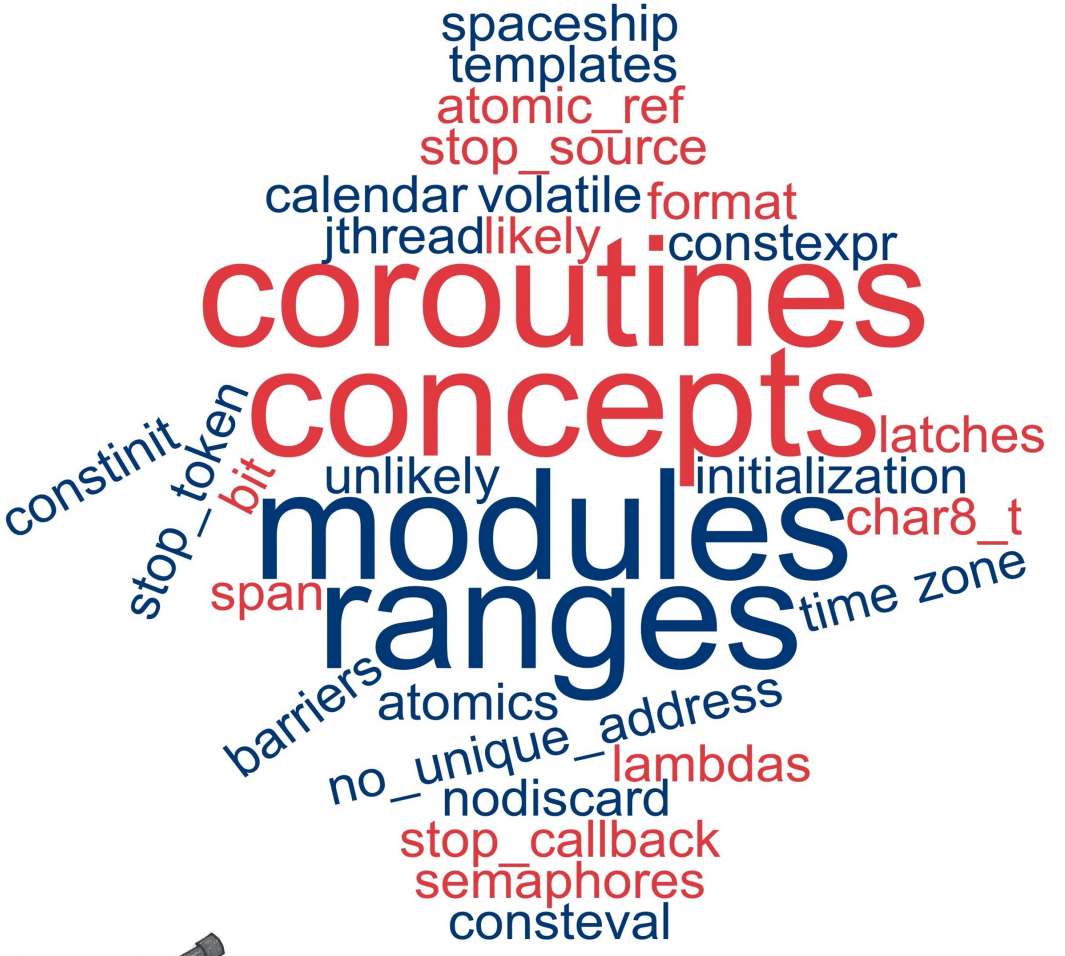
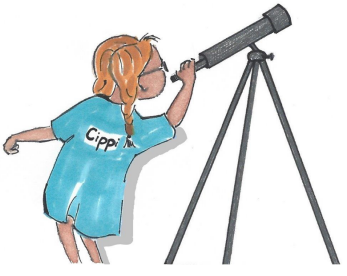
C++20

Get the Details





Rainer Grimm

ModernesCpp.com

C++20

Rainer Grimm

This book is for sale at http://leanpub.com/c20

This version was published on 2023-09-13

Contents

Reader Testimonials . . . . . **i**

Introduction . . . . . **ii**
Conventions ii
Special Fonts ii
Special Boxes iii
Source Code iii
Compilation of the Programs iii
How should you read the Book? v
Personal Notes v
Acknowledgments v
About Me vi

About C++ 1

1. Historical Context **2**
1.1 C++98 2
1.2 C++03 2
1.3 TR1 3
1.4 C++11 3
1.5 C++14 3
1.6 C++17 3

2. Standardization **4**
2.1 Stage 3 4
2.2 Stage 2 5
2.3 Stage 1 5

A Quick Overview of C++20 7

3. C++20 . . . . . **8**

3.1 The *Big Four* . . . 9
3.1.1 Concepts . . . 9
3.1.2 Modules . . . 10
3.1.3 The Ranges Library . . . 11
3.1.4 Coroutines . . . 12
3.2 Core Language . . . 14
3.2.1 Three-Way Comparison Operator . . . 14
3.2.2 Designated Initialization . . . 14
3.2.3 `consteval` and `constinit` . . . 17
3.2.4 Template Improvements . . . 18
3.2.5 Lambda Improvements . . . 19
3.2.6 New Attributes . . . 19
3.3 The Standard Library . . . 20
3.3.1 `std::span` . . . 20
3.3.2 Container Improvements . . . 21
3.3.3 Arithmetic Utilities . . . 21
3.3.4 Formatting Library . . . 21
3.3.5 Calendar and Time Zones . . . 22
3.4 Concurrency . . . 24
3.4.1 Atomics . . . 24
3.4.2 Semaphores . . . 25
3.4.3 Latches and Barriers . . . 25
3.4.4 Cooperative Interruption . . . 26
3.4.5 `std::jthread` . . . 27
3.4.6 Synchronized Outputstreams . . . 29

The Details . . . 32

4. Core Language . . . 33
4.1 Concepts . . . 34
4.1.1 Two Wrong Approaches . . . 34
4.1.2 Advantages of Concepts . . . 41
4.1.3 The long, long History . . . 42
4.1.4 Use of Concepts . . . 42
4.1.5 Constrained and Unconstrained Placeholders . . . 54
4.1.6 Abbreviated Function Templates . . . 57
4.1.7 Predefined Concepts . . . 61
4.1.8 Define Concepts . . . 69
4.1.9 Requires Expressions . . . 77
4.1.10 User-Defined Concepts . . . 81
4.2 Modules . . . 94
4.2.1 A First Example . . . 94

4.2.2 Advantages . . . 97
4.2.3 The Details . . . 104
4.2.4 Further Aspects . . . 134
4.3 Equality Comparison and Three-Way Comparison . . . 140
4.3.1 Comparison before C++20 . . . 140
4.3.2 Comparison since C++20 . . . 142
4.3.3 Comparison Categories . . . 146
4.3.4 Compiler-Generated Equality and Spaceship Operator . . . 150
4.3.5 Rewriting Expressions . . . 154
4.3.6 User-Defined and Auto-Generated Comparison Operators . . . 158
4.4 Designated Initialization . . . 160
4.4.1 Aggregate Initialization . . . 160
4.4.2 Named Initialization of Class Members . . . 161
4.5 `consteval` and `constinit` . . . 167
4.5.1 `consteval` . . . 167
4.5.2 `constinit` . . . 170
4.5.3 Comparison of `const`, `constexpr`, `consteval`, and `constinit` . . . 171
4.5.4 Solving the Static Initialization Order Fiasco . . . 174
4.6 Template Improvements . . . 180
4.6.1 Conditionally Explicit Constructor . . . 180
4.6.2 Non-Type Template Parameters (NTTP) . . . 183
4.7 Lambda Improvements . . . 188
4.7.1 Template Parameter for Lambdas . . . 188
4.7.2 Detection of the Implicit Copy of the `this` Pointer . . . 192
4.7.3 Lambdas in an Unevaluated Context and Stateless Lambdas can be Default-Constructed and Copy-Assigned . . . 194
4.7.4 `consteval` Lambdas . . . 198
4.7.5 Pack Expansion in Init-Capture . . . 199
4.8 New Attributes . . . 203
4.8.1 `[[nodiscard("reason")]]` . . . 204
4.8.2 `[[likely]]` and `[[unlikely]]` . . . 209
4.8.3 `[[no_unique_address]]` . . . 210
4.9 Further Improvements . . . 213
4.9.1 `volatile` . . . 213
4.9.2 Range-based for loop with Initializers . . . 215
4.9.3 Virtual `constexpr` function . . . 216
4.9.4 The new Character Type of UTF-8 Strings: `char8_t` . . . 217
4.9.5 `using enum` in Local Scopes . . . 219
4.9.6 Default Member Initializers for Bit Fields . . . 220

5. The Standard Library . . . 223
5.1 The Ranges Library . . . 224
5.1.1 Ranges . . . 225

5.1.2 Views . . . 230
5.1.3 Range Adaptors . . . 233
5.1.4 Direct on the Container . . . 243
5.1.5 Function Composition . . . 249
5.1.6 Lazy Evaluation . . . 251
5.1.7 Define a View . . . 255
5.1.8 `std` Algorithms versus `std::ranges` Algorithms . . . 260
5.1.9 Design Choices . . . 273
5.2 `std::span` . . . 281
5.2.1 Static versus Dynamic Extent . . . 281
5.2.2 Creation . . . 283
5.2.3 Automatically Deduces the Size of a Contiguous Sequence of Objects . . . 288
5.2.4 Modifying the Referenced Objects . . . 289
5.2.5 `std::span`'s Operations . . . 291
5.2.6 A Constant Range of Modifiable Elements . . . 293
5.2.7 Dangers of `std::span` . . . 295
5.3 Container and Algorithm Improvements . . . 298
5.3.1 `constexpr` Containers and Algorithms . . . 298
5.3.2 `std::array` . . . 301
5.3.3 Consistent Container Erasure . . . 303
5.3.4 `contains` for Associative Containers . . . 308
5.3.5 Shift the Content of a Container . . . 311
5.3.6 String prefix and suffix checking . . . 313
5.3.7 Vectorized Execution Policy: `std::execution::unseq` . . . 315
5.4 Arithmetic Utilities . . . 317
5.4.1 Safe Comparison of Integers . . . 317
5.4.2 Mathematical Constants . . . 322
5.4.3 Midpoint and Linear Interpolation . . . 324
5.4.4 Bit Manipulation . . . 327
5.5 Formatting Library . . . 333
5.5.1 Formatting Functions . . . 333
5.5.2 Format String . . . 336
5.5.3 User-Defined Types . . . 346
5.5.4 Internationalization . . . 353
5.6 Calendar and Time Zones . . . 357
5.6.1 Basic Chrono Terminology . . . 357
5.6.2 Basic Types and Literals . . . 358
5.6.3 Time of Day . . . 365
5.6.4 Calendar Dates . . . 368
5.6.5 Time Zones . . . 389
5.6.6 Chrono I/O . . . 396
5.7 Further Improvements . . . 411
5.7.1 `std::bind_front` . . . 411

5.7.2 std::is_constant_evaluated . . . 413
5.7.3 std::ssize . . . 415
5.7.4 std::source_location . . . 416
5.7.5 std::to_address . . . 418

6. Concurrency . . . 420
6.1 Coroutines . . . 421
6.1.1 A Generator Function . . . 422
6.1.2 Characteristics . . . 425
6.1.3 The Framework . . . 427
6.1.4 Awaitables and Awaiters . . . 438
6.1.5 The Workflows . . . 445
6.1.6 co_return . . . 448
6.1.7 co_yield . . . 450
6.1.8 co_await . . . 453
6.2 Atomics . . . 462
6.2.1 std::atomic_ref . . . 462
6.2.2 Atomic Smart Pointer . . . 470
6.2.3 std::atomic_flag Extensions . . . 474
6.2.4 std::atomic Extensions . . . 482
6.3 Semaphores . . . 486
6.4 Latches and Barriers . . . 491
6.4.1 std::latch . . . 491
6.4.2 std::barrier . . . 496
6.5 Cooperative Interruption . . . 501
6.5.1 std::stop_source . . . 502
6.5.2 std::stop_token . . . 503
6.5.3 std::stop_callback . . . 504
6.5.4 A General Mechanism to Send Signals . . . 507
6.5.5 Joining Threads . . . 510
6.5.6 New wait Overloads for the condition_variable_any . . . 510
6.6 std::jthread . . . 515
6.6.1 Automatically Joining . . . 516
6.6.2 Cooperative Interruption of a std::jthread . . . 519
6.7 Synchronized Output Streams . . . 522

7. Case Studies . . . 532
7.1 A Flavor of Python . . . 533
7.1.1 filter . . . 533
7.1.2 map . . . 535
7.1.3 List Comprehension . . . 536
7.2 Variations of Futures . . . 539
7.2.1 A Lazy Future . . . 541

7.2.2 Execution on Another Thread . . . 545
7.3 Modification and Generalization of a Generator . . . 550
7.3.1 Modifications . . . 554
7.3.2 Generalization . . . 557
7.3.3 Iterator Protocol . . . 560
7.4 Various Job Workflows . . . 564
7.4.1 The Transparent Awaiter Workflow . . . 564
7.4.2 Automatically Resuming the Awaiter . . . 567
7.4.3 Automatically Resuming the Awaiter on a Separate Thread . . . 570
7.5 Fast Synchronization of Threads . . . 574
7.5.1 Condition Variables . . . 575
7.5.2 `std::atomic_flag` . . . 577
7.5.3 `std::atomic<bool>` . . . 581
7.5.4 Semaphores . . . 583
7.5.5 All Numbers . . . 585

Epilogue . . . 587

Further Information . . . 588

8. C++23 and Beyond . . . 589
8.1 C++23 . . . 590
8.1.1 Core Language . . . 590
8.1.2 The Standard Library . . . 597
8.2 Beyond C++23 . . . 614
8.2.1 Contracts . . . 614
8.2.2 Reflection . . . 618
8.2.3 Pattern Matching . . . 622

9. Feature Testing . . . 625

10. Glossary . . . 637
10.1 Aggregate . . . 637
10.2 Automatic Storage Duration . . . 637
10.3 Awaitable . . . 637
10.4 Awaiter . . . 637
10.5 Callable . . . 638
10.6 Callable Unit . . . 638
10.7 Concurrency . . . 638
10.8 Critical Section . . . 638
10.9 Data Race . . . 638

10.10 Deadlock . . . 639
10.11 Dynamic Storage Duration . . . 639
10.12 Eager Evaluation . . . 639
10.13 Executor . . . 639
10.14 Function Objects . . . 639
10.15 Lambda Expressions . . . 640
10.16 Lazy Evaluation . . . 640
10.17 Literal Type . . . 640
10.18 Lock-free . . . 641
10.19 Lost Wakeup . . . 641
10.20 Math Laws . . . 641
10.21 Memory Location . . . 641
10.22 Memory Model . . . 642
10.23 Non-blocking . . . 642
10.24 Object . . . 642
10.25 Parallelism . . . 642
10.26 POD (Plain Old Data) . . . 642
10.27 Predicate . . . 642
10.28 RAII . . . 642
10.29 Race Conditions . . . 643
10.30 Regular Type . . . 643
10.31 Scalar Type . . . 643
10.32 SemiRegular . . . 643
10.33 Short-Circuit Evaluation . . . 643
10.34 Standard-Layout Type . . . 643
10.35 Static Storage Duration . . . 644
10.36 Spurious Wakeup . . . 644
10.37 The Big Four . . . 644
10.38 The Big Six . . . 645
10.39 Thread . . . 645
10.40 Thread Storage Duration . . . 645
10.41 Time Complexity . . . 645
10.42 Translation Unit . . . 645
10.43 Trivial Type . . . 646
10.44 Type Erasure . . . 646
10.45 Undefined Behavior . . . 646

Index . . . 647
A . . . 649
B . . . 650
C . . . 651
DE . . . 652
FG . . . 653

HIJKL 654
M 655
NOPR 656
S 657
T 658
UV 659
WYZ 660

Reader Testimonials

Sandor Dargo

Senior Software Development Engineer at Amadeus

"'C++ 20: Get the details' is exactly the book you need right now if you want to immerse yourself in the latest version of C++. It's a complete guide, Rainer doesn't only discuss the flagship features of C++20, but also every minor addition to the language. Luckily, the book includes tons of example code, so even if you don't have direct access yet to the latest compilers, you will have a very good idea of what you can expect from the different features. A highly recommended read!"

Adrian Tam

Director of Data Science, Synechron Inc.

"C++ has evolved a lot from its birth. With C++20, it is like a new language now. Surely this book is not a primer to teach you inheritance or overloading, but if you need to bring your C++ knowledge up to date, this is the right book. You will be surprised about the new features C++20 brought into C++. This book gives you clear explanations with concise examples. Its organization allows you to use it as a reference later. It can help you unleash the old language into its powerful future."

Introduction

My book C++20 is both a tutorial and a reference. It teaches you C++20 and provides you with the details of this new thrilling C++ standard. The thrill factor is mainly due to the big four of C++20:

- **Concepts** change the way we think about and program with templates. They are semantic categories for template parameters. They enable you to express your intention directly in the type system. If something goes wrong, the compiler gives you a clear error message.
- **Modules** overcome the restrictions of header files. They promise a lot. For example, the separation of header and source files becomes as obsolete as the preprocessor. In the end, we have faster build times and an easier way to build packages.
- The new **ranges library** supports performing algorithms directly on the containers, composing algorithms with the pipe symbol, and applying algorithms lazily on infinite data streams.
- Thanks to **coroutines**, asynchronous programming in C++ becomes mainstream. Coroutines are the basis for cooperative tasks, event loops, infinite data streams, or pipelines.

Of course, this is not the end of the story. Here are more C++20 features:

- Auto-generated comparison operators
- Calendar and time-zone libraries
- Format library
- Views on contiguous memory blocks
- Improved, interruptible threads
- Atomic smart pointers
- Semaphores
- Coordination primitives such as latches and barriers

Conventions

Here are only a few conventions.

Special Fonts

Italic: I use *Italic* to emphasize a quote.

Bold: I use **Bold** to emphasize a name.

`Monospace`: I use `Monospace` for code, instructions, keywords, and names of types, variables, functions, and classes.

Special Boxes

Boxes contain tips, warnings, and distilled information.

Tip Headline

This box provides tips and additional information about the presented material.

Warning Headline

Warning boxes should help you to avoid pitfalls.

Distilled Information

This box summarizes at the end of each main section the important things to remember.

Source Code

The source code examples–starting with the details part–shown in the book are complete. That means assuming you have a conforming compiler, you can compile and run them. I put the name of the source file in the title of each source code example. The source code uses four whitespaces for indentation. Only for layout reasons, I sometimes use two whitespaces.

Furthermore, I'm not a fan of namespace directives such as `using namespace std` because they make the code more difficult to read and pollute namespaces. Consequently, I use them only when it improves the code's readability (e.g.: `using namespaces std::chrono_literals` or `using namespace std::chrono`).

When necessary for layout reasons, I indent two characters instead of four, and I apply using directives such as `using std::chrono::Monday`. Using directives allows it to use the names unqualified: `constexpr auto monday = Monday` instead of `constexpr auto monday = std::chrono::Monday`.

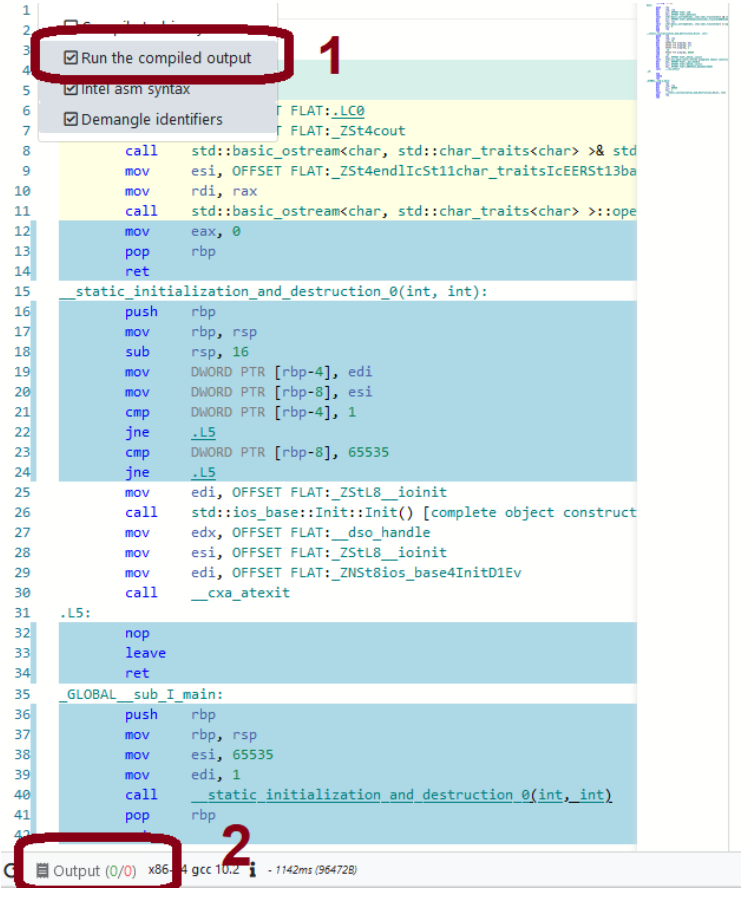
Compilation of the Programs

As the C++20 standard is brand-new, many examples can only be compiled and executed with a specific compiler. I use the newest GCC[1], Clang[2], and MSVC[3] compilers. When you compile the

[1] https://gcc.gnu.org/
[2] https://clang.llvm.org/
[3] https://en.wikipedia.org/wiki/Microsoft_Visual_C%2B%2B

program, you must specify the applied C++ standard. This means, with GCC or Clang you must provide the flag `-std=c++20`, and with MSVC `/std:c++latest`. When using concurrency features, unlike with MSVC, the GCC and Clang compilers require that you link the pthread library using `-pthread`.

If you don't have an appropriate C++ compiler at your disposal, use an online compiler such as Wandbox[4] or Compiler Explorer[5]. If you use Compiler Explorer with GCC or Clang, you can also execute the program. First, you should enable `Run the compiled output` (1) and, second, open the `Output` window (2).



Run code in the Compiler Explorer

You can get more details about the C++20 conformity of various C++ compilers at cppreference.com[6].

[4] https://wandbox.org/
[5] https://godbolt.org/
[6] https://en.cppreference.com/w/cpp/compiler_support

How should you read the Book?

If you are not familiar with C++20, start at the very beginning with a quick overview to get the big picture.

Once you get the big picture, you can proceed with the core language. The presentation of each feature should be self-contained, but reading the book from the beginning to the end would be the preferable way. On first reading, you can skip the features not mentioned in the quick overview chapter.

Personal Notes

Acknowledgments

I started a request for proofreading on my English blog: ModernesCpp Cpp[7], and received more responses than I expected. Special thanks to all of you. Here are the names of the proofreaders in alphabetic order: Bob Bird, Nicola Bombace, Dave Burchill, Sandor Dargo, James Drobina, Frank Grimm, Kilian Henneberger, Nicola Jaud-Stoll, Ivan “espkk” Kondakov, Péter Kardos, Rakesh Mane, Jonathan O’Connor, John Plaice, Iwan Smith, Paul Targosz, Steve Vinoski, and Greg Wagner.

Special thanks also to my daughter Juliette, and my wife Beatrix. Juliette improved my wording and fixed many of my typos. Beatrix created Cippi and illustrated the book.

Cippi

Let me introduce Cippi. Cippi will accompany you in this book. I hope, you like her.

[7] http://www.modernescpp.com

I'm Cippi, the C ++ Pippi Longstocking: curious, clever and - yes - feminine!

About Me

I've worked as a software architect, team lead, and instructor since 1999. In 2002, I created company-intern meetings for further education. I have given training courses since 2002. My first tutorials were about proprietary management software, but I began teaching Python and C++ soon after. In my spare time, I like to write articles about C++, Python, and Haskell. I also like to speak at conferences. I publish weekly on my English blog Modernes Cpp[8] and the German blog[9], hosted by Heise Developer.

Since 2016, I have been an independent instructor giving seminars about modern C++ and Python. I have published several books in various languages about modern C++ and, in particular, about concurrency. Due to my profession, I always search for the best way to teach modern C++.

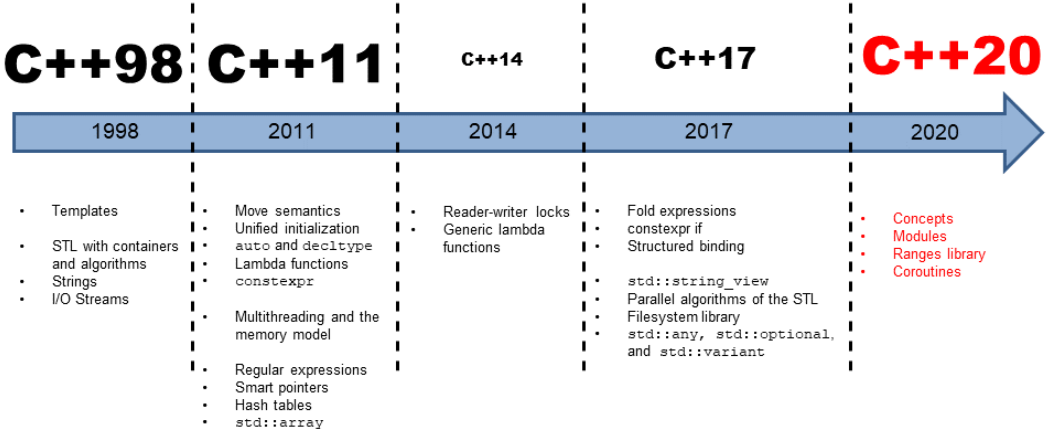
[8] https://www.modernescpp.com/
[9] https://www.grimm-jaud.de/index.php/blog

Rainer Grimm

About C++

1. Historical Context

C++20 is the next big C++ standard after C++11. Like C++11, C++20 changes the way we program in modern C++. This change mainly results from the addition of Concepts, Modules, Ranges, and Coroutines to the language. To understand this next big step in the evolution of C++, let me write a few words about the historical context of C++20.



C++ History

C++ is about 40 years old. Here is a brief overview of what has changed in the previous years.

1.1 C++98

At the end of the '80s, Bjarne Stroustrup and Margaret A. Ellis wrote their famous book Annotated C++ Reference Manual [1](ARM). This book served two purposes, to define the functionality of C++ in a world with many implementations and to provide the basis for the first C++ standard C++98 (ISO/IEC 14882). Some of the essential features of C++98 were: templates, the Standard Template Library (STL) with its containers and algorithms, strings, and IO streams.

1.2 C++03

With C++03 (14882:2003), C++98 received a technical correction, so small that is doesn't fit the timeline above. In the community, C++03, which includes C++98, is called **legacy C++**.

[1]https://www.stroustrup.com/arm.html

1.3 TR1

In 2005, something exciting happened. The so-called Technical Teport 1 (TR1) was published. TR1 was a big step toward C++11 and, therefore, towards Modern C++. TR1 (TR 19768) is based on the Boost project[2], founded by members of the C++ standardization committee. TR1 had 13 libraries destined to become part of the C++11 standard: For example, the regular expression library, the random number library, smart pointers, and hashtables. Only the so-called special mathematical functions had to wait until C++17.

1.4 C++11

We call the C++11 standard *Modern C++*. The name Modern C++ is also used for C++14 and C++17. C++11 introduced many features that fundamentally changed the way we program in C++. For example, C++11 had the additions of TR1 but also move semantics, perfect forwarding, variadic templates, and `constexpr`. But that was not all. With C++11, we also got, for the first time, a memory model as the fundamental basis of threading and the standardization of a threading API.

1.5 C++14

C++14 is a small C++ standard. It brought read-writer locks, generalized lambdas, and extended `constexpr` functions.

1.6 C++17

C++17 is neither a big nor a small C++ standard. It has two outstanding features: the parallel STL and the standardized filesystem API. About 80 algorithms of the Standard Template Library can be executed in parallel or vectorized. As with C++11, the boost libraries were highly influential for C++17. Boost provided the filesystem library and new data types: `std::string_view`, `std::optional`, `std::variant`, and `std::any`.

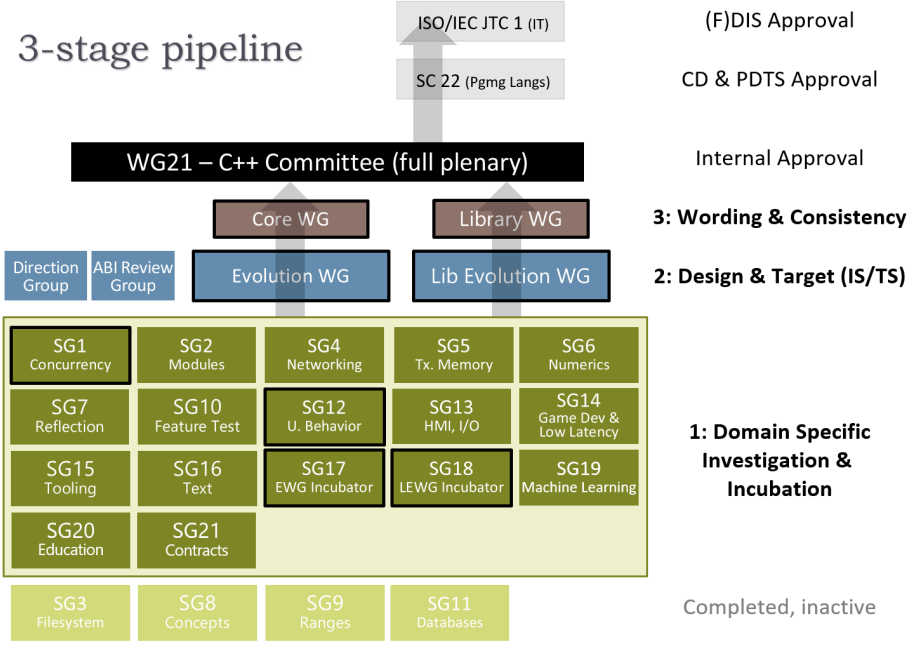
[2] https://www.boost.org/

2. Standardization

The C++ standardization process is democratic. The committee called WG21 (Working Group 21) was formed in 1990-91. The officers of WG 21 are:

- Convener: chairs the WG21, sets the meeting schedule, and appoints Study Groups
- Project Editor: applies changes to the working draft of the C++ standard
- Secretary: assigns minutes of the WG21 meetings

The image shows you the various subgroups and Study Groups of the committee.



Study groups in the C++ standardization process

The committee is organized into a three-stage pipeline consisting of several subgroups. SG stands for Study Group.

2.1 Stage 3

Stage 3 for the wording and the change proposal's consistency has two groups: core language wording (CWG) and library wording (LWG).

2.2 Stage 2

Stage 2 has two groups: core language evolution (EWG) and library evolution (LEWG). EWG and LEWG are responsible for new features that involve language and standard library extensions, respectively.

2.3 Stage 1

Stage 1 aims for domain-specific investigation and incubation. The study groups' members meet in face-to-face meetings, between the meeting by telephone or video conferences. Central groups may review the work of the study groups to ensure consistency.

These are the domain-specific Study Groups:

- **SG1, Concurrency**: Concurrency and parallelism topics, including the memory model
- **SG2, Modules**: Modules-related topics
- **SG3, File System**
- **SG4, Networking**: Networking library development
- **SG5, Transactional Memory**: Transactional memory constructs for future addition
- **SG6, Numerics**: Numerics topics such as fixed-point numbers, floating-point numbers, and fractions
- **SG7, Compile time programming**: compile time programming in general
- **SG8, Concepts**
- **SG9, Ranges**
- **SG10, Feature Test**: Portable checks to test whether a particular C++ supports a specific feature
- **SG11, Databases**: Database-related library interfaces
- **SG12, UB & Vulnerabilities**: Improvements against vulnerabilities and undefined/unspecified behavior in the standard
- **SG13, HMI & I/O (Human/Machine Interface)**: Support for output and input devices
- **SG14, Game Development & Low Latency**: Game developers and (other) low-latency programming requirements
- **SG15, Tooling**: Developer tools, including modules and packages
- **SG16, Unicode**: Unicode text processing in C++
- **SG17, EWG Incubator**: Early discussion about the core language evolution
- **SG18, LEWG Incubator**: Early discussions about the library language evolution
- **SG19, Machine Learning**: Artificial intelligence (AI) specific topics but also linear algebra
- **SG20, Education**: Guidance for modern course materials for C++ education

- **SG21, Contracts**: Language support for Design by Contract
- **SG22, C/C++ Liaison**: Discussion of C and C++ coordination

This section provided you with a concise overview of the standardization in C++ and, in particular, the C++ committee. You can find more details about the standardization at https://isocpp.org/std[1].

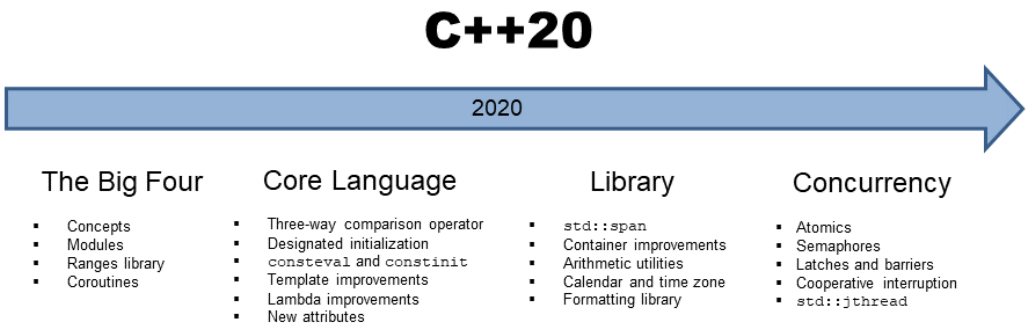
[1] https://isocpp.org/std

A Quick Overview of C++20

3. C++20

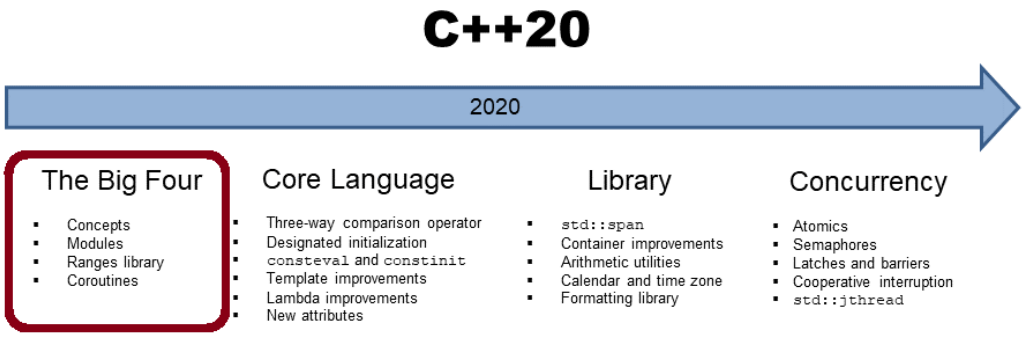
Before I dive into the details of C++20, I want to give a quick overview of C++20's features. This overview should serve two purposes; to give a first impression and to provide links to the relevant sections you can use to dive directly into the details. Consequently, this chapter has only code snippets but no complete programs.

My book starts with a short historical detour into the previous C++ standards. This detour provides context when comparing C++20 to previous revisions and demonstrates the importance of C++20 by providing a historical context.



C++20 has four outstanding features: concepts, ranges, coroutines, and modules. Each deserves its own subsection.

3.1 The *Big Four*



Each feature of the *Big Four* changes the way we program in modern C++. Let me start with concepts.

3.1.1 Concepts

Generic programming with templates enables it to define functions and classes which can be used with various types. As a result, it is not uncommon for you to instantiate a template with the wrong type. The result can be many pages of cryptic error messages. This problem ends with concepts. Concepts empower you to write requirements for template parameters that are checked by the compiler and revolutionize how we think about and write generic code. Here is why:

- Requirements for template parameters become part of their public interface.
- The overloading of functions or specializations of class templates can be based on concepts.
- We get improved error messages because the compiler checks the defined template parameter requirements against the given template arguments.

Additionally, this is not the end of the story.

- You can use predefined concepts or define your own.
- The usage of `auto` and concepts is unified. Instead of `auto`, you can use a concept.
- If a function declaration uses a concept, it automatically becomes a function template. Writing function templates is, therefore, as easy as writing a function.

The following code snippet demonstrates the definition and the use of the straightforward concept `Integral`:

Definition and use of the `Integral` concept

```
template <typename T>
concept Integral = std::is_integral<T>::value;

Integral auto gcd(Integral auto a, Integral auto b) {
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}
```

The `Integral` concept requires from its type parameter `T` that `std::is_integral<T>::value` evaluates to `true`. `std::is_integral<T>::value` is a function from the type traits library[1] checking at compile time if `T` is integral. If `std::is_integral<T>::value` evaluates to `true`, all is fine; otherwise, you get a compile-time error.

The `gcd` algorithm determines the greatest common divisor based on the Euclidean[2] algorithm. The code uses the so-called abbreviated function template syntax to define `gcd`. Here, `gcd` requires that its arguments and return type support the concept `Integral`. In other words, `gcd` is a function template that puts requirements on its arguments and return value. When I remove the syntactic sugar, you can see the real nature of `gcd`.

The semantically equivalent `gcd` algorithm using a `requires` clause.

Use of the concept `Integral` in the `requires` clause

```
template<typename T>
requires Integral<T>
T gcd(T a, T b) {
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}
```

The `requires` clause states the requirements on the type parameters of `gcd`.

3.1.2 Modules

Modules promise a lot:

- Faster compile times
- Reduce the need to define macros
- Express the logical structure of the code
- Make header files obsolete
- Get rid of ugly macro workarounds

Here is the first simple `math` module:

[1] https://en.cppreference.com/w/cpp/header/type_traits
[2] https://en.wikipedia.org/wiki/Euclid

The `math` module

```
1 export module math;
2
3 export int add(int fir, int sec) {
4     return fir + sec;
5 }
```

The expression `export module math` (line 1) is the module declaration. Putting `export` before the function `add` (line 3) exports the function. Now, it can be used by a consumer of the module.

Use of the `math` module

```
import math;

int main() {

   add(2000, 20);

}
```

The expression `import math` imports the `math` module and makes the exported names visible in the current scope.

3.1.3 The Ranges Library

The ranges library supports algorithms which

- can operate directly on containers; you don't need iterators to specify a range
- can be evaluated lazily
- can be composed

To make it short: The ranges library supports functional patterns.

The following example demonstrates function composition using the pipe symbol.

Function composition with the pipe symbol

```
1  int main() {
2      std::vector<int> ints{0, 1, 2, 3, 4, 5};
3      auto even = [](int i){ return i % 2 == 0; };
4      auto square = [](int i) { return i * i; };
5
6      for (int i : ints | std::views::filter(even) |
7                          std::views::transform(square)) {
8          std::cout << i << ' ';             // 0 4 16
9      }
10 }
```

Lambda expression `even` (line 3) is a lambda expression that returns `true` if an argument `i` is even. Lambda expression `square` (line 4) maps the argument `i` to its square. Lines 6 and 7 demonstrate function composition, which you have to read from left to right: `for (int i : ints | std::views::filter(even) | std::views::transform(square))`. Apply on each element of `ints` the `even` filter and map each remaining element to its square. If you are familiar with functional programming, this reads like prose.

3.1.4 Coroutines

Coroutines are generalized functions that can be suspended and resumed later while maintaining their state. Coroutines are a convenient way to write event-driven applications. Event-driven applications can be simulations, games, servers, user interfaces, or even algorithms. Coroutines are also typically used for cooperative multitasking.

C++20 does not provide concrete coroutines, but C++20 provides a framework for implementing coroutines. This framework consists of more than 20 functions, and some of which you must implement, some of which you can override. Therefore, you can tailor coroutines to your needs.

The following code snippet uses a generator to create a potentially infinite data stream. The coroutines chapter provides the implemenation of the `Generator`.

A generator for an infinite data-stream

```
1  Generator<int> getNext(int start = 0, int step = 1){
2      auto value = start;
3      while (true) {
4          co_yield value;
5          value += step;
6      }
7  }
8
9  int main() {
10
```

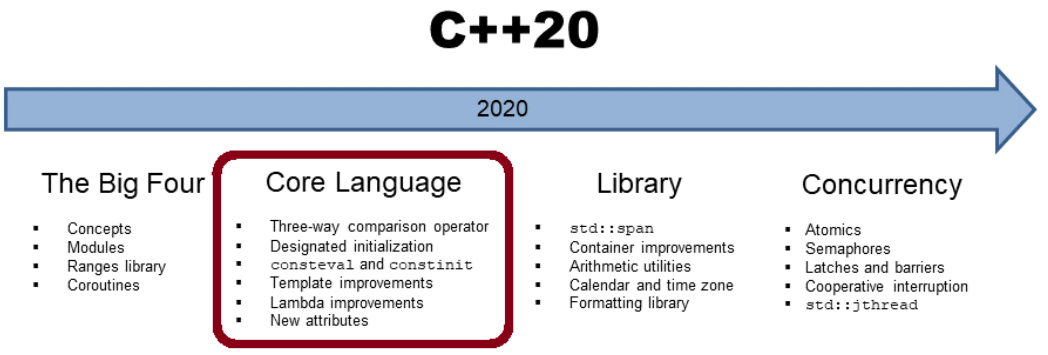
```
11      std::cout << '\n';
12
13      std::cout << "getNext():";
14      auto gen1 = getNext();
15      for (int i = 0; i <= 10; ++i) {
16          gen1.next();
17          std::cout << " " << gen1.getValue();
18      }
19
20      std::cout << "\n\n";
21
22      std::cout << "getNext(100, -10):";
23      auto gen2 = getNext(100, -10);
24      for (int i = 0; i <= 20; ++i) {
25          gen2.next();
26          std::cout << " " << gen2.getValue();
27      }
28
29      std::cout << "\n";
30
31  }
```

The function `getNext` is a coroutine because it uses the keyword `co_yield`. There is an infinite loop that returns the value at `co_yield` (line 4). A call to `next` (lines 16 and 25) resumes the coroutine and the following `getValue` call gets the value. After the `getNext` call returns, the coroutine pauses once again until the next call `next`. There is one big unknown in this example: the return value `Generator<int>` of the `getNext` function. This is where the complication begins, which I describe in full depth in the coroutines section.

```
Start

getNext(): 0 1 2 3 4 5 6 7 8 9 10

getNext(100, -10): 100 90 80 70 60 50 40 30 20 10 0 -10 -20 -30 -40 -50 -60 -70 -80 -90 -100

0

Finish
```

An infinite data-generator

3.2 Core Language



3.2.1 Three-Way Comparison Operator

The three-way comparison operator `<=>`, or spaceship operator, determines, for two values A and B, whether A < B, A == B, or A > B.

By declaring the three-way comparison operator `default`, the compiler will attempt to generate a consistent relational operator for the class. In this case, you get all six comparison operators: ==, !=, <, <=, >, and >=.

Auto-generating the three-way comparison operator

```
struct MyInt {
  int value;
  MyInt(int value): value{value} { }
  auto operator<=>(const MyInt&) const = default;
};
```

The compiler-generated `operator <=>` performs a lexicographical comparison, starting with the base classes and taking into account all the non-static data members in their declaration order. Here is a quite sophisticated example from the Microsoft blog: Simplify Your Code with Rocket Science: C++20's Spaceship Operator[3].

3.2.2 Designated Initialization

[3] https://devblogs.microsoft.com/cppblog/simplify-your-code-with-rocket-science-c20s-spaceship-operator/

Spaceship operator for derived classes

```
struct Basics {
  int i;
  char c;
  float f;
  double d;
  auto operator<=>(const Basics&) const = default;
};

struct Arrays {
  int ai[1];
  char ac[2];
  float af[3];
  double ad[2][2];
  auto operator<=>(const Arrays&) const = default;
};

struct Bases : Basics, Arrays {
  auto operator<=>(const Bases&) const = default;
};

int main() {
  constexpr Bases a = { { 0, 'c', 1.f, 1. },
    { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, { { 1., 2. }, { 3., 4. } } } };
  constexpr Bases b = { { 0, 'c', 1.f, 1. },
    { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, { { 1., 2. }, { 3., 4. } } } };
  static_assert(a == b);
  static_assert(!(a != b));
  static_assert(!(a < b));
  static_assert(a <= b);
  static_assert(!(a > b));
  static_assert(a >= b);
}
```

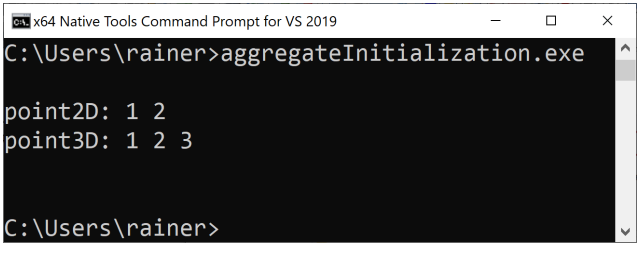
I assume the most complicated stuff in this code snippet is not the spaceship operator but the initialization of `Base` using aggregate initialization. Aggregate initialization essentially means that you can directly initialize the members of class types (`class`, `struct`, or `union`) if all members are `public`. In this case, you can use a braced initialization list, as in the example.

Before I discuss **designated initialization**, let me show more about aggregate initialization. Here is a straightforward example.

Aggregate initialization

```
struct Point2D{
  int x;
  int y;
};

class Point3D{
public:
  int x;
  int y;
  int z;
};

int main(){

  std::cout << "\n";

  Point2D point2D {1, 2};
  Point3D point3D {1, 2, 3};

  std::cout << "point2D: " << point2D.x << " " << point2D.y << "\n";
  std::cout << "point3D: " << point3D.x << " "
            << point3D.y << " " << point3D.z << "\n";

  std::cout << '\n';

}
```

This is the output of the program:



Aggregate initialization

The aggregate initialization is quite error-prone, because you can swap the constructor arguments without realizing it. Explicit is better than implicit. Let's see what that means. Take a look at how designated initializers from C99[4], now part of the C++ standard, intervene.

[4] https://en.wikipedia.org/wiki/C99

Designated initialization

```
1  struct Point2D{
2    int x;
3    int y;
4  };
5
6  class Point3D{
7  public:
8    int x;
9    int y;
10   int z;
11 };
12
13 int main(){
14
15   Point2D point2D {.x = 1, .y = 2};
16   // Point2D point2d {.y = 2, .x = 1};          // error
17   Point3D point3D {.x = 1, .y = 2, .z = 2};
18   // Point3D point3D {.x = 1, .z = 2}           // {1, 0, 2}
19
20
21   std::cout << "point2D: " << point2D.x << " " << point2D.y << "\n";
22   std::cout << "point3D: " << point3D.x << " " << point3D.y << " " << point3D.z
23             << "\n";
24
25 }
```

The arguments for the instances of `Point2` and `Point3D` are explicitly named. The output of the program is identical to the output of the previous one. The commented-out lines 16 and 18 are quite interesting. Line 16 would give an error because the order of the designators does not match the declaration order of the data members. As for line 18, the designator for `y` is missing. In this case, `y` is initialized to 0, such as when using braced initialization list {1, 0, 3}.

3.2.3 `consteval` and `constinit`

The new **`consteval`** specifier added in C++20 creates an immediate function. For an immediate function, each invocation of the function must produce a compile-time constant expression. An immediate function is implicitly a `constexpr` function but not necessarily the other way around.

An immediate function

```
consteval int sqr(int n) {
    return n*n;
}
constexpr int r = sqr(100);  // OK

int x = 100;
int r2 = sqr(x);             // Error
```

The final assignment gives an error because `x` is not a constant expression and, therefore, `sqr(x)` cannot be performed at compile time.

constinit ensures that the variable with static storage duration or thread storage duration is initialized at compile time. Static storage duration means that the object is allocated when the program begins and is deallocated when the program ends. Thread storage duration means that the object's lifetime is bound to the lifetime of the thread.

`constinit` ensures for this kind of variable (static storage duration or thread storage duration) that they are initialized at compile time. `constinit` does not imply constness.

3.2.4 Template Improvements

C++20 offers **various improvements** to programming with templates. A generic constructor is a catch-all constructor because you can invoke it with any type.

An implicit and explicit generic constructor

```
struct Implicit {
    template <typename T>
    Implicit(T t) {
        std::cout << t << '\n';
    }
};

struct Explicit {
    template <typename T>
    explicit Explicit(T t) {
        std::cout << t << '\n';
    }
};

Explicit exp1 = "implicit";  // Error
Explicit exp2{"explicit"};
```

The generic constructor of the class `Implicit` is way too generic. By putting the keyword `explicit` in front of the constructor, as for `Explicit`, the constructor becomes explicit. This means that implicit conversions are not valid anymore.

3.2.5 Lambda Improvements

Lambdas get many improvements in C++20. They can have template parameters and can be used in unevaluated contexts, and stateless lambdas can also be default-constructed and copy-assigned. Furthermore, the compiler can now detect when you implicitly copy the `this` pointer, which means a significant cause of undefined behavior with lambdas is gone.

If you want to define a lambda that accepts only a `std::vector`, template parameters for lambdas enable this:

Template parameters for lambdas

```
auto foo = []<typename T>(std::vector<T> const& vec) {
        // do vector-specific stuff
    };
```

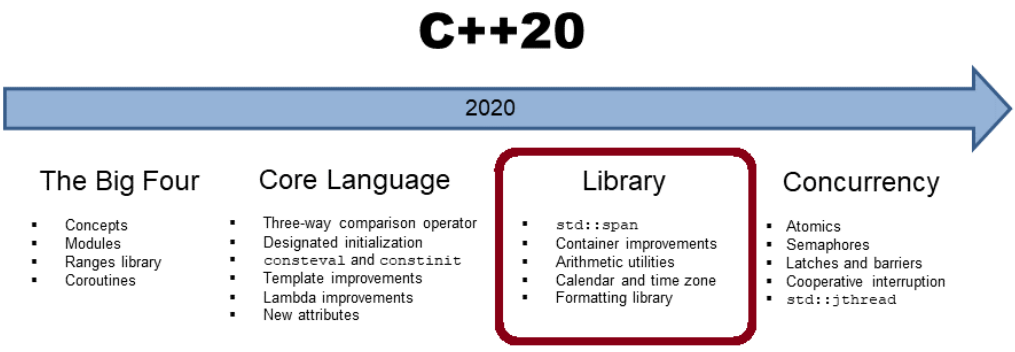
3.2.6 New Attributes

C++20 has **new attributes**, including `[[likely]]` and `[[unlikely]]`. Both attributes allow us to give the optimizer a hint, specifying which path of execution is more or less likely.

The attribute `[[likely]]`

```
for(size_t i=0; i < v.size(); ++i){
    if (v[i] < 0) [[likely]] sum -= sqrt(-v[i]);
    else sum += sqrt(v[i]);
}
```

3.3 The Standard Library



3.3.1 `std::span`

A **`std::span`** represents an object that can refer to a contiguous sequence of objects. A `std::span`, sometimes also called a view, is never an owner. This view can be a C-array, a `std::array`, a pointer with a size, or a `std::vector`. A typical implementation of a `std::span` needs a pointer to its first element and a size. The main reason for having a `std::span` is that a plain array will decay to a pointer if passed to a function; therefore, its size is lost. `std::span` automatically deduces the size of an array, a `std::array`, or a `std::vector`. If you use a pointer to initialize a `std::span`, you have to provide the size in the constructor.

`std::span` as function argument

```
void copy_n(const int* src, int* des, int n){}

void copy(std::span<const int> src, std::span<int> des){}

int main(){

  int arr1[] = {1, 2, 3};
  int arr2[] = {3, 4, 5};

  copy_n(arr1, arr2, 3);
  copy(arr1, arr2);

}
```

Compared to the function `copy_n`, `copy` doesn't need the number of elements. Hence, a common cause of errors is gone with `std::span<T>`.

3.3.2 Container Improvements

C++20 has many improvements regarding containers of the Standard Template Library. First of all, `std::vector` and `std::string` have **constexpr constructors** and can, therefore, be used at compile time. All standard library containers support **consistent container erasure**, and the associative containers support a **`contains`** member function. Additionally, `std::string` allows **checking for a prefix or suffix**.

3.3.3 Arithmetic Utilities

The comparison of signed and unsigned integers is a subtle cause of unexpected behavior and, therefore, of bugs. Thanks to the new **safe comparison functions for integers**, `std::cmp_*`, a subtle source of bugs is gone.

Safe comparison of integers

```
int x = -3;
unsigned int y = 7;

if (x < y) std::cout << "expected";
else std::cout << "not expected";                  // not expected

if (std::cmp_less(x, y)) std::cout << "expected";  // expected
else std::cout << "not expected";
```

Additionally, C++20 includes **mathematical constants**, including e, π, or ϕ in the namespace `std::numbers`.

The new bit manipulation enables accessing individual bits and bit sequences and reinterpreting them.

Accessing individual bits and bit sequences

```
std::uint8_t num= 0b10110010;

std::cout << std::has_single_bit(num) << '\n';            // false
std::cout << std::bit_width(unsigned(5)) << '\n';         // 3
std::cout << std::bitset<8>(std::rotl(num, 2)) << '\n';   // 11001010
std::cout << std::bitset<8>(std::rotr(num, 2)) << '\n';   // 10101100
```

3.3.4 Formatting Library

The **new formatting library** provides a safe and extensible alternative to the `printf` functions. It's intended to complement the existing I/O streams and reuse some of its infrastructure, such as overloaded insertion operators for user-defined types.

```
std::string message = std::format("The answer is {}.", 42);
```

`std::format` uses Python's syntax for formatting. The following examples show a few typical use cases:

- Format and use positional arguments

```
std::string s = std::format("I'd rather be {1} than {0}.", "right", "happy");
// s == "I'd rather be happy than right."
```

- Convert an integer to a string in a safe way

```
memory_buffer buf;
std::format_to(buf, "{}", 42);    // replaces itoa(42, buffer, 10)
std::format_to(buf, "{:x}", 42);  // replaces itoa(42, buffer, 16)
```

- Format user-defined types

3.3.5 Calendar and Time Zones

The chrono library[5] from C++11 is extended with **calendar and time-zone** functionality. The calendar consists of types representing a year, a month, a day of the week, and an n-th weekday of a month. These elementary types can be combined into complex types such as `year_month`, `year_month_day`, `year_month_day_last`, `year_month_weekday`, and `year_month_weekday_last`. The operator "/" is overloaded for the convenient specification of time points. Additionally, we get new literals: `d` for a day and `y` for a year.

Time points can be displayed in various time zones. Due to the extended chrono library, the following use cases are now trivial to implement:

- representing dates in specific formats
- get the last day of a month
- get the number of days between two dates
- printing the current time in various time zones

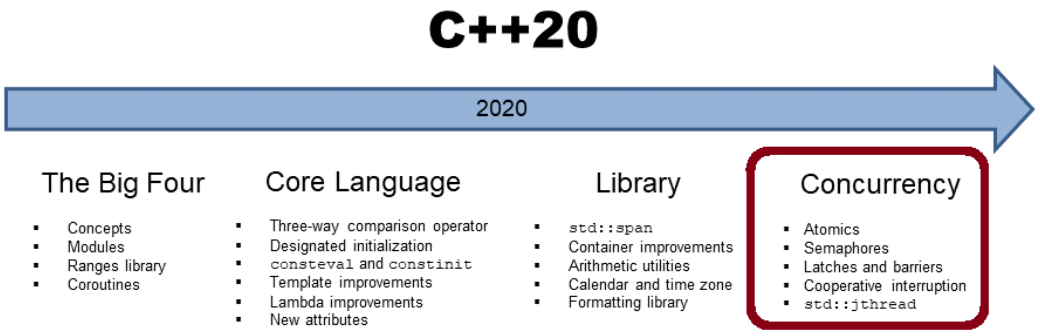
The following program presents the local time in different time zones.

[5] https://en.cppreference.com/w/cpp/chrono

The local time in various time zones

```
using namespace std::chrono;

auto time = floor<milliseconds>(system_clock::now());
auto localTime = zoned_time<milliseconds>(current_zone(), time);
auto berlinTime = zoned_time<milliseconds>("Europe/Berlin", time);
auto newYorkTime = zoned_time<milliseconds>("America/New_York", time);
auto tokyoTime = zoned_time<milliseconds>("Asia/Tokyo", time);

std::cout << time << '\n';        // 2020-05-23 19:07:20.290
std::cout << localTime << '\n';   // 2020-05-23 21:07:20.290 CEST
std::cout << berlinTime << '\n';  // 2020-05-23 21:07:20.290 CEST
std::cout << newYorkTime << '\n'; // 2020-05-23 15:07:20.290 EDT
std::cout << tokyoTime << '\n';   // 2020-05-24 04:07:20.290 JST
```

3.4 Concurrency



3.4.1 Atomics

The class template **`std::atomic_ref`** applies atomic operations to the referenced non-atomic object. Concurrent writing and reading of the referenced object can take place, therefore, with no data race. The lifetime of the referenced object must exceed the lifetime of the `std::atomic_ref`. Accessing a subobject of the referenced object with `std::atomic_ref` is not thread-safe.

According to `std::atomic`[6], `std::atomic_ref` can be specialized and supports specializations for the built-in data types.

```
struct Counter {
    int a;
    int b;
};

Counter counter;

std::atomic_ref<Counter> cnt(counter);
```

With C++20, we get two **atomic smart pointers** that are partial specializations of `std::atomic`: there are `std::atomic<std::shared_ptr<T>>` and `std::atomic<std::weak_ptr<T>>`. Both atomic smart pointers guarantee that not only the control block, as in the case of std::shared_ptr[7], is thread-safe, but also the associated object.

`std::atomic` gets more extensions. C++20 provides specializations for atomic floating-point types. This is quite convenient when you have a concurrently incremented floating-point type.

[6] https://en.cppreference.com/w/cpp/atomic/atomic
[7] https://en.cppreference.com/w/cpp/memory/shared_ptr

A value of type `std::atomic_flag`[8] is a kind of atomic boolean. It has a cleared and set state. For simplicity reasons, I call the clear state `false` and the set state `true`. The `clear()` member function enables you to set its value to `false`. With the `test_and_set()` member function, you can set the value to `true` and get the previous value. There is no member function to ask for the current value. This will change with C++20 because `std::atomic_flag` has a `test()` method.

Furthermore, `std::atomic_flag` can be used for thread synchronization via the member functions `notify_one()`, `notify_all()`, and `wait()`. With C++20, notifying and waiting are available on all partial and full specializations of `std::atomic` and `std::atomic_ref`. Specializations are available for bools, integrals, floats, and pointers.

3.4.2 Semaphores

Semaphores are a synchronization mechanism used to control concurrent access to a shared resource. A counting semaphore, such as the one which was added in C++20, is a special semaphore whose initial counter is bigger than zero. The counter is initialized in the constructor. Acquiring the semaphore decreases the counter, and releasing the semaphore increases the counter. If a thread tries to acquire the semaphore when the counter is zero, the thread blocks until another thread increments the counter by releasing the semaphore.

3.4.3 Latches and Barriers

Latches and barriers are straightforward thread synchronization mechanisms that enable some threads to block until a counter becomes zero. What are the differences between these two mechanisms to synchronize threads? You can use a `std::latch` only once, but you can use a `std::barrier` more than once. A `std::latch` is useful for managing one task by multiple threads; a `std::barrier` is useful for managing repeated tasks by multiple threads. Furthermore, a `std::barrier` can adjust the counter in each iteration.

The following is based on a code snippet from proposal N4204[9]. I fixed a few typos and reformatted it.

Thread-synchronization with a `std::latch`

```
void DoWork(threadpool* pool) {

    std::latch completion_latch(NTASKS);
    for (int i = 0; i < NTASKS; ++i) {
        pool->add_task([&] {
            // perform work
            ...
            completion_latch.count_down();
        });
```

[8] https://en.cppreference.com/w/cpp/atomic/atomic_flag
[9] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4204.html

```
10      }
11      // Block until work is done
12      completion_latch.wait();
13  }
```

The counter of the `std::latch completion_latch` is set to `NTASKS` (line 3). The thread pool executes `NTASKS` jobs (lines 4 - 10). At the end of each job, the counter is decremented (line 8). The thread running function `DoWork` blocks in line 12 until all tasks are done.

3.4.4 Cooperative Interruption

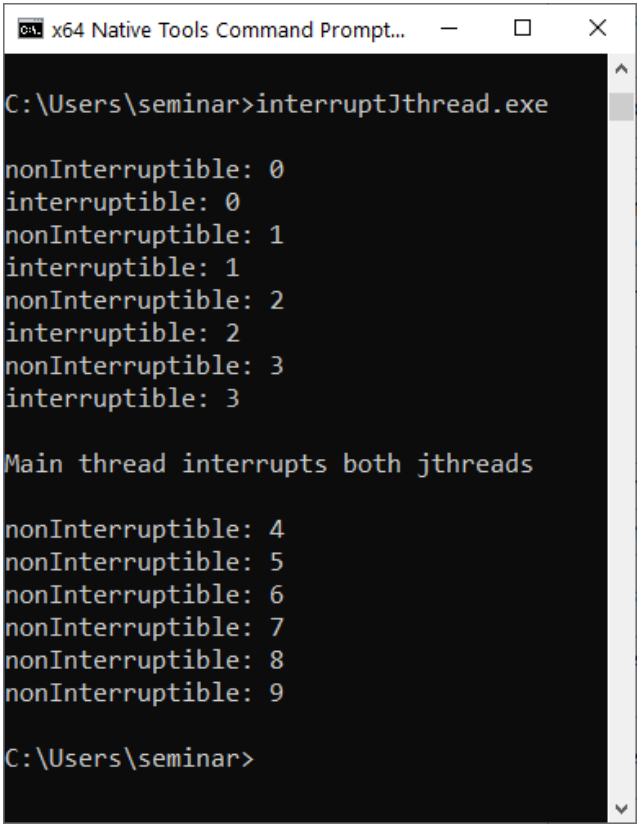
Thanks to **`std::stop_token`**, a `std::jthread` can be interrupted cooperatively.

Interrupting a `std::jthread`

```
1   int main() {
2
3       std::cout << '\n';
4
5       std::jthread nonInterruptible([]{
6           int counter{0};
7           while (counter < 10){
8               std::this_thread::sleep_for(0.2s);
9               std::cerr << "nonInterruptible: " << counter << '\n';
10              ++counter;
11          }
12      });
13
14      std::jthread interruptible([](std::stop_token stoken){
15          int counter{0};
16          while (counter < 10){
17              std::this_thread::sleep_for(0.2s);
18              if (stoken.stop_requested()) return;
19              std::cerr << "interruptible: " << counter << '\n';
20              ++counter;
21          }
22      });
23
24      std::this_thread::sleep_for(1s);
25
26      std::cerr << '\n';
27      std::cerr << "Main thread interrupts both jthreads" << std:: endl;
28      nonInterruptible.request_stop();
29      interruptible.request_stop();
30
```

```
31      std::cout << '\n';
32
33  }
```

The `main` program starts two threads, `nonInterruptible` and `interruptible` (lines 5 and 14). Only thread `interruptible` gets a `std::stop_token`, which it uses in line 18 to check if it is interrupted. The lambda immediately returns in case of an interruption. The call to `interruptible.request_stop()` triggers the cancellation of the thread. Calling `nonInterruptible.request_stop()` has no effect.



Cooperative interruption of a thread

3.4.5 `std::jthread`

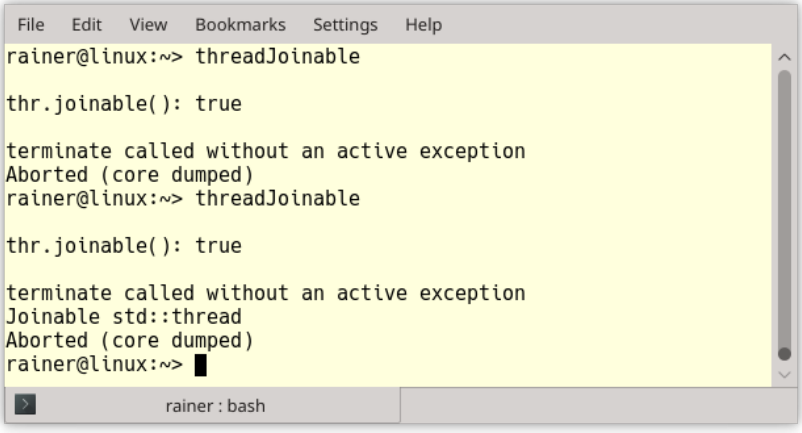
`std::jthread` stands for joining thread. std::jthread extends `std::thread`[10] by automatically joining the started thread. `std::jthread` can also be interrupted.

[10] https://en.cppreference.com/w/cpp/thread/thread

`std::jthread` is added to the C++20 standard because of the non-intuitive behavior of `std::thread`. If a `std::thread` is still joinable, `std::terminate`[11] is called in its destructor. A thread `thr` is joinable if neither `thr.join()` nor `thr.detach()` was called.

Thread `thr` is still joinable

```
int main() {

    std::cout << '\n';

    std::cout << std::boolalpha;
    std::thread thr{[]{ std::cout << "Joinable std::thread" << '\n'; }};
    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';

}
```



`std::terminate` with a still joinable thread

Both executions of the program terminate. In the second run, the thread `thr` has enough time to display its message: "Joinable std::thread".

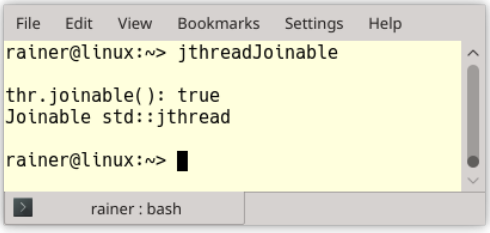
In the modified example, I use `std::jthread` from the C++20 standard.

[11] https://en.cppreference.com/w/cpp/error/terminate

Thread `thr` joins automatically

```
int main() {

    std::cout << '\n';

    std::cout << std::boolalpha;
    std::jthread thr{[]{ std::cout << "Joinable std::jthread" << '\n'; }};
    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';

}
```

Now, thread `thr` automatically joins in its destructor if necessary.



Thread `thr` joins automatically

3.4.6 Synchronized Outputstreams

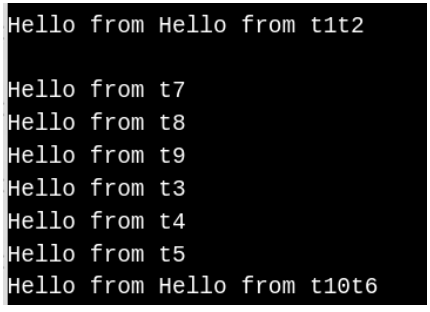
With C++20, we get synchronized outputstreams. What happens when more threads write concurrently to `std::cout` without synchronization?

Unsynchronized writing to `std::cout`

```
void sayHello(std::string name) {
    std::cout << "Hello from " << name <<  '\n';
}

int main() {

    std::cout << "\n";

   std::jthread t1(sayHello, "t1");
   std::jthread t2(sayHello, "t2");
   std::jthread t3(sayHello, "t3");
```

```
    std::jthread t4(sayHello, "t4");
    std::jthread t5(sayHello, "t5");
    std::jthread t6(sayHello, "t6");
    std::jthread t7(sayHello, "t7");
    std::jthread t8(sayHello, "t8");
    std::jthread t9(sayHello, "t9");
    std::jthread t10(sayHello, "t10");

     std::cout << '\n';

}
```

You may get a mess.

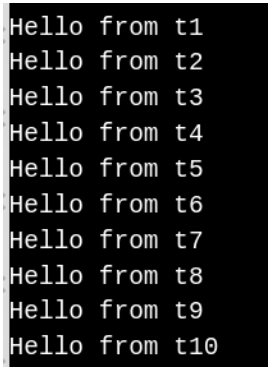


Unsynchronized writing to `std::cout`

Switching from `std::cout` in the function `sayHello` to `std::osyncstream(std::cout)` turns the mess into harmony.

Synchronized writing to `std::cout`

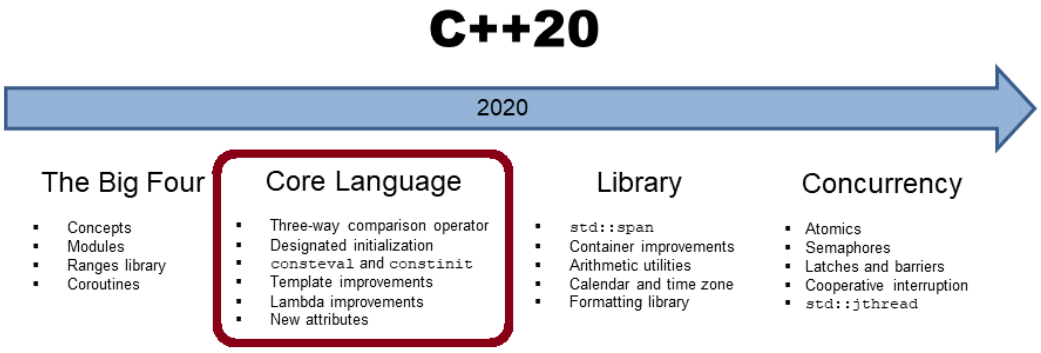
```
void sayHello(std::string name) {
    std::osyncstream(std::cout) << "Hello from " << name <<  '\n';
}
```



Synchronized writing to `std::cout`

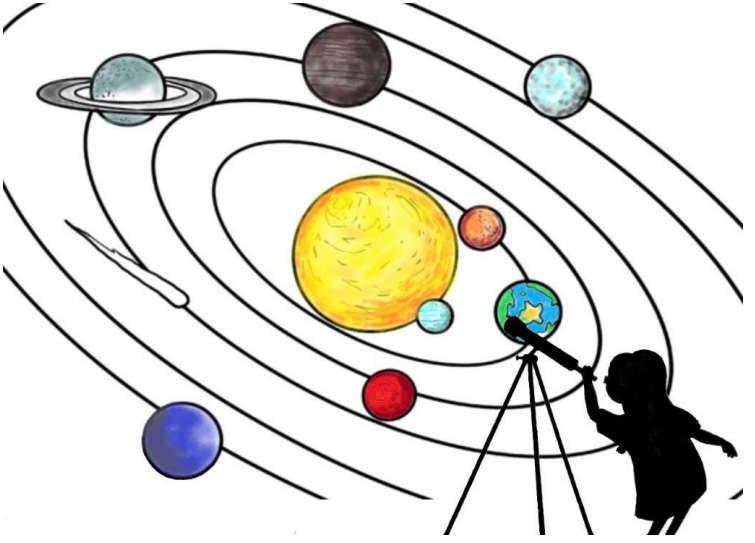
The Details

4. Core Language



Concepts are one of the most impactful features of C++20. Consequently, it is an ideal starting point to present the core language features of C++20.

4.1 Concepts



Cippi studies the stars

To appreciate the impact of concepts to their full extent, I want to start with a short motivation for concepts.

4.1.1 Two Wrong Approaches

Before C++20, we had two opposed ways to think about functions or classes: defining them for specific types or defining them for generic types. In the latter case, we call them function templates or class templates. Both approaches have their own set of problems:

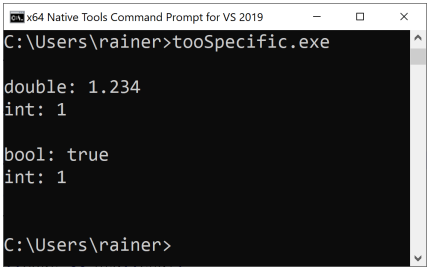
4.1.1.1 Too Specific

It's tedious work to overload a function or reimplement a class for each type. To avoid that burden, type conversion often comes to our rescue. What seems like rescue is often a curse.

Implicit conversions

```
1  // tooSpecific.cpp
2
3  #include <iostream>
4
5  void needInt(int i){
6      std::cout << "int: " << i << '\n';
7  }
8
9  int main(){
10
11     std::cout << std::boolalpha << '\n';
12
13     double d{1.234};
14     std::cout << "double: " << d << '\n';
15     needInt(d);
16
17     std::cout << '\n';
18
19     bool b{true};
20     std::cout << "bool: " << b << '\n';
21     needInt(b);
22
23     std::cout << '\n';
24
25 }
```

In the first case (line 13), I start with a `double` and end with an `int` (line 15). In the second case, I start with a `bool` (line 19) and nd with an `int` (line 21).



Implicit conversions

The program exemplifies two implicit conversions.

4.1.1.1.1 Narrowing Conversion

Invoking `getInt(int a)` with a `double` gives you a narrowing conversion. Narrowing conversion is a conversion, including a loss of accuracy. I assume this is not what you want.

4.1.1.1.2 Integral Promotion

But the other way around is also not better. Invoking `getInt(int a)` with a `bool` promotes the `bool` to an `int`. Surprised? Many C++ developers don't know which data type they get when they add two `bool`s.

Adding two `bool`s

```
template <typename T>
auto add(T first, T second){
    return first + second;
}

int main(){
    add(true, false);
}
```

C++ Insights[1] visualizes the source code above after the compiler transformed the function template in an instantiation.

[1] https://cppinsights.io/s/9bd14f99

```
1 template <typename T>
2 auto add(T first, T second){
3     return first + second;
4 }
5
6 #ifdef INSIGHTS_USE_TEMPLATE
7 template<>
8 int add<bool>(bool first, bool second)
9 {
10   return static_cast<int>(first) + static_cast<int>(second);
11 }
12 #endif
13
14
15 int main()
16 {
17   add(true, false);
18 }
```

`bool` to `int` promotion

Lines 6 - 12 are the crucial ones in this screenshot of C++ Insights[2]. The template instantiation of the function template `add` creates a full specialization with the return type `int`. Both `bool`s are implicitly promoted to `int`.

My belief is that we rely for convenience on the magic of conversions because we don't want to overload a function or reimplement a class for each type.

Let me try the other way and use a generic function. Maybe this is our rescue?

4.1.1.2 Too Generic

Sorting a container is a general idea. It should work for each container if its elements support ordering. In the following example, I apply the standard algorithm `std::sort` to the standard container `std::list`.

[2] https://cppinsights.io/

Sorting a `std::list`

```cpp
// tooGeneric.cpp

#include <algorithm>
#include <list>

int main(){

    std::list<int> myList{1, 10, 3, 2, 5};

    std::sort(myList.begin(), myList.end());

}
```

A compiler error when trying to sort a `std::list`

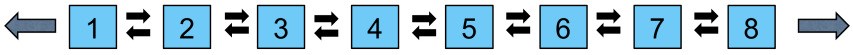
I don't even want to decipher this long message. What's gone wrong? Let's take a look at the signature of the specific overload of `std::sort`[3] used in this example.

---

[3]https://en.cppreference.com/w/cpp/algorithm/sort

```
template< class RandomIt >
constexpr void sort( RandomIt first, RandomIt last );
```

`std::sort` uses strange-named argument types such as `RandomIt`. `RandomIt` stands for a random-access iterator and gives the decisive hint for the overwhelming error message. A `std::list` only provides a bidirectional iterator, but `std:sort` requires a random-access iterator. The following graphic shows why a `std::list` does not support a random access iterator.



The structure of a `std::list`

If you study the `std::sort` documentation on cppreference.com, you will find something exciting: type requirements on template parameters. They place conceptual requirements on the types that have been formalized into the C++20 feature: concepts.

4.1.1.3 Concepts to the Rescue

Concepts are compile-time predicates. They put semantic constraints on template parameters. `std::sort` has overloads that accept a comparator.

```
template< class RandomIt, class Compare >
constexpr void sort(RandomIt first, RandomIt last, Compare comp);
```

These are the type requirements for the more powerful overload of `std::sort`:

- `RandomIt` must meet the requirements of `ValueSwappable` and `LegacyRandomAccessIterator`.
- The type of the dereferenced `RandomIt` must meet the requirements of `MoveAssignable` and `MoveConstructible`.
- The type of the dereferenced `RandomIt` must meet the requirements of `Compare`.

Requirements such as `ValueSwappable` or `LegacyRandomAccessIterator` are so-called named requirements. Some of these requirements are formalized in C++20 in concepts[4].

In particular, `std::sort` requires a `LegacyRandomAccessIterator`. Let's have a closer look at the named requirement `LegacyRandomAccessIterator` that is called `random_access_iterator` (part of `<iterator>`) in C++20:

[4] https://en.cppreference.com/w/cpp/language/constraints

std::random_access_iterator

```
template<class I>
    concept random_access_iterator =
        bidirectional_iterator<I> &&
        derived_from<ITER_CONCEPT(I), random_access_iterator_tag> &&
        totally_ordered<I> &&
        sized_sentinel_for<I, I> &&
        requires(I i, const I j, const iter_difference_t<I> n) {
            { i += n } -> same_as<I&>;
            { j + n } -> same_as<I>;
            { n + j } -> same_as<I>;
            { i -= n } -> same_as<I&>;
            { j - n } -> same_as<I>;
            { j[n] } -> same_as<iter_reference_t<I>>;
        };
```

A type `I` supports the concept `random_access_iterator` if it supports the `bidirectional_iterator` concept and all the following requirements. For example, the requirement `{ i += n } -> same_as<I&>` as part of the `requires` expression means that for a value of type `I`, `{ i += n }` is a valid expression, and it returns a value of type `I&`. To complete the sorting story, `std::list` does support a `bidirectional_iterator` and not a `random_access_iterator` that `std::sort` requires.

When you now use an algorithm that requires a `random_access_iterator`, but you only provide a `birectional_iterator`, you get a concise and readable error message saying that your iterator does not satisfy the concept `random_access_iterator`.

The Standard Template Library

The Essence of Generic Programming

I want to start this short historical detour with a quote from the invaluable book **From Mathematics to Generic Programming**[5], written by Alexander Stepanov (creator of the Standard Template Library) and Daniel Rose (information retrieval researcher): "*The essence of generic programming lies in the idea of concepts. A concept is a way of describing a family of related object types.*" These related object types can be integral types such as `bool`, `char`, or `int`. A concept embodies a set of requirements on related types such as their supported operations, their semantics, and their time and space complexity. For example, on average, accessing an unordered associative container's elements has constant-time complexity.

The Standard Template Library (STL) as a generic library is based on concepts. From a bird's-eye view, the STL consists of three components. Those are containers, algorithms that run on containers, and iterators that connect both of them.

Each container provides iterators that respect its structure, and the algorithms operate on these iterators. A container, such as a sequence container or an associative container, models a semi-open range. The elements of the container are accessed via iterators, as well as iterating through them, and comparing their equality. The abstraction of the STL is based on concepts such as semi-open range and iterator and allows for transparent use of the containers and algorithms of the STL.

More generally, what are the advantages of concepts?

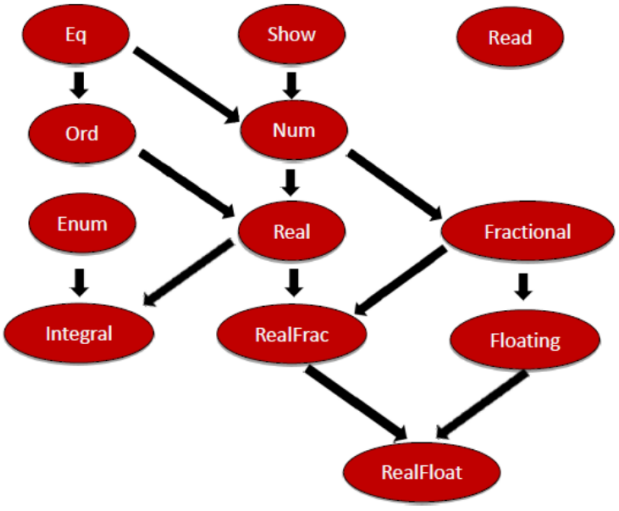
4.1.2 Advantages of Concepts

- Requirements for template parameters are part of the interface.
- Concepts are executable documentation. They document the restrictions on the generic code that the compiler verifies.
- The overloading of functions and specialization of class templates can be based on concepts.
- Concepts can be used for function templates, class templates, and generic member functions of classes or class templates, but also variable templates[6] and alias templates[7]
- You get improved error messages because the compiler compares the requirements of the template parameters with the given template arguments.
- You can use predefined concepts or define your own.
- The usage of `auto` and concepts is unified. Instead of `auto`, you can use a concept.
- If a function declaration uses a concept, it automatically becomes a function template. Writing function templates is, therefore, as easy as writing a function.

[5] https://www.fm2gp.com/
[6] https://en.cppreference.com/w/cpp/language/variable_template
[7] https://en.cppreference.com/w/cpp/language/type_alias

4.1.3 The long, long History

The first time I heard about concepts was around 2005 - 2006. They reminded me of Haskell type classes. Type classes in Haskell are interfaces for similar types. Here is a part of Haskell's[8] type classes hierarchy.



Haskell Type Classes Hierarchy

But C++ concepts are different. Here are a few observations.

- In Haskell, any type has to be an instance of a type class. In C++20, a type has to fulfill the requirements of a concept.
- Concepts can be used on non-type arguments of templates in C++. For example, numbers such as the value 5 are non-type arguments. For example, when you want to have a `std::array` of `int`s with 5 elements, you use the non-type argument 5: `std::array<int, 5> myArray`.
- Concepts add no run-time costs.

Originally, concepts were going to be the main feature of C++11, but they were removed during a standardization meeting in July 2009 in Frankfurt. The quote from Bjarne Stroustrup speaks for itself: "*The C++0x concept design evolved into a monster of complexity.*"[9]. A few years later, the next try was also not successful: concepts lite was removed from the C++17 standard. They finally became part of C++20.

4.1.4 Use of Concepts

Essentially, there are four ways to use a concept.

[8] https://en.wikipedia.org/wiki/Haskell_(programming_language)
[9] https://isocpp.org/blog/2013/02/concepts-lite-constraining-templates-with-predicates-andrew-sutton-bjarne-s

4.1.4.1 Four Ways to use a Concept

I apply the predefined concept `std::integral` in the program `conceptsIntegralVariations.cpp` in all four ways.

Four variations using the concept `std::integral`

```
// conceptsIntegralVariations.cpp

#include <concepts>
#include <iostream>

template<typename T>
requires std::integral<T>
auto gcd(T a, T b) {
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}

template<typename T>
auto gcd1(T a, T b) requires std::integral<T> {
    if( b == 0 ) return a;
    else return gcd1(b, a % b);
}

template<std::integral T>
auto gcd2(T a, T b) {
    if( b == 0 ) return a;
    else return gcd2(b, a % b);
}

auto gcd3(std::integral auto a, std::integral auto b) {
    if( b == 0 ) return a;
    else return gcd3(b, a % b);
}

int main(){

    std::cout << '\n';

    std::cout << "gcd(100, 10)= "  <<  gcd(100, 10)  << '\n';
    std::cout << "gcd1(100, 10)= " <<  gcd1(100, 10)  << '\n';
    std::cout << "gcd2(100, 10)= " <<  gcd2(100, 10)  << '\n';
    std::cout << "gcd3(100, 10)= " <<  gcd3(100, 10)  << '\n';

    std::cout << '\n';
```

```
40
41  }
```

Thanks to the header `<concepts>` in line 3, I can use the concept `std::integral`. The concept is fulfilled if `T` is integral[10]. The function name `gcd` stands for the greatest-common-divisor algorithm based on the Euclidean[11] algorithm.

Here are the four ways to use concepts:

- Requires clause (line 6)
- Trailing requires clause (line 13)
- Constrained template parameter (line 19)
- Abbreviated function template (line 25)

For simplicity reasons, each function template returns `auto`. There is a semantic difference between the function templates `gcd`, `gcd1`, `gcd2`, and the function `gcd3`. In the case of `gcd`, `gcd1`, or `gcd2`, arguments `a` and `b` must have the same type. This does not hold for the function `gcd3`. Parameters `a` and `b` can have different types but must both fulfill the concept `integral`.

```
gcd(100, 10)= 10
gcd1(100, 10)= 10
gcd2(100, 10)= 10
gcd3(100, 10)= 10
```

Use of the concept `std::integral`

The functions `gcd` and `gcd1` use requires clauses. Requires clauses are more powerful than you may think. Let me discuss more details to requires clauses.

4.1.4.2 Requires Clause

The previous program, `conceptsIntegralVariations.cpp`, exemplifies that you can use a concept to define a function or function template. Of course, there are more use cases. For completeness, I want to add that you can specify the return type of a function or a function template using concepts.

The keyword `requires` introduces a requires clause that specifies constraints on a template argument (`gcd`) or on a function declaration (`gcd1`). `requires` must be followed by a compile-time predicate, a named concept (`gcd`), or a requires expression. Of course, you can combine all of the three mentioned.

The compile-time predicate can also be an expression:

[10] https://en.cppreference.com/w/cpp/types/is_integral
[11] https://en.wikipedia.org/wiki/Euclid

Using a compile-time predicate in a requires clause

```
1  // requiresClause.cpp
2
3  #include <iostream>
4
5  template <unsigned int i>
6  requires (i <= 20)
7  int sum(int j) {
8      return i + j;
9  }
10
11
12 int main() {
13
14     std::cout << '\n';
15
16     std::cout << "sum<20>(2000): " << sum<20>(2000) << '\n',
17     // std::cout << "sum<23>(2000): " << sum<23>(2000) << '\n',  // ERROR
18
19     std::cout << '\n';
20
21 }
```

The compile-time predicate used in line 6 exemplifies an interesting point: the requirement is applied to the non-type `i`, and not on a type as usual.

```
sum<20>(2000): 2020
```

Compile-time predicates in a requires clause

When you use line 17, the clang compiler reports the following error:

```
<source>:17:39: error: no matching function for call to 'sum'
    std::cout << "sum<23>(2000): " << sum<23>(2000) << '\n',  // ERROR
                                      ^~~~~~~
<source>:7:5: note: candidate template ignored: constraints not satisfied [with i = 23]
int sum(int j) {
    ^
<source>:6:11: note: because '23U <= 20' (23 <= 20) evaluated to false
requires (i <= 20)
          ^
```

Failing compile time predicates in a requires clauses

Avoid Compile-Time Predicates in Requires Clauses

When you constrain template parameters or function templates using concepts, you should use named concepts or combinations of them. Concepts are meant to be semantic categories, but not syntactic constraints like `i <= 20`. Giving concepts a name enables their reuse.

4.1.4.3 Concepts as Return Type of a Function

Here are the definitions of the function template `gcd` and the function `gcd1` using concepts as return types.

Using a concept as return type

```
template<typename T>
requires std::integral<T>
std::integral auto gcd(T a, T b) {
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}

std::integral auto gcd1(std::integral auto a, std::integral auto b) {
    if( b == 0 )return a;
    else return gcd1(b, a % b);
}
```

4.1.4.4 Use-Cases for Concepts

First and foremost, concepts are compile-time predicates. A compile-time predicate is a function that is executed at compile time and returns a boolean. Before I dive into the various use cases of concepts, I want to demystify concepts and present them simply as functions returning a boolean at compile time.

4.1.4.4.1 Compile-Time Predicates

A concept can be used in a control structure, which is executed at run time or compile time.

Concepts as compile-time predicates

```
// compileTimePredicate.cpp

#include <compare>
#include <iostream>
#include <string>
#include <vector>

struct Test{};

int main() {

    std::cout << '\n';

    std::cout << std::boolalpha;

    std::cout << "std::three_way_comparable<int>: "
              << std::three_way_comparable<int> << "\n";

    std::cout << "std::three_way_comparable<double>: ";
    if (std::three_way_comparable<double>) std::cout << "True";
    else std::cout << "False";

    std::cout << "\n\n";

    static_assert(std::three_way_comparable<std::string>);

    std::cout << "std::three_way_comparable<Test>: ";
    if constexpr(std::three_way_comparable<Test>) std::cout << "True";
    else std::cout << "False";

    std::cout << '\n';

    std::cout << "std::three_way_comparable<std::vector<int>>: ";
    if constexpr(std::three_way_comparable<std::vector<int>>) std::cout << "True";
    else std::cout << "False";

    std::cout << '\n';

}
```

In the program above, I use the concept `std::three_way_comparable<T>`, which checks at compile time if `T` supports the six comparison operators. Being a compile-time predicate means, that `std::three_-`

`way_comparable` can be used at run time (lines 16 and 20) or at compile time. `static_assert` (line 25) and `constepr if`[12] (lines 28 and 34) are evaluated at compile time.

```
std::three_way_comparable<int>: true
std::three_way_comparable<double>: True

std::three_way_comparable<Test>: False
std::three_way_comparable<std::vector<int>>: True
```

Concepts as compile-time predicates

Test of Concepts

Using a concept in `static_assert(Concept<T>)` is essentially the test if the type `T` fulfills the concept. The following short program checks, if `int` is a regular type. A regular type behaves such as an `int`. The formal definition of `regular` is provided in the define concepts section.

Test if `int` models the concept `regular`

```
#include <concepts>

int main() {

   static_assert(std::regular<int>);  //  int is a regular type

}
```

After this short detour on concepts as compile-time predicates, let me continue this section with the various use cases of concepts. The concepts' applications are not too elaborate, and I mainly use predefined concepts, which I describe in more depth in the section predefined concepts.

4.1.4.4.2 Class Templates

The class template `MyVector` requires that its template parameter `T` be `regular`.

[12] https://en.cppreference.com/w/cpp/language/if

Using a concept in a class definition

```
// conceptsClassTemplate.cpp

#include <concepts>
#include <iostream>

template <std::regular T>
class MyVector{};

int main() {

    MyVector<int> myVec1;
    MyVector<int&> myVec2;  // ERROR because a reference is not regular

}
```

Line 12 causes a compile-time error because a reference is not `regular`. Here is the essential part of the GCC compiler message:

```
<source>:13:18: error: template constraint failure for 'template<class T>  requires  regular<T> class MyVector'
   13 |     MyVector<int&> myVec2;
```

A reference is not `regular`

4.1.4.4.3 Generic Member Functions

In this example, I add a generic `push_back` member function to the class `MyVector`. The `push_back` requires that its arguments be copyable.

Using a concept in a generic member function

```
1  // conceptMemberFunction.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  struct NotCopyable {
7      NotCopyable() = default;
8      NotCopyable(const NotCopyable&) = delete;
9  };
10
11 template <typename T>
12 struct MyVector{
13     void push_back(const T&) requires std::copyable<T> {}
14 };
```

```
15
16  int main() {
17
18      MyVector<int> myVec1;
19      myVec1.push_back(2020);
20
21      MyVector<NotCopyable> myVec2;
22      myVec2.push_back(NotCopyable());  // ERROR because not copyable
23
24  }
```

The compilation fails intentionally in line 22. Instances of `NotCopyable` are not copyable because the copy constructor is declared as deleted.

4.1.4.4.4 Variadic Templates

You can use concepts in variadic templates.

Applying concepts to variadic templates

```
1   // allAnyNone.cpp
2
3   #include <concepts>
4   #include <iostream>
5
6   template<std::integral... Args>
7   bool all(Args... args) { return (... && args); }
8
9   template<std::integral... Args>
10  bool any(Args... args) { return (... || args); }
11
12  template<std::integral... Args>
13  bool none(Args... args) { return not(... || args); }
14
15  int main(){
16
17      std::cout << std::boolalpha << '\n';
18
19      std::cout << "all(5, true, false): " << all(5, true, false) << '\n';
20
21      std::cout << "any(5, true, false): " << any(5, true, false) << '\n';
22
23      std::cout << "none(5, true, false): " << none(5, true, false) << '\n';
24
25  }
```

The definitions of the function templates above are based on fold expressions. C++11 supports variadic templates that can accept an arbitrary number of template arguments. Any number of template parameters is held by a so-called parameter pack. Additionally, with C++17, you can directly reduce a parameter pack with a binary operator. This reduction is called a fold expression[13]. In this example, the logical and `&&` (line 7), the logical or `||` (line 10), and the negation of the logical or (line 13) are applied as binary operators. Furthermore, `all`, `any`, and `none` requires from their type parameters that they have to support the concept `std::integral`.

```
all(5, true, false): false
any(5, true, false): true
none(5, true, false): false
```

Applying concepts onto a fold expression

4.1.4.4.5 Overloading

`std::advance`[14] is an algorithm of the Standard Template Library. It increments a given iterator `iter` by `n` elements. Based on the capabilities of the given iterator, a different advance strategy could be used. For example, a `std::forward_list` supports an iterator that can only advance in one direction, while a `std::list` supports a bidirectional iterator, and a `std::vector` supports a random access iterator. Consequently, for an iterator provided by a `std::forward_list` or `std::list`, a call to `std::advance(iter, n)` has to be incremented `n` times (see the structure of a `std::list`). This time complexity does not hold for a `std::random_access_iterator` provided by a `std::vector`. The number `n` can just be added to the iterator. A linear time complexity `O(n)` becomes, therefore, a constant complexity `O(1)`. To distinguish iterator types, concepts can be used. The program `conceptsOverloadingFunctionTemplates.cpp` should give you the general idea.

Overloading function templates on concepts

```
1  // conceptsOverloadingFunctionTemplates.cpp
2
3  #include <concepts>
4  #include <iostream>
5  #include <forward_list>
6  #include <list>
7  #include <vector>
8
9  template<std::forward_iterator I>
10 void advance(I& iter, int n){
11     std::cout << "forward_iterator" << '\n';
12 }
13
```

[13] https://www.modernescpp.com/index.php/fold-expressions
[14] https://en.cppreference.com/w/cpp/iterator/advance

```
14  template<std::bidirectional_iterator I>
15  void advance(I& iter, int n){
16      std::cout << "bidirectional_iterator" << '\n';
17  }
18
19  template<std::random_access_iterator I>
20  void advance(I& iter, int n){
21      std::cout << "random_access_iterator" << '\n';
22  }
23
24  int main() {
25
26      std::cout << '\n';
27
28      std::forward_list forwList{1, 2, 3};
29      std::forward_list<int>::iterator itFor = forwList.begin();
30      advance(itFor, 2);
31
32      std::list li{1, 2, 3};
33      std::list<int>::iterator itBi = li.begin();
34      advance(itBi, 2);
35
36      std::vector vec{1, 2, 3};
37      std::vector<int>::iterator itRa = vec.begin();
38      advance(itRa, 2);
39
40      std::cout << '\n';
41  }
```

The three variations of the function `advance` are overloaded on the concepts `std::forward_iterator` (line 9), `std::bidirectional_iterator` (line 14), and `std::random_access_iterator` (line 19). The compiler chooses the best-fitting overload. It means that for a `std::forward_list` (line 28) the overload based on the concept `std::forward_list`, for a `std::list` (line 32) the overload based on the concept `std::bidirectional_iterator`, and for a `std::vector` (line 36) the overload based on the concept `std::random_access_iterator` is used.

```
forward_iterator
bidirectional_iterator
random_access_iterator
```

Overloading function templates on concepts

A `std::random_access_iterator` is a `std::bidirectional_iterator`, and `std::bidirectional_iterator` is a `std::forward_iterator`.

4.1.4.4.6 Template Specialization

You can also specialize templates using concepts.

Template specialization on concepts

```
1  // conceptsSpecialization.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  template <typename T>
7  struct Vector {
8      Vector() {
9          std::cout << "Vector<T>" << '\n';
10     }
11 };
12
13 template <std::regular Reg>
14 struct Vector<Reg> {
15     Vector() {
16         std::cout << "Vector<std::regular>" << '\n';
17     }
18 };
19
20 int main() {
21
22     std::cout << '\n';
23
24     Vector<int> myVec1;
25     Vector<int&> myVec2;
26
27     std::cout << '\n';
28
29 }
```

When instantiating the class template, the compiler chooses the most specialized one. This means for the call `Vector<int> myVec` (line 24), the partial template specialization for `std::regular` (line 13) is chosen. A reference `Vector<int&> myVec2` (line 25) is not `regular`. Consequently, the primary template (line 6) is chosen.

```
Vector<std::regular>
Vector<T>
```

Partial template specialization of concepts

4.1.4.4.7 Using More than One Concept

So far, using the concepts has been easy, but most of the time more than one concept is used at a time.

Using more than one concept

```
template<typename Iter, typename Val>
    requires std::input_iterator<Iter>
          && std::equality_comparable<Value_type<Iter>, Val>
Iter find(Iter b, Iter e, Val v)
```

`find` requires for the iterator `Iter` and its comparison with `Val` that

- the Iterator has to be an input iterator;
- the Iterator's value type must be equality comparable with `Val`.

The same restriction on the iterator can also be expressed as a constrained template parameter.

Using more than one concept

```
template<std::input_iterator Iter, typename Val>
    requires std::equality_comparable<Value_type<Iter>, Val>
Iter find(Iter b, Iter e, Val v)
```

4.1.5 Constrained and Unconstrained Placeholders

First, let me tell you about an asymmetry in C++14.

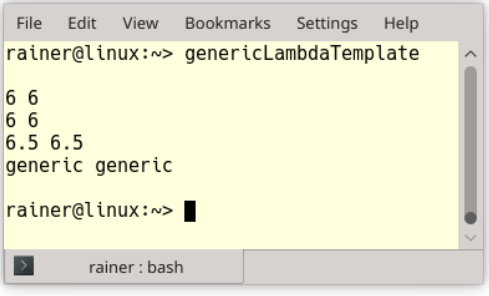
4.1.5.1 The Big Asymmetry in C++14

I often have a discussion in my classes that goes the following way. With C++14, we had generic lambdas. Generic lambdas are lambdas that use `auto` instead of a concrete type.

Comparison of a generic lambda and a function template

```
1  // genericLambdaTemplate.cpp
2
3  #include <iostream>
4  #include <string>
5
6  auto addLambda = [](auto fir, auto sec){ return fir + sec; };
7
8  template <typename T, typename T2>
9  auto addTemplate(T fir, T2 sec){ return fir + sec; }
10
11 int main(){
12
13     std::cout << std::boolalpha << '\n';
14
15     std::cout << addLambda(1, 5) << " " << addTemplate(1, 5) << '\n';
16     std::cout << addLambda(true, 5) << " " << addTemplate(true, 5) << '\n';
17     std::cout << addLambda(1, 5.5) << " " << addTemplate(1, 5.5) << '\n';
18
19     const std::string fir{"ge"};
20     const std::string sec{"neric"};
21     std::cout << addLambda(fir, sec) << " " << addTemplate(fir, sec) << '\n';
22
23     std::cout << '\n';
24
25 }
```

The generic lambda (line 6) and the function template (line 8) produce the same results.



Use of a generic lambda and a function template

Generic lambdas introduce a new way to define function templates. In my classes, I'm often asked: Can we use `auto` in functions to get function templates? Not with C++14, but you can with C++20.

In C++20, you can use unconstrained placeholders (`auto`) or constrained placeholders (concepts) in function declarations to automatically get function templates. The rule for applying is as simple as it can be. Any place where you can use an unconstrained placeholder `auto`, you can use a concept. I will explain this in detail in the section on abbreviated function templates.

4.1.5.2 Placeholders

Use of constrained placeholders instead of unconstrained placeholders

```
1  // placeholders.cpp
2
3  #include <concepts>
4  #include <iostream>
5  #include <vector>
6
7  std::integral auto getIntegral(int val){
8      return val;
9  }
10
11 int main(){
12
13     std::cout << std::boolalpha << '\n';
14
15     std::vector<int> vec{1, 2, 3, 4, 5};
16     for (std::integral auto i: vec) std::cout << i << " ";
17     std::cout << '\n';
18
19     std::integral auto b = true;
20     std::cout << b << '\n';
21
22     std::integral auto integ = getIntegral(10);
23     std::cout << integ << '\n';
24
25     auto integ1 = getIntegral(10);
26     std::cout << integ1 << '\n';
27
28     std::cout << '\n';
29
30 }
```

The concept `std::integral` can be used as a return type (line 7), in a range-based for loop (line 16), or as a type for variable `b` (line 19), or variable `integ` (line 22). To see the symmetry between `auto` and concepts, line 25 uses `auto` alone instead of `std::integral auto`, which is used on line 22. Hence, `integ1` can accept a value of any type.

```
1 2 3 4 5
true
10
10
```

Constrained placeholders instead of unconstrained placeholders in action

4.1.6 Abbreviated Function Templates

With C++20, you can use an unconstrained placeholder (`auto`) or a constrained placeholder (concept) in a function declaration including member functions and operators. This function declaration automatically becomes a function template.

Abbreviated function templates

```
1  // abbreviatedFunctionTemplates.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  template<typename T>
7  requires std::integral<T>
8  T gcd(T a, T b) {
9      if( b == 0 ) return a;
10     else return gcd(b, a % b);
11 }
12
13 template<typename T>
14 T gcd1(T a, T b) requires std::integral<T> {
15     if( b == 0 ) return a;
16     else return gcd1(b, a % b);
17 }
18
19 template<std::integral T>
20 T gcd2(T a, T b) {
21     if( b == 0 ) return a;
22     else return gcd2(b, a % b);
23 }
24
25 std::integral auto gcd3(std::integral auto a, std::integral auto b) {
26     if( b == 0 ) return a;
27     else return gcd3(b, a % b);
28 }
29
```

```
30  auto gcd4(auto a, auto b){
31      if( b == 0 ) return a;
32      return gcd4(b, a % b);
33  }
34
35  int main() {
36
37      std::cout << '\n';
38
39      std::cout << "gcd(100, 10)= "  <<  gcd(100, 10)  << '\n';
40      std::cout << "gcd1(100, 10)= " <<  gcd1(100, 10)  << '\n';
41      std::cout << "gcd2(100, 10)= " <<  gcd2(100, 10)  << '\n';
42      std::cout << "gcd3(100, 10)= " <<  gcd3(100, 10)  << '\n';
43      std::cout << "gcd4(100, 10)= " <<  gcd4(100, 10)  << '\n';
44
45      std::cout << '\n';
46
47  }
```

The definitions of the function templates `gcd` (line 6), `gcd1` (line 13), and `gcd2` (line 19) are the ones I already presented in section Four ways to use a concept. `gcd` uses a requires clause, `gcd1` a trailing requires clause and `gcd2` a constrained template parameter. Now to something new. Function template `gcd3` has the concept `std::integral` as a type parameter and thus becomes a function template with restricted type parameters. In contrast, `gcd4` is equivalent to function templates with no restriction on its type parameters. The syntax used in `gcd3` and `gcd4` to create a function template is called abbreviated function template syntax.

```
gcd(100, 10)=  10
gcd1(100, 10)= 10
gcd2(100, 10)= 10
gcd3(100, 10)= 10
gcd4(100, 10)= 10
```

Constrained

Let me stress this symmetry by demonstaiting it in another example below.

Using `auto` as a type parameter, the function `add` becomes a function template and is equivalent to the equally-named function template `add`.

The equivalent function and function template `add`

```
template<typename T, typename T2>
auto add(T fir, T2 sec) {
    return fir + sec;
}

auto add(auto fir, auto sec) {
    return fir + sec;
}
```

Accordingly, due to the usage of the concept `std::integral`, the function `sub` is equivalent to the function template `sub`.

The equivalent function and function template `sub`

```
template<std::integral T, std::integral T2>
std::integral auto sub(T fir, T2 sec) {
    return fir - sec;
}

std::integral auto sub(std::integral auto fir, std::integral auto sec) {
    return fir - sec;
}
```

The function and the function template can have arbitrary types. This means both types can be different but must be integral. For example, a call `sub(100, 10)` and also `sub(100, true)` would be valid.

Additionally, you can also explicitly specify the template parameter:

Explicit template parameters

```
add<int>(100, 10);  // equivalent to add(100, 10)
sub<int>(100, 10);  // equivalent to sub(100, 10)
```

There is one interesting feature still missing in the abbreviated function templates syntax: you can overload on `auto` or concepts.

4.1.6.1 Overloading

The following functions `overload` are overloaded on `auto`, the concept `std::integral`, and the type `long`.

Abbreviated function templates and overloading

```
1  // conceptsOverloading.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  void overload(auto t){
7      std::cout << "auto : " << t << '\n';
8  }
9
10 void overload(std::integral auto t){
11     std::cout << "Integral : " << t << '\n';
12 }
13
14 void overload(long t){
15     std::cout << "long : " << t << '\n';
16 }
17
18 int main(){
19
20     std::cout << '\n';
21
22     overload(3.14);
23     overload(2010);
24     overload(2020L);
25
26     std::cout << '\n';
27
28 }
```

The compiler chooses the overload on `auto` (line 6) with a double, the overload on the concept `std::integral` (line 10) with an `int`, and the overload on `long` (line 14) with a `long`.

```
auto : 3.14
Integral : 2010
long : 2020
```

Abbreviated function templates and overloading

What we don't get: Template Introduction

Maybe you are missing one feature in this chapter on concepts: template introduction. Template introduction was part of the technical specification on concepts, TS ISO/IEC TS 19217:2015[15], and was an experimental implementation of concepts. GCC 6[16] fully implemented the concepts TS. Aside the syntactic differences to concepts in C++20, the concepts TS supports a concise way of defining templates.

In the example below, assume that `Integral` is a concept.

Template introduction in the concepts TS

```
Integral{T}
Integral gcd(T a, T b){
   if( b == 0 ){ return a; }
   else{
       return gcd(b, a % b);
   }
}

Integral{T}
class ConstrainedClass{};
```

This small code snippet above used template introduction in two ways. First, to define a function template with a constrained template parameter; second, to define a class template with a constrained template parameter. Template introduction had one limitation. You could only use it with a constrained template parameter (concept), but not with an unconstrained template parameter (`auto`). This asymmetry could easily be overcome by defining a concept that always returns `true`:

The concept `Generic` is always fulfilled

```
template<typename T>
concept bool Generic(){
   return true;
}
```

Don't be irritated, I used in the example the concepts TS syntax to define the `Generic` concept. The C++20 syntax is slightly more concise. Read more details of the C++20 syntax in section Defining Concepts.

4.1.7 Predefined Concepts

The golden rule "Don't reinvent the wheel" also applies to concepts. The C++ Core Guidelines[17] are very clear about this rule: T.11: Whenever possible, use standard concepts. Consequently, I want to give

[15] https://www.iso.org/standard/64031.html
[16] https://en.wikipedia.org/wiki/GNU_Compiler_Collection
[17] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

you an overview of the important predefined concepts. I intentionally ignore any special or auxiliary concepts.

All predefined concepts are detailed in the latest C++ working draft. In April 2023, this is the C++23 standard draft N4928[18]. Finding them all can be quite a challenge! Most concepts are in chapter 18 (concepts library) and chapter 24 (ranges library). Additionally, a few concepts are in chapter 17 (language support library), chapter 20 (general utilities library), chapter 23 (iterators library), and chapter 26 (numerics library). The C++20 draft N4928 also has an index of all library concepts and shows how the concepts are implemented.

4.1.7.1 Language Support Library

This section discusses an interesting concept, `three_way_comparable`. It is used to support the three-way comparison operator. It is specified in the header `<compare>`.

More formally, let `a` and `b` be values of type `T`. These values are `three_way_comparable` only if:

- `(a <=> b == 0) == bool(a == b)` is `true`
- `(a <=> b != 0) == bool(a != b)` is `true`
- `((a <=> b) <=> 0)` and `(0 <=> (b <=> a))` are equal
- `(a <=> b < 0) == bool(a < b)` is `true`
- `(a <=> b > 0) == bool(a > b)` is `true`
- `(a <=> b <= 0) == bool(a <= b)` is `true`
- `(a <=> b >= 0) == bool(a >= b)` is `true`

4.1.7.2 Concepts Library

The most frequently used concepts can be found in the concepts library. They are defined in the `<concepts>` header.

4.1.7.2.1 Language-related concepts

This section has about 15 concepts that should be self-explanatory. These concepts express relationships between types, type classifications, and fundamental type properties. Their implementation is often directly based on the corresponding function from the type-traits library[19]. Where deemed necessary, I provide additional explanation.

- `std::default_initializable`: `std::default_initializable<T>` guarantees the `T` can be default constructed
- `std::same_as`
- `std::derived_from`

[18] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4928.pdf
[19] https://en.cppreference.com/w/cpp/header/type_traits

- `std::convertible_to`: `std::convertible_to<T, U>` guarantees that `T` is implicitly or explicitly convertible to `U`
- `std::common_reference_with`: `std::common_reference_with<T, U>` must be well-formed and `T` and `U` must be convertible to a reference type `C`, where `C` is the same as `std::common_reference_t<T, U>`
- `std::common_with`: similar to `std::common_reference_with`, but the common type `C` is the same as `common_type_t<T, U>` and may not be a reference type
- `std::assignable_from`
- `std::swappable`

4.1.7.2.2 Arithmetic Concepts

- `std::integral`
- `std::signed_integral`
- `std::unsigned_integral`
- `std::floating_point`

The standard's definition of the arithmetic concepts is straightforward:

```
template<class T>
concept integral = is_integral_v<T>;

template<class T>
concept signed_integral = integral<T> && is_signed_v<T>;

template<class T>
concept unsigned_integral = integral<T> && !signed_integral<T>;

template<class T>
concept floating_point = is_floating_point_v<T>;
```

4.1.7.2.3 Lifetime Concepts

- `std::destructible`
- `std::constructible_from`
- `std::default_constructible`
- `std::move_constructible`
- `std::copy_constructible`

4.1.7.2.4 Comparison Concepts

- `std::equality_comparable`
- `std::totally_ordered`

Maybe you know it from your mathematics studies: For values `a`, `b`, and `c` of type `T`, `T` models `std::totally_ordered` if and only if

- Exactly one of `bool(a < b)`, `bool(a > b)`, or `bool(a == b)` is `true`
- If `bool(a < b)` and `bool(b < c)`, then `bool(a < c)`
- `bool(a > b) == bool(b < a)`
- `bool(a <= b) == !bool(b < a)`
- `bool(a >= b) == !bool(a < b)`

4.1.7.2.5 Object Concepts

- `std::movable`
- `std::copyable`
- `std::semiregular`
- `std::regular`

Here are the concise definitions of the four concepts:

```
template<class T>
concept movable = is_object_v<T> && move_constructible<T> &&
                  assignable_from<T&, T> && swappable<T>;

template<class T>
concept copyable = copy_constructible<T> && movable<T> &&
                   assignable_from<T&, T&> &&
                   assignable_from<T&, const T&> && assignable_from<T&, const T>;

template<class T>
concept semiregular = copyable<T> && default_initializable<T>;

template<class T>
concept regular = semiregular<T> && equality_comparable<T>;
```

I have to add a few words. The concept `std::movable` requires for `T` that `std::is_object_v<T>` holds. From the definition of the type-trait `std::is_object_v<T>`, this means that `T` is either a scalar, an array, a union, or a class.

I implement the concept `semiregular` and `regular` in the section define concepts. Informally, a `semiregular` type behaves similarly to an `int`, and a `regular` type behaves similarly to an `int` and can be compared using ==.

4.1.7.2.6 Callable Concepts

- `std::invocable`
- `std::regular_invocable`: a type models `std::invocable` and equality-preserving, and does not modify the function arguments; equality-preserving means the it produces the same output when given the same input
- `std::predicate`: a type models a predicate if it models `std::invocable` and returns a boolean

4.1.7.3 General Utilities Library

This chapter in the standard has only special memory concepts; therefore I don't refer to them here.

4.1.7.4 Iterators Library

The iterators library has many important concepts. They are defined in the `<iterator>` header. Here are the iterator categories:

- `std::input_output_iterator`
- `std::input_iterator`
- `std::output_iterator`
- `std::forward_iterator`
- `std::bidirectional_iterator`
- `std::random_access_iterator`
- `std::contiguous_iterator`

The six categories of iterators correspond to the respective iterator concepts. The table below provides two interesting pieces of information. For the four most prominent iterator categories, the table shows their properties and the associated standard library containers.

Properties and Containers of each iterator category

Iterator Category	Properties	Containers

`std::forward_iterator`	`++It`, `It++` , `*It`
	`It == It2`, `It != It2`
		`std::unordered_set`
		`std::unordered_map`
		`std::unordered_multiset`
		`std::unordered_multimap`
		`std::forward_list`

`std::bidirectional_iterator`	`--It`, `It--`	`std::set`
		`std::map`
		`std::multiset`
		`std::multimap`

Properties and Containers of each iterator category

Iterator Category	Properties	Containers
		`std::list`

`std::random_access_iterator`	`It[i]`
	`It += n`, `It -= n`
	`It + n` , `It - n`
	`n + It`
	`It - It2`
	`It < It2`, `It <= It2`
	`It > It2`, `It >= It2`
		`std::deque`

`std::contiguous_iterator`	`It[i]`
	`It += n`, `It -= n`
	`It + n` , `It - n`
	`n + It`
	`It - It2`
	`It < It2`, `It <= It2`
	`It > It2`, `It >= It2`
		`std::array`
		`std::vector`
		`std::string`

A `std::input_output_iterator` support the operations `++It`, `It++`, and `*It`. `std::input_iterator` and `std::output_iterator` are `std::input_output_iterator`. The following relation holds: A contiguous iterator is a random-access iterator, a random-access iterator is a bidirectional iterator, and a bidirectional iterator is a forward iterator. A contiguous iterator requires that the elements of the container are stored contiguously in memory.

4.1.7.4.1 Algorithm Concepts

- `std::permutable`: in-place reordering of elements is possible
- `std::mergeable`: merging sorted sequences into an output sequence is possible
- `std::sortable`: permuting a sequence into an ordered sequence is possible

4.1.7.5 Ranges Library

The ranges library contains the concepts critical to the ranges and views features. They are similar to the concepts in the iterators library and are defined in the `<ranges>` header.

4.1.7.5.1 Ranges

- `std::ranges::range`: A range specifies a group of items that you can iterate over. It provides a begin iterator and an end sentinel. Of course, the containers of the STL are ranges.

The concept range

```
template<typename T>
concept range = requires(T& t) {
    ranges::begin(t);
    ranges::end(t);
};
```

There are further refinements for `std::ranges::range`.

- `std::ranges::input_range`: specifies a range whose iterator type satisfies `std::input_iterator` (e.g. can iterate from beginning to end at least once)
- `std::ranges::output_range`: specifies a range whose iterator type satisfies `std::output_iterator`
- `std::ranges::forward_range`: specifies a range whose iterator type satisfies `std::forward_iterator` (can iterate from beginning to end more than once)
- `std::ranges::bidirectional_range`: specifies a range whose iterator type satisfies `std::bidirectional_iterator` (can iterate forward and backward more than once)
- `std::ranges::random_access_range`: specifies a range whose iterator type satisfies `std::random_access_iterator` (can jump in constant time to an arbitrary element with the index operator [])
- `std::ranges::contiguous_range`: specifies a range whose iterator type satisfies `std::contiguous_iterator` (elements are stored consecutively in memory)

Each container of the Standard Template Library supports a specific range. The supported range specifies the capabilities of its iterators.

Properties and containers of each range concept

Concept	Properties	Containers

`std::ranges::input_range`	`++It`, `It++` , `*It`
	`It == It2`, `It != It2`
		`std::unordered_set`
		`std::unordered_map`
		`std::unordered_multiset`
		`std::unordered_multmap`
		`std::forward_list`

`std::ranges::bidirectional_range`	`--It`, `It--`	`std::set`
		`std::map`
		`std::multiset`
		`std::multimap`
		`std::list`

`std::ranges::random_access_range`	`It[i]` `It += n`, `It -= n`	`std::deque`

Properties and containers of each range concept

Concept	Properties	Containers
	`It + n , It - n`	
	`n + It`	
	`It - It2`	
	`It < It2, It <= It2`	
	`It > It2, It >= It2`	
`std::ranges::contiguous_range`	`It[i]`	`std::array`
	`It += n, It -= n`	`std::vector`
	`It + n , It - n`	`std::string`
	`n + It`	
	`It - It2`	
	`It < It2, It <= It2`	
	`It > It2, It >= It2`	

A container supporting the `std::ranges::contiguous_range` concept supports all previously mentioned concepts in the table such as `std::ranges::random_access_range`, `std::ranges::bidirectional_range`, and `std::ranges::input_range`. The same holds for all other ranges. A `std::ranges::input_range` is a `std::ranges::range`.

There are a few special ranges:

- `std::ranges::borrowed_range` guarantees that the iterators are not bound to the lifetime of the range.
- `std::ranges::common_range` guarantees that the begin and end iterator have the same type. All classical iterators are a `common_range`.
- `std::ranges::sized_range` guarantees that the number of elements can be computed in constant time using the difference of its begin and end iterator.
- `std::ranges::viewable_range` guarantees that the range can be converted into a view using `std::ranges::all`.

4.1.7.5.2 Views

A `std::ranges::view` is a range that has constant time copy, move, and assignment operations.

The following lines show the definition of the concept view.

The concept view

```
template<typename T>
concept view = range<T> &&
               movable<T> &&
               enable_view<T>;

template<class T>
inline constexpr bool enable_view = derived_from<T, view_base>;
```

A view is typically something that you apply on a range. and it performs some operation. A view does not own data, and the time a view takes to copy, move, or assign is constant. It should be read-only, stateless, and equality-preserving. Here is a quote from Eric Niebler's range-v3 implementation, which is the basis for the C++20 ranges: "*Views are composable adaptations of ranges where the adaptation happens lazily as the view is iterated.*"

Consequently, the containers of the STL are ranges but not views.

4.1.7.6 Numeric Library

The numeric library provides the concept of a `std::uniform_random_bit_generator` that is defined in the header `<random>`. A `std::uniform_random_bit_generator g` of type `G` must return uniformly-distributed unsigned integers. Additionally, a uniform random-bit generator `g` of type `G` has to support the member functions `G::min` and `G::max`.

4.1.8 Define Concepts

When the concept you are looking for is not one of the predefined concepts in C++20, you must define your concept. In this section, I will define a few concepts that will be distinguishable from the predefined concepts through the use of CamelCase syntax. Consequently, my concept for a signed integral is named `SignedIntegral`, whereas the C++ standard concept goes by the name `signed_integral`.

The syntax to define a concept is straightforward:

Concept definition

```
template <template-parameter-list>
concept concept-name = constraint-expression;
```

A concept definition starts with the keyword `template` and has a template parameter list. The second line is more interesting. It uses the keyword `concept` followed by the concept name and the constraint expression.

A `constraint-expression` is a compile-time predicate that can either be:

- A logical combination of other concepts or compile-time predicates
  - Logical combination can be built out of conjunctions (`&&`), disjunctions (`||`), or negations (`!`)
  - Compile-time predicates are callables that return a boolean value at compile time
- A requires expression
  - Simple requirements
  - Type requirements
  - Compound requirements
  - Nested requirements

In the next two sections, I will demonstrate various ways of defining concepts.

4.1.8.1 A Logical Combination of other Concepts and Compile-Time Predicates

You can combine concepts and compile-time predicates using conjunctions (`&&`) and disjunctions (`||`). You can negate components using the exclamation mark (`!`). Evaluation of this logical combination of concepts and compile-time predicates obeys short-circuit evaluation[20]. Short circuit evaluation means that the evaluation of a logical expression automatically stops when its overall result is already determined.

Thanks to the many compile-time predicates of the type-traits library[21], you have all tools required to build powerful concepts at your disposal.

[20] https://en.wikipedia.org/wiki/Short-circuit_evaluation
[21] https://en.cppreference.com/w/cpp/header/type_traits

Don't define Concepts Recursively or try to Constrain them

A recursive definition of a concept is not valid:

Recursively defining a concept

```
template<typename T>
concept Recursive = Recursive<T*>;
```

The GCC compiler complains in this case that `'Recursive' was not declared in this scope`.

When you try to constrain a concept such as in the following code snippet, the GCC compiler unambiguously complains that `a concept cannot be constrained`.

Constraining a concept

```
template<typename T>
concept AlwaysTrue = true;

template<typename T>
requires AlwaysTrue<T>
concept Error = true;
```

Let's start with the concepts `Integral`, `SignedIntegral`, and `UnsignedIntegral`.

The concepts `Integral`, `SignedIntegral`, and `UnsignedIntegral`

```
1 template <typename T>
2 concept Integral = std::is_integral<T>::value;
3
4 template <typename T>
5 concept SignedIntegral = Integral<T> && std::is_signed<T>::value;
6
7 template <typename T>
8 concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

I used the type-traits function `std::is_integral`[22] to define the concept `Integral` (line 2). Thanks to the function `std::is_signed`, I refine the concepts `Integral` to the concept `SignedIntegral` (line 4). Finally, negating the concept `SignedIntegral` gives me the concept `UnsignedIntegral` (line 7).

Okay, let's try it out.

[22] https://en.cppreference.com/w/cpp/types/is_integral

Use of the concepts `Integral`, `SignedIntegral`, and `UnsignedIntegral`

```
1  // SignedUnsignedIntegrals.cpp
2
3  #include <iostream>
4  #include <type_traits>
5
6  template <typename T>
7  concept Integral = std::is_integral<T>::value;
8
9  template <typename T>
10 concept SignedIntegral = Integral<T> && std::is_signed<T>::value;
11
12 template <typename T>
13 concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
14
15 void func(SignedIntegral auto integ) {
16     std::cout << "SignedIntegral: " << integ << '\n';
17 }
18
19 void func(UnsignedIntegral auto integ) {
20     std::cout << "UnsignedIntegral: " << integ << '\n';
21 }
22
23 int main() {
24
25     std::cout << '\n';
26
27     func(-5);
28     func(5u);
29
30     std::cout << '\n';
31
32 }
```

I use the abbreviated function-template syntax to overload the function `func` on the concept `SignedIntegral` (line 15) and `UnsignedIntegral` (line 19). The compiler chooses the expected overload:

```
SignedIntegral: -5
UnsignedIntegral: 5
```

Use of the concepts `SignedIntegral`, and `UnsignedIntegral`

For completeness reasons, the following concept `Arithmetic` uses disjunction.

The concept `Arithmetic`

```
template <typename T>
concept Arithmetic = std::is_integral<T>::value || std::is_floating_point<T>::value;
```

4.1.8.2 Requires Expressions

Thanks to `requires` expressions, you can define powerful concepts. A `requires` expression has the following form:

Requires expression

```
requires (parameter-list(optional)) {requirement-seq}
```

- `parameter-list`: A comma-separated list of parameters, such as in a function declaration
- `requirement-seq`: A sequence of requirements consisting of simple, type, compound, or nested requirements

Requires expressions can also be used as a standalone feature when a compile-time predicate is required. Read more about this feature in the secion require expression.

4.1.8.2.1 Simple Requirements

The following concept `Addable` is a simple requirement:

The concept `Addable`

```
template<typename T>
concept Addable = requires (T a, T b) {
    a + b;
};
```

The concept `Addable` requires that the addition `a + b` of two values of the same type `T` is possible.

4.1.8.2.2 Type Requirements

In a type requirement, you have to use the keyword `typename` together with a type name.

The concept `TypeRequirement`

```
template<typename T>
concept TypeRequirement = requires {
    typename T::value_type;
    typename Other<T>;
};
```

The concept `TypeRequirement` requires that type `T` has a nested member `value_type` and that the class template `Other` can be instantiated with `T`.

Let's try this out:

Use of the concepts `TypeRequirement`

```
1  #include <iostream>
2  #include <vector>
3
4  template <typename>
5  struct Other;
6
7  template <>
8  struct Other<std::vector<int>> {};
9
10 template<typename T>
11 concept TypeRequirement = requires {
12     typename T::value_type;
13     typename Other<T>;
14 };
15
16 int main() {
17
18     TypeRequirement auto myVec= std::vector<int>{1, 2, 3};
19
20 }
```

The expression `TypeRequirement auto myVec = std::vector<int>{1, 2, 3}` (line 18) is valid. A `std::vector`[23] has an inner member `value_type` (line 12) and the class template `Other` can be instantiated with `std::vector<int>` (line 13).

4.1.8.2.3 Compound Requirements

A compound requirement has the form

[23] https://en.cppreference.com/w/cpp/container/vector

Compound requirement

```
{expression} noexcept(optional) return-type-requirement(optional);
```

In addition to a simple requirement, a compound requirement can have a `noexcept` specifier[24] and a requirement on its return type.

The concept `Equal`, demonstrated in the following example, uses compound requirements.

Definition and use of the concept `Equal`

```
1  // conceptsDefinitionEqual.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  template<typename T>
7  concept Equal = requires(T a, T b) {
8      { a == b } -> std::convertible_to<bool>;
9      { a != b } -> std::convertible_to<bool>;
10 };
11
12 bool areEqual(Equal auto a, Equal auto b){
13   return a == b;
14 }
15
16 struct WithoutEqual{
17   bool operator==(const WithoutEqual& other) = delete;
18 };
19
20 struct WithoutUnequal{
21   bool operator!=(const WithoutUnequal& other) = delete;
22 };
23
24 int main() {
25
26     std::cout << std::boolalpha << '\n';
27     std::cout << "areEqual(1, 5): " << areEqual(1, 5) << '\n';
28
29     /*
30
31     bool res = areEqual(WithoutEqual(),  WithoutEqual());
32     bool res2 = areEqual(WithoutUnequal(),  WithoutUnequal());
33
```

[24] https://en.cppreference.com/w/cpp/language/noexcept_spec

```
34      */
35
36      std::cout << '\n';
37
38  }
```

The concept `Equal` (line 6) requires that its type parameter `T` supports the equal and not-equal operators. Additionally, both operators have to return a value that is convertible to a boolean. Of course, `int` supports the concept `Equal`, but this does not hold for the types `WithoutEqual` (line 16) and `WithoutUnequal` (line 20). Consequently, when I use the type `WithoutEqual` (line 31), I get the following error message when using the GCC compiler.

```
<source>:6:17:   in requirements with 'T a', 'T b' [with T = WithoutEqual]
<source>:7:9: note: the required expression '(a == b)' is invalid
    7 |     { a == b } -> std::convertible_to<bool>;
      |       ~~^~~~
<source>:8:9: note: the required expression '(a != b)' is invalid
    8 |     { a != b } -> std::convertible_to<bool>;
      |       ~~^~~~
```

`WithoutEqual` does not fulfill the concept `Equal`

4.1.8.2.4 Nested Requirements

A nested requirement has the form

Nested requirement

```
requires constraint-expression;
```

Nested requirements are used to specify requirements on type parameters.

Here is another way to define the concept `UnsignedIntegral` (see logical combinations of concepts and predicates):

The concepts `Integral`, `SignedIntegral`, and `UnsignedIntegral`

```
1  // nestedRequirements.cpp
2
3  #include <type_traits>
4
5  template <typename T>
6  concept Integral = std::is_integral<T>::value;
7
8  template <typename T>
9  concept SignedIntegral = Integral<T> && std::is_signed<T>::value;
```

```
10
11  // template <typename T>
12  // concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
13
14  template <typename T>
15  concept UnsignedIntegral = Integral<T> &&
16  requires(T) {
17      requires !SignedIntegral<T>;
18  };
19
20  int main() {
21
22      UnsignedIntegral auto n = 5u;  // works
23      // UnsignedIntegral auto m = 5;   // compile time error, 5 is a signed literal
24
25  }
```

Line 14 uses the concept `SignedIntegral` as a nested requirement to refine the concept `Integral`. Honestly, the commented-out concept `UnsignedIntegral` in line 11 is more convenient to read.

The concept `Ordering` in the following section demonstrates the use of nested requirements.

4.1.9 Requires Expressions

Requires expressions can also be used as a standalone feature when a compile-time predicate is required. Therefore, use cases for requires expression can be a [`static_assert`], *constexpr if*[25], or a requires clause,

4.1.9.1 `static_assert`

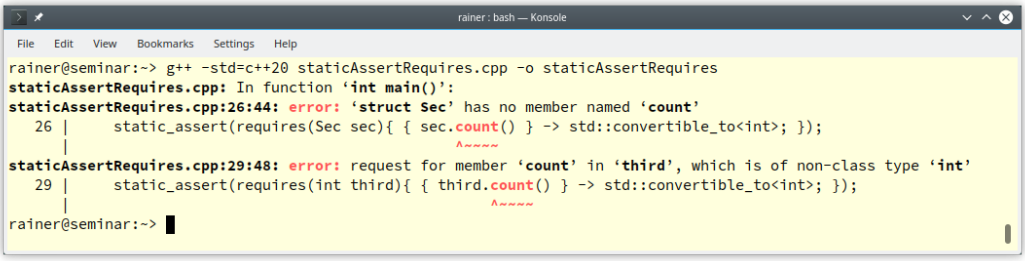
`static_assert` requires a compile-time predicate and a message displayed when the compile-time predicate fails. With C++17, the message is optional. With C++20, this compile-predicate can be a requires expression.

[25] https://en.cppreference.com/w/cpp/language/if

Requires expressions as predicates for `static_assert`

```
1  // staticAssertRequires.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  struct Fir {
7      int count() const {
8          return 2020;
9      }
10 };
11
12 struct Sec {
13     int size() const {
14         return 2021;
15     }
16 };
17
18 int main() {
19
20     std::cout << '\n';
21
22     First first;
23     static_assert(requires(Fir fir){ { fir.count() } -> std::convertible_to<int>; });
24
25     Second second;
26     static_assert(requires(Sec sec){ { sec.count() } -> std::convertible_to<int>; });
27
28     int third;
29     static_assert(requires(int third){ { third.count() } -> std::convertible_to<int>; });
30
31     std::cout << '\n';
32
33 }
```

The requires expressions (lines 23, 26, and 29) check if the object has a member function `count` and its result is convertible to `int`. This check is only valid for the class `First` (lines 6 - 10). On the contrary, the checks in lines 26 and 29 fail.



Requires expressions as predicates for `static_assert`

Maybe, you want to compile code depending on a compile-time check. In this case, *constexpr if*, combined with requires expressions provides you with the necessary tool.

4.1.9.2 *constexpr if*

constexpr if in C++17 allows it to compile source code conditionally. For the condition, the requires expression comes into play. All branches of the if statement have to be valid.

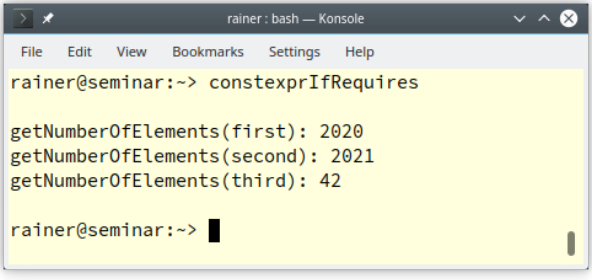
Thanks to *constexpr if*, you can define functions that inspect their arguments at compile time and generated different functionality based on their analysis.

Requires expressions as predicates for *constexpr if*

```
1  // constexprIfRequires.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  struct First {
7      int count() const {
8          return 2020;
9      }
10 };
11
12 struct Second {
13     int size() const {
14         return 2021;
15     }
16 };
17
18 template <typename T>
19 int getNumberOfElements(T t) {
20
21     if constexpr (requires(T t){ { t.count() } -> std::convertible_to<int>; }) {
22         return t.count();
```

```
23      }
24      if constexpr (requires(T t){ { t.size() } -> std::convertible_to<int>; }) {
25          return t.size();
26      }
27      else return 42;
28
29  }
30
31  int main() {
32      std::cout << '\n';
33
34      First first;
35      std::cout << "getNumberOfElements(first): "  << getNumberOfElements(first) << '\n';
36
37      Second second;
38      std::cout << "getNumberOfElements(second): "  << getNumberOfElements(second) << '\n';
39
40      int i;
41      std::cout << "getNumberOfElements(i): " << getNumberOfElements(i) << '\n';
42
43      std::cout << '\n';
44
45  }
```

Lines 21 and 24 are crucial in this code example. In line 21, the requires expressions determine if the variable `t` has a member function `count` that returns an `int`. Accordingly, line 24 determines if the variable `t` has a member function `size`. The else statement in line 27 is applied as a fallback.



Requires expressions as predicates for *constexpr if*

4.1.9.3 `requires requires` or Anonymous Concepts

You can define an anonymous concept and directly use it. In general, you should not do it. Anonymous concepts make your code hard to read, and you cannot reuse your concepts.

An anonymous concept for adding two concepts

```
template<typename T>
    requires requires (T x) { x + x; }
T add1(T a, T b) { return a + b; }
```

The function template defines its concept ad-hoc. `add1` uses a requires expression inside a requires clause. The anonymous concept is equivalent to the previously defined concept `Addable` and so is the following function template `add2` using the named concept `Addable`.

Use of the concept `Addable`

```
template<Addable T>
T add2(T a, T b) { return a + b; }
```

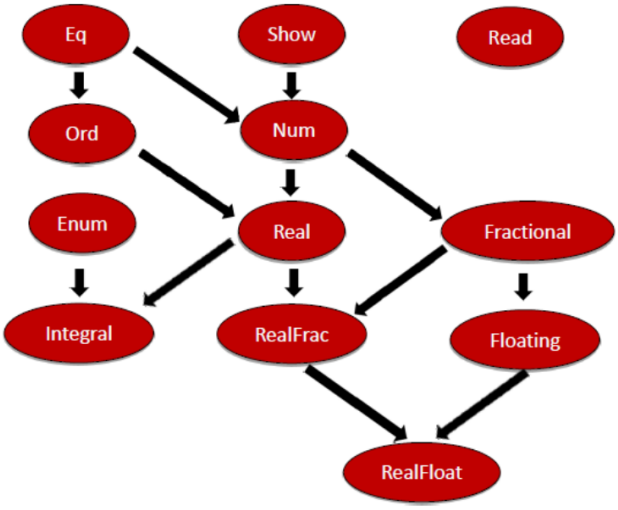
Concepts should encapsulate general ideas and give them a self-explanatory name for reuse. They are invaluable for maintaining code. Anonymous concepts read more like syntactic constraints on template parameters and should, therefore, be avoided.

4.1.10 User-Defined Concepts

In the previous sections I answered two essential questions about concepts: "How can a concept be used?" and "How can you define your concepts?". In this section, I want to apply the theoretical knowledge provided in those sections to define more advanced concepts such as `Ordering`, `SemiRegular`, and `Regular`.

4.1.10.1 The Concepts `Equal` and `Ordering`

I presented already in the short detour to the long, long history of concepts a part of Haskell's type classes hierarchy:



Haskell Type Classes Hierarchy

The class hierarchy shows that the type class `Ord` is a refinement of the type class `Eq`. Haskell expresses this elegantly.

A part of Haskell's type classes hierarchy

```
1  class Eq a where
2      (==) :: a -> a -> Bool
3      (/=) :: a -> a -> Bool
4
5  class Eq a => Ord a where
6      compare :: a -> a -> Ordering
7      (<) :: a -> a -> Bool
8      (<=) :: a -> a -> Bool
9      (>) :: a -> a -> Bool
10     (>=) :: a -> a -> Bool
11     max :: a -> a -> a
```

Each type `a` supporting the type class `Eq` (line 1), has to support equality (line 2) and inequality (line 3). Now to the interesting part of this definition. Each type `a` supporting the type class `Ord` has to support the type class `Eq` (`class Eq a => Ord a` in line 5). Additionally, type `a` has to support the four comparison operators and the functions `compare` and `max` (lines 6 - 11).

Here is my challenge. Can we express Haskell's relationship between the type classes `Eq` and `Ord` with concepts in C++20? For simplicity, I ignore Haskell's functions `compare` and `max`.

4.1.10.1.1 The Concept `Ordering`

Thanks to the `requires` expression, the definition of the concept `Ordering` looks quite similar to the definition of the type class `ord` in Haskell.

The concept `Ordering`

```
template <typename T>
concept Ordering =
    Equal<T> &&
    requires(T a, T b) {
        { a <= b } -> std::convertible_to<bool>;
        { a < b } -> std::convertible_to<bool>;
        { a > b } -> std::convertible_to<bool>;
        { a >= b } -> std::convertible_to<bool>;
    };
```

The `Ordering` concept uses nested requirements under the hood. A type `T` supports the concept `Ordering` if it supports the concept `Equal` and, additionally, the four comparison operators. Let's try it out.

Definition and usage of the concept `Ordering`

```
1  // conceptsDefinitionOrdering.cpp
2
3  #include <concepts>
4  #include <iostream>
5  #include <unordered_set>
6
7  template<typename T>
8  concept Equal =
9      requires(T a, T b) {
10         { a == b } -> std::convertible_to<bool>;
11         { a != b } -> std::convertible_to<bool>;
12     };
13
14
15 template <typename T>
16 concept Ordering =
17     Equal<T> &&
18     requires(T a, T b) {
19         { a <= b } -> std::convertible_to<bool>;
20         { a < b } -> std::convertible_to<bool>;
21         { a > b } -> std::convertible_to<bool>;
22         { a >= b } -> std::convertible_to<bool>;
23     };
```

```
24
25  template <Equal T>
26  bool areEqual(const T& a, const T& b) {
27      return a == b;
28  }
29
30  template <Ordering T>
31  T getSmaller(const T& a, const T& b) {
32      return (a < b) ? a : b;
33  }
34
35  int main() {
36
37      std::cout << std::boolalpha << '\n';
38
39      std::cout << "areEqual(1, 5): " << areEqual(1, 5) << '\n';
40
41      std::cout << "getSmaller(1, 5): " << getSmaller(1, 5) << '\n';
42
43      std::unordered_set<int> firSet{1, 2, 3, 4, 5};
44      std::unordered_set<int> secSet{5, 4, 3, 2, 1};
45
46      std::cout << "areEqual(firSet, secSet): " << areEqual(firSet, secSet) << '\n';
47
48      // auto smallerSet = getSmaller(firSet, secSet);
49
50      std::cout << '\n';
51
52  }
```

The function template `areEqual` (line 25) requires that both arguments `a` and `b` have the same type and support the concept `Equal`. Additionally, the function template `getSmaller` (line 30) requires that both arguments support the concept `Ordering`. Of course, integrals such as `1` and `5` support both concepts. A `std::unordered_set`[26], as its name implies, does not fulfill the concept `Ordering`. Consequently, I commented out line 48.

```
areEqual(1, 5): false
getSmaller(1, 5): 1
areEqual(firSet, secSet): true
```

Use of the concept `Ordering`

[26] https://en.cppreference.com/w/cpp/container/unordered_set

Let's look at the more interesting case now. What happens, when we compile line 48: `auto smallerSet = getSmaller(firSet, secSet);`? The GCC compiler complains unambiguously that a `std::unordered_set` is not a valid argument for the function template `getSmaller`.

```
<source>:48:48:   required from here
<source>:16:9:   required for the satisfaction of 'Ordering<T>' [with T = std::unordered_set<int, std::hash<int>, std::equal_to<int>, std::allocator<int> >]
<source>:18:5:   in requirements with 'T a', 'T b' [with T = std::unordered_set<int, std::hash<int>, std::equal_to<int>, std::allocator<int> >]
<source>:19:13: note: the required expression '(a <= b)' is invalid
   19 |         { a <= b } -> std::convertible_to<bool>;
      |           ~~^~~~
<source>:20:13: note: the required expression '(a < b)' is invalid
   20 |         { a < b } -> std::convertible_to<bool>;
      |           ~~^~~
<source>:21:13: note: the required expression '(a > b)' is invalid
   21 |         { a > b } -> std::convertible_to<bool>;
      |           ~~^~~
<source>:22:13: note: the required expression '(a >= b)' is invalid
   22 |         { a >= b } -> std::convertible_to<bool>;
      |           ~~^~~~
```

Erroneous usage of the function template `getSmaller`

The `Ordering` concept is already part of the C++20 standard.

- `std::three_way_comparable`: is equivalent to the concept `Ordering` presented above
- `std::three_way_comparable_with`: allows the comparison of values of different types; e.g.: `1.0 < 1.0f`

With C++20, we get the three-way comparison operator, also known as the spaceship operator `<=>`. I present it in full depth in the equality operator and three-way comparison chapter.

4.1.10.2 The Concepts `SemiRegular` and `Regular`

When you want to define a concrete type that works well in the C++ ecosystem, you should define a type that "behaves like an `int`". Formally, your concrete type should be a `regular` type. In this section, I define the concepts `SemiRegular` and `Regular`.

`SemiRegular` and `Regular` are essential ideas in C++. Sorry, I should say concepts. For example, here is rule T.46 from the C++ Core Guidelines: T.46: Require template arguments to be at least `Regular` or `SemiRegular`[27]. Now, only one important question remains to answer: What are `Regular` or `SemiRegular` types? Before I dive into the details, this is the informal answer:

- A `regular` type "behaves like an `int`." It can be copied and the result of the copy operation is independent of the original one and has the same value.

Okay, let me be more formal. A `regular` type is also a `semiregular` type, so let's begin.

[27] http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-regular

`Regular` Types

Alexander Stepanov[28], the designer of the Standard Template Library, defined the terms `regular` type and `semiregular` type. A type, according to him, is `regular` if it supports these functions:

- Copy construction
- Assignment
- Equality
- Destruction
- Total ordering

Copy construction implies default construction and Equality implies Inequality. When Stepanov defined the requirements above, move semantics was not present in C++. The book Elements of Programming[29], which Alexander Stepanov wrote together with Paul McJones[30], is devoted to `regular` types.

4.1.10.2.1 The Concept `SemiRegular`

A `semiregular` type `X` must support the Big Six and be swappable. The Big Six consists of the following functions:

- Default constructor: `X()`
- Copy constructor: `X(const X&)`
- Copy assignment: `X& operator = (const X&)`
- Move constructor: `X(X&&)`
- Move assignment: `X& operator = (X&&)`
- Destructor: `~X()`

Additionally, `X` has to be swappable: `swap(X&, X&)`

Thanks to the type-traits library[31], defining the corresponding concept is a no-brainer. First, I define the type trait `isSemiRegular` and then use it to define the concept `SemiRegular`.

[28] https://en.wikipedia.org/wiki/Alexander_Stepanov
[29] http://elementsofprogramming.com/
[30] https://www.mcjones.org/paul/
[31] https://en.cppreference.com/w/cpp/header/type_traits

```
1  template<typename T>
2  struct isSemiRegular: std::integral_constant<bool,
3                                      std::is_default_constructible<T>::value &&
4                                      std::is_copy_constructible<T>::value &&
5                                      std::is_copy_assignable<T>::value &&
6                                      std::is_move_constructible<T>::value &&
7                                      std::is_move_assignable<T>::value &&
8                                      std::is_destructible<T>::value &&
9                                      std::is_swappable<T>::value >{};
10
11
12 template<typename T>
13 concept SemiRegular = isSemiRegular<T>::value;
```

The type trait `isSemiRegular` (line 1) is fulfilled when all type traits to the Big Six (lines 3 - 8) and the type trait `std::is_swappable` (line 9) are fulfilled. The remaining step to define the concept `SemiRegular` is to use the type traits `isSemiRegular` (line 13).

Let's continue with the concept `Regular`.

4.1.10.2.2 The Concept `Regular`

There is only one step and we are ready defining the concept `Regular`. In addition to the requirements of the concept `SemiRegular`, the concept `Regular` requires that the type is equally comparable. I already defined the `Equal` concept in the section on requires expressions. Consequently, you are already done. You only have to conjunct the concepts `Equal` and `SemiRegular`.

Definition of the concept `Regular`

```
template<typename T>
concept Regular = Equal<T> &&
                  SemiRegular<T>;
```

Now, I'm curious. How can we define the corresponding concepts `std::semiregular` and `std::regular` in C++20?

4.1.10.2.3 `std::semiregular` and `std::regular`

C++20 combines the concepts `std::semiregular` and `std::regular` using of existing type traits and concepts.

Definition of the concept std::semiregular and std::regular

```
template<class T>
concept movable = is_object_v<T> && move_constructible<T> &&
                  assignable_from<T&, T> && swappable<T>;

template<class T>
concept copyable = copy_constructible<T> && movable<T> &&
                   assignable_from<T&, T&> &&
                   assignable_from<T&, const T&> && assignable_from<T&, const T>;

template<class T>
concept semiregular = copyable<T> && default_initializable<T>;

template<class T>
concept regular = semiregular<T> && equality_comparable<T>;
```

Interestingly, the `std::regular` concept is defined similarly to concept `Regular`. On the other hand, the `std::semiregular` concept is combined with more elementary concepts, such as `std::copyable` and `std::moveable`. The concept `std::movable` is based on the type-traits function `std::is_object`[32]. cppreference.com also provides a possible implementation of the compile-time predicate.

A possible implementation of the type trait std::is_object

```
template< class T>
struct is_object : std::integral_constant<bool,
                     std::is_scalar<T>::value ||
                     std::is_array<T>::value  ||
                     std::is_union<T>::value  ||
                     std::is_class<T>::value> {};
```

A type is an object if it is either a scalar, an array, a union, or a class.

To conclude this section, I want to apply the user-defined concept `Regular` and the C++20 concept `std::regular`. The program `regularSemiRegular.cpp` does this job.

[32] https://en.cppreference.com/w/cpp/types/is_object

Application of the concepts `Regular` and `SemiRegular`

```
// regularSemiRegular.cpp

#include <concepts>
#include <vector>
#include <type_traits>

template<typename T>
struct isSemiRegular: std::integral_constant<bool,
                                    std::is_default_constructible<T>::value &&
                                    std::is_copy_constructible<T>::value &&
                                    std::is_copy_assignable<T>::value &&
                                    std::is_move_constructible<T>::value &&
                                    std::is_move_assignable<T>::value &&
                                    std::is_destructible<T>::value &&
                                    std::is_swappable<T>::value >{};

template<typename T>
concept SemiRegular = isSemiRegular<T>::value;

template<typename T>
concept Equal =
    requires(T a, T b) {
        { a == b } -> std::convertible_to<bool>;
        { a != b } -> std::convertible_to<bool>;
};

template<typename T>
concept Regular = Equal<T> &&
                  SemiRegular<T>;

template <Regular T>
void behavesLikeAnInt(T) {
    // ...
}

template <std::regular T>
void behavesLikeAnInt2(T) {
    // ...
}

struct EqualityComparable { };
bool operator == (EqualityComparable const&,
                  EqualityComparable const&) {
    return true;
```

```
45  }
46
47  struct NotEqualityComparable { };
48
49  int main() {
50
51      int myInt{};
52      behavesLikeAnInt(myInt);
53      behavesLikeAnInt2(myInt);
54
55      std::vector<int> myVec{};
56      behavesLikeAnInt(myVec);
57      behavesLikeAnInt2(myVec);
58
59      EqualityComparable equComp;
60      behavesLikeAnInt(equComp);
61      behavesLikeAnInt2(equComp);
62
63      NotEqualityComparable notEquComp;
64      behavesLikeAnInt(notEquComp);
65      behavesLikeAnInt2(notEquComp);
66
67  }
```

I put all pieces from the previous code-snippets together to define the concept `Regular` (line 27). The function templates `behavesLikeAnInt` (line 31) and `behavesLikeAnInt2` (line 36) check if the arguments "behave like an int." This means the user-defined concept `Regular` and the C++20 concept `std::regular` are used to establish the condition. As the name suggests, the type `EqualityComparable` (line 41) supports equality, but the type `NotEqualityComparable` (line 47) does not. The use of the type `NotEqualityComparable` in both function calls (lines 64 and 65) is the most interesting part of the program.

Although I'm in the early stage of concepts implementation, I want to compare the error messages of a new GCC and MSVC compilers.

- GCC

I used the current GCC 10.2 with the command line argument `-std=c++20` on Compiler Explorer[33]. These are essentially the error messages when I use the user-defined concept `Regular` (line 64):

[33] https://godbolt.org/

```
<source>:23:13: note: the required expression '(a == b)' is invalid
   23 |         { a == b } -> std::convertible_to<bool>;
      |           ~~^~~~
<source>:24:13: note: the required expression '(a != b)' is invalid
   24 |         { a != b } -> std::convertible_to<bool>;
      |           ~~^~~~
```

Error message when using the concept `Regular`

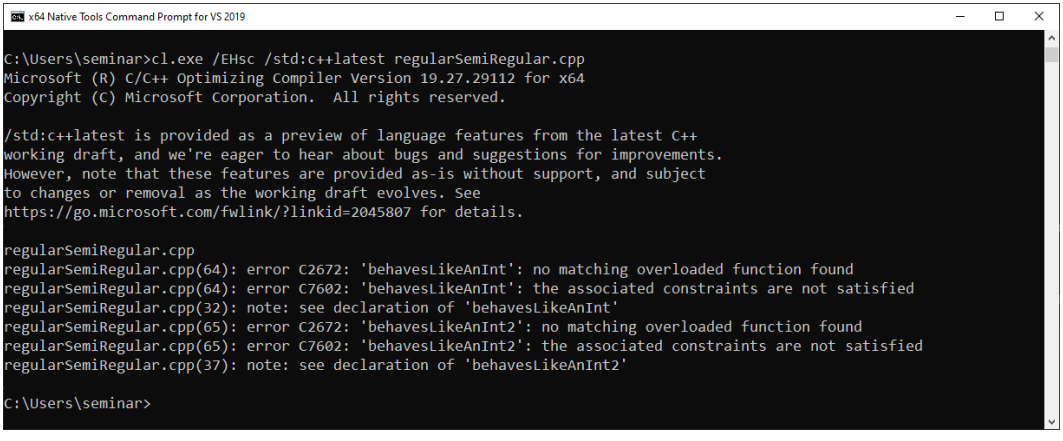
The C++20 concept `std::regular` is more comprehensive. Consequently, the call in line 65 gives a more comprehensive error message:

```
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:282:10: note: the required expression '(__t == __u)' is invalid
  282 |     { __t == __u } -> __boolean_testable;
      |       ~~~~^~~~~~
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:283:10: note: the required expression '(__t != __u)' is invalid
  283 |     { __t != __u } -> __boolean_testable;
      |       ~~~~^~~~~~
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:284:10: note: the required expression '(__u == __t)' is invalid
  284 |     { __u == __t } -> __boolean_testable;
      |       ~~~~^~~~~~
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:285:10: note: the required expression '(__u != __t)' is invalid
  285 |     { __u != __t } -> __boolean_testable;
      |       ~~~~^~~~~~
```

Error message when using the concept std::regular

- **MSVC**

The error message given by the MSVC compiler is too unspecific.



Error message when using the concepts `Regular` and `std::regular`

As you can see from the screenshot, I applied version 19.27.29112 for x64 with the command line `/EHSC /std:c++latest`.

Concepts in C++20: An Evolution or a Revolution?

This small detour expresses my opinion. First, I present the facts, then I draw my conclusion. The facts are based on what has been presented in this chapter. So which arguments speak for evolution or revolution?

Evolution

- Concepts promote working with generic code at a **higher level of abstraction**.
- Concepts give you **understandable error messages** when compiling a template fails. They provide nothing you could not achieve with the type-traits library[34], SFINAE[35], and `static_assert`[36].
- `auto` is a kind of unconstrained placeholder. With C++20, we can use concepts as **constrained placeholders**.
- With C++14, we could use **generic lambdas** as a convenient way to define function templates.

Revolution

- Concepts allow us to **verify template requirements** for the first time. Of course, you can also achieve the verification of template parameters with a combination of type-traits library[37], SFINAE[38], and `static_assert`[39], but this technique is way too advanced to regard it as a general solution.
- Thanks to the abbreviated function-templates syntax, defining templates has been radically improved.
- Concepts represent **semantic categories**, but not syntactic constraints. Instead of a concept such as `Addable`, which requires that a type supports the `+` operator, we should think in terms of a concept `Number`, where `Number` is a semantic category such as `Equal` or `Ordering`.

My Conclusion

There are many arguments whether concepts are an evolutionary step or a revolutionary jump. Mainly because of the semantic categories, I'm on the revolution side. Concepts such as `Number`, `Equality`, or `Ordering` remind me of Plato's[40] world of ideas. *It is revolutionary that we can now reason about programming in such categories.*

[34] https://en.cppreference.com/w/cpp/header/type_traits
[35] https://en.cppreference.com/w/cpp/language/sfinae
[36] https://en.cppreference.com/w/cpp/language/static_assert
[37] https://en.cppreference.com/w/cpp/header/type_traits
[38] https://en.cppreference.com/w/cpp/language/sfinae
[39] https://en.cppreference.com/w/cpp/language/static_assert
[40] https://en.wikipedia.org/wiki/Plato

Distilled Information

- Functions or classes defined on a specific type or a type parameter have their set of problems. Concepts overcome these problems by putting semantic constraints on type parameters.
- Concepts can be applied in requires clauses, in trailing requires clauses, as constrained template parameters, or in the abbreviated function templates.
- Concepts are compile-time predicates that can be used for all kinds of templates. You can overload on concepts, specialize templates on concepts, use concepts for member functions or variadic templates.
- Thanks to C++20 and concepts, the use of unconstrained placeholders (`auto`) and constrained placeholders (concepts) is unified. Whenever you use `auto`, you can use concepts in C++20.
- Thanks to the new abbreviated function-templates syntax, defining a function template has become a piece of cake.
- Don't reinvent the wheel. Before you define your concepts, study the rich set of predefined concepts in the C++20 standard. When you define your concepts, you can apply two techniques: combine concepts and compile-time predicates or use requires expressions.
- Requires expressions can be used as a compile-time predicate in `static_assert`, or *constexpr if*.

4.2 Modules

Cippi prepares the packages

Modules are one of the four big features of C++20: concepts, modules, ranges, and coroutines. Modules promise much: shorter compile times, macro isolation, abolishing header files, and avoiding ugly workarounds. Before I dive into mdules, I want to provide a first example.

4.2.1 A First Example

Let's start with a simple `math` module.

A simple math module

```
// math.ixx

export module math;

export int add(int fir, int sec){
    return fir + sec;
}
```

The expression `export module math` is the module declaration. By putting `export` before the function `add`'s declaration, `add` is exported and can, therefore, be used by a module consumer.

Use of the simple math module

```
// client.cpp

import math;

int main() {

   add(2000, 20);

}
```

`import math` imports module `math` and makes the exported names in the client visible.

Let me start with the module declaration file.

4.2.1.1 Module Declaration File

Did you notice the strange name of the module: `math.ixx`.

- The **Microsoft compiler** uses the extension `ixx`. The suffix `ixx` stands for a module interface source.
- The **Clang compiler** uses the extension `cppm`. The `m` in the suffix probably stands for module.
- The **GCC compiler** uses no special extension.

The global module fragment starts with the keyword `module` and ends with the module declaration. The global module fragment is the place to use preprocessor directives such as `#include` so that the module unit can compile. Preprocessor entities used inside global module fragment are only visible inside the module.

The second module `math` version, supports the two functions `add` and `getProduct`.

A module definition with a global module fragment

```
1  // math1.ixx
2
3  module;
4
5  #include <numeric>
6  #include <vector>
7
8  export module math;
9
10 export int add(int fir, int sec){
11     return fir + sec;
12 }
13
```

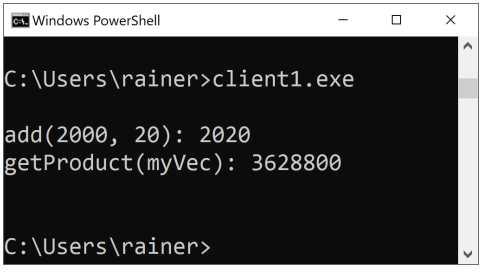
```
14 export int getProduct(const std::vector<int>& vec) {
15     return std::accumulate(vec.begin(), vec.end(), 1, std::multiplies<int>());
16 }
```

I included the necessary headers in the global module fragment (line 3) and the module declaration (line 8).

Use of the improved module math

```
// client1.cpp

#include <iostream>
#include <vector>

import math;

int main() {

    std::cout << '\n';

    std::cout << "add(2000, 20): " << add(2000, 20) << '\n';

    std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    std::cout << "getProduct(myVec): " << getProduct(myVec) << '\n';

    std::cout << '\n';

}
```

The client imports the module `math` and uses its functionality:



Execution of the program `client1.exe`

What are the advantages of modules?

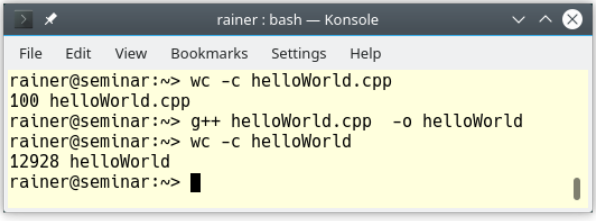
4.2.2 Advantages

Let me start with a simple executable. For obvious reasons, I create a `helloWorld.cpp` program.

A simple hello world program

```
// helloWorld.cpp

#include <iostream>

int main() {
    std::cout << "Hello World" << '\n';
}
```

Making an executable `helloWorld` out of the program `helloWorld.cpp` with GCC[41] increases its size by factor 130.



Size of an object file

The numbers 100 and 12928 in the screenshot represent the number of bytes. Okay. We should have a basic understanding of what's happening under the hood.

4.2.2.1 The Classical Build Process

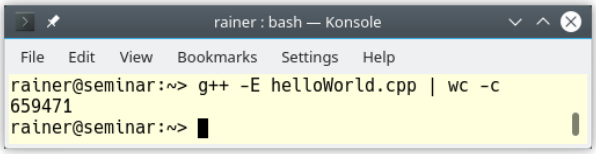
The build process consists of three steps: preprocessing, compilation, and linking.

4.2.2.1.1 Preprocessing

The preprocessor handles the directives as `#include` and `#define`. The preprocessor substitutes `#include` directives with the corresponding header files, and it substitutes the macros (`#define`). Thanks to directives such as `#if`, `#else`, `#elif`, `#ifdef`, `#ifndef`, and `#endif`, parts of the source code can be included or excluded.

You can observe this straightforward text substitution process by using the compiler flag `-E` on GCC/Clang or `/E` on Windows.

[41] http://gcc.gnu.org/



Preprocessors output

WOW!!! The output of the preprocessing step has more than half a million bytes. I don't want to blame GCC; the other compilers are similarly verbose. The output of the preprocessor is the input for the compiler.

The result of this preprocessing step is the translation unit.

4.2.2.1.2 Compilation

The compilation is performed separately on each output of the preprocessor. The compiler parses the C++ source code and converts it into assembly code. The generated file is called an object file and contains the compiled code in binary form. The object file, which can build archives for later reuse, can refer to symbols that don't have a definition. These archives are called static libraries.

The object files that the compiler produces are the inputs for the linker.

4.2.2.1.3 Linking

The linker's output is an executable or a static or shared library. The linker's job is to resolve the references to undefined symbols. Symbols are defined in object files or libraries. The typical error in this phase is that symbols aren't defined or are defined more than once.

C++ inherited the build process from C. It works sufficiently well if you have only one translation unit. But when you have more than one, many issues can occur.

4.2.2.2 Issues of the Build Process

Here's an incomplete list of the flaws in a classical build process. Modules solve these flaws.

4.2.2.2.1 Repeated Substitution

The preprocessor substitutes `#include` directives with the corresponding header files. Let me change my initial `helloWorld.cpp` program to make the repetition visible.

I refactored the program and added two source files `hello.cpp` and `world.cpp`. The source file `hello.cpp` provides the function `hello`, and the source file `world.cpp` provides the function `world`. Both source files include the corresponding headers. Refactoring means the program has the same external behavior as the previous program, `helloWorld.cpp`, but the internal structure is improved. Here are the new files:

- `hello.cpp` and `hello.h`

Implementation of hello

```
// hello.cpp

#include "hello.h"

void hello() {
    std::cout << "hello ";
}
```

Header of hello

```
// hello.h

#include <iostream>

void hello();
```

- `world.cpp` and `world.h`

Implementation of world

```
// world.cpp

#include "world.h"

void world() {
    std::cout << "world";
}
```

Header of world

```
// world.h

#include <iostream>

void world();
```

- `helloWorld2.cpp`

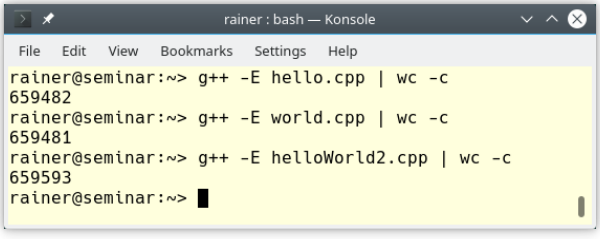
Use of hello and world

```
// helloWorld2.cpp

#include <iostream>

#include "hello.h"
#include "world.h"

int main() {

    hello();
    world();
    std::cout << '\n';

}
```

Building and executing the program works as expected:

Compilation of a simple program

Here is the issue. The preprocessor runs on each source file. Consequentially, the header file `<iostream>` is included three times. Consequently, each source file is blown up to over half a million lines.



Size of the preprocessed source file

This is a waste of compile time.

Unlike header files, a module is only imported once and is literally for free.

4.2.2.2.2 Isolation from Preprocessor Macros

If there is one consensus in the C++ community, it's the following: we should eliminate the preprocessor macros. Why? Using a macro is simply text substitution, excluding any C++ semantics. Of course, this has many negative consequences: for example, it may depend on which sequence you include macros, or macros can clash with already defined macros or names in your application.

Imagine you have two header files `webcolors.h` and `productinfo.h`.

First definition of macro RED

```
// webcolors.h

#define RED   0xFF0000
```

Second definition of macro RED

```
// productinfo.h

#define RED   0
```

When a source file `client.cpp` includes both headers, the value of the macro `RED` depends on the order of the included header. This dependency is very error-prone.

With modules, import order makes no difference.

4.2.2.2.3 Multiple Definitions of Symbols

ODR stands for the One Definition Rule and says in the case of a function:

- A function can have not more than one definition in any translation unit.
- A function can not have more than one definition in the program.

Inline functions with external linkage can be defined in more than one translation unit. The definitions must satisfy the requirement that all definitions have to be the same.

Let's see what my linker says when I try to link a program that violates the one-definition rule. The following code example has two header files, `header.h` and `header2.h`. The main program includes the header files `header.h` twice, breaking the one-definition rule because two definitions of `func` are included.

Definition of the function func

```
// header.h

void func() {}
```

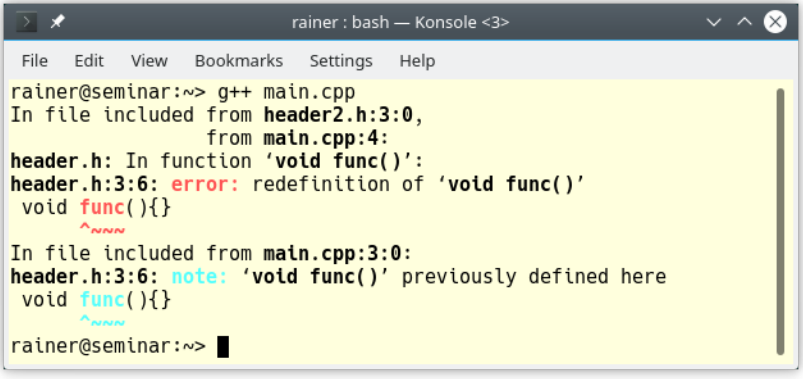
Indirect inclusion of the function definition to func

```
// header2.h

#include "header.h"
```

Double definitions of the function func

```
// main.cpp

#include "header.h"
#include "header2.h"

int main() {}
```

The linker complains about the multiple definitions of `func`:



Breaking the one definition rule

We are used to ugly workarounds, such as putting an include guard around your header. Adding the include guard `FUNC_H` to the header file `header.h` solves the issue.

Using include guards to solve ODR

```
// header.h

#ifndef FUNC_H
#define FUNC_H

void func(){}

#endif
```

With modules, duplicate symbols are very unlikely.

I will now summarize the advantages of modules.

4.2.2.3 All Advantages

Here are the advantages of modules in a concise form:

- Modules are imported only once and are literally for free.
- It makes no difference in which order you import a module.
- Duplicate symbols with modules are very unlikely.
- Modules enable you to express the logical structure of your code. You can explicitly specify names that should be exported or not. Additionally, you can bundle a few modules into a bigger module and provide them to your customer as a logical package.
- Thanks to modules, there is no need to separate your source code into an interface and an implementation part.
- The first experience from real-world examples shows that compilation times decrease by at least ten when you switch from headers to modules.

The Long History

Modules in C++ may be older than you think. My short historic detour should show how long it takes to get something so valuable into the C++ standard.

In 2004, Daveed Vandevoorde wrote a proposal N1736.pdf[42], which described for the first time the idea of modules. It took until 2012 to get a dedicated Study Group (SG2, Modules). In 2017, Clang 5.0 and MSVC 19.1 provided the first implementations. One year later, the Modules TS (technical specification) was finalized. Around the same time, Google proposed the so-called ATOM (Another Take On Modules) proposal (P0947[43]) for modules. In 2019, the Modules TS and the ATOM proposal were merged into the C++20 committee draft (N4842[44]).

Now, it is time to dive into the details of modules.

[42] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1736.pdf
[43] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0947r1.html
[44] https://github.com/cplusplus/draft/releases/tag/n4842

4.2.3 The Details

Modules introduces a few new terms which I want to present before I use them.

4.2.3.1 Terminology

A module consists of one or more module units. A **module unit** is a special translation unit that has a module declaration. The module declaration must be the first declaration of this special translation unit, except the global module fragment. Each module unit is associated with a `ModuleName`:

Module declaration ModuleName

```
[export] module ModuleName[:ModulePartition]
```

The keyword `export` and the module partition `:ModulePartition` are optional.

- **`ModuleName`:** The `ModuleName` can have a dot. Dots have no special meaning but help to express hierarchical modules.
- **`export`:** Module declarations using the keyword `export` are called **module interface units**; otherwise, they are called **module implementation units**.
- **`ModulePartition`:** A module partition is either an **internal partition** or an **interface partition**. An internal partition is not visible from outside the module and provides declarations and definitions of the module. An interface partition participates extends the exported interface of the module.
- **Named module**: A named module is the collection of module units with the same module name.
- **Primary module interface**: Each named module must have precisely one module interface unit that is not a module partition. This module interface unit is called the primary module interface unit. Its exported content will be available to exporting clients.

4.2.3.2 Compiler Support

Using modules, you must use a very recent Clang, GCC, or Microsoft compiler. Even if you have the newest C++ compiler, not all features of modules in C++20 are supported. This holds, in particular, true for the Clang and GCC compiler.

The compilation of a module is challenging. For that reason, I show as an example the compilation of the module with the big three: the Microsoft compiler, the Clang compiler, and the GCC compiler. Additionally, I present the various flags you must use to use modules successfully.

Compilation of a Module

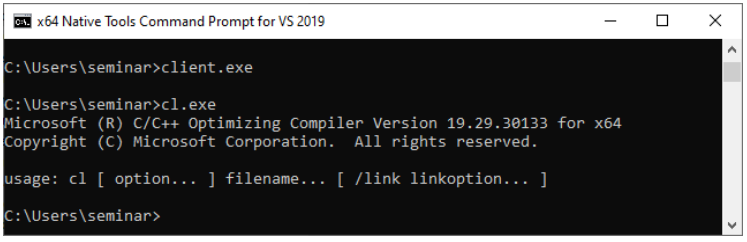
Typically, a module consists of declarations and definitions. Consequentially, compiling a module consists of two steps.

- Precompile the module's declarations into a compiler-specific format.
- Compile the module's definitions into an object file.

Because the module's support of the big three is only partial, I will update this section when appropriate.

4.2.3.2.1 Microsoft Visual Compiler

First, I use the cl.exe 19.29.30133 for the x64 compiler.



Microsoft compiler for modules

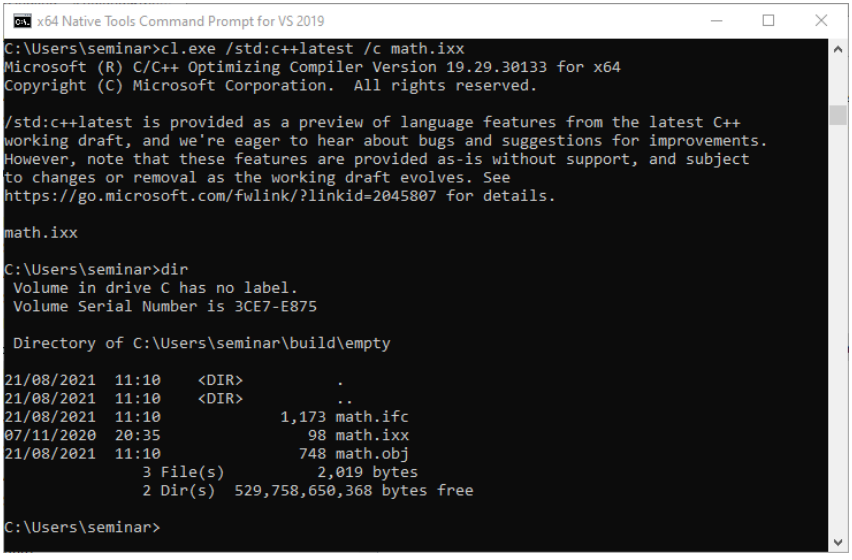
These are the steps to compile and use the module with the Microsoft compiler. I only show the minimal command line. As promised, more details will follow. Additionally, with an older Microsoft compiler, you must use the flag `/std:c++latest`.

Building the executable with the Microsoft compiler

```
1 cl.exe /std:c++latest /c math.ixx
2 cl.exe /std:c++latest client.cpp math.obj
```

- Line 1 creates an obj file `math.obj` and an IFC file `math.ifc`. The IFC is the module and contains the metadata description of the module interface. The binary format of the IFC is modeled after the Internal Program Representation[45] by Gabriel Dos Reis and Bjarne Stroustrup (2004/2005).
- Line 2 creates the executable `client.exe`. The linker cannot find the module without the implicitly used `math.ifc` file from the first step.

[45] https://www.stroustrup.com/gdr-bs-macis09.pdf



Implicitly created IFC file

For obvious reasons, I do not show the output of the program execution.

The Microsoft Visual Compiler provides various options for the creation of modules.

4.2.3.2.2 Module Options

The following table gives an overview of the modules compiler options.

Modules compiler options

Modules Compiler Options	Description
`/interface`	Specifies that the input file is a module interface unit.
`/internalPartition`	Specifies that the input file is an internal partition unit.
`/reference`	Specifies that the input file is an IFC file.
`/ifcSearchDir`	Specifies the search path for the IFC file.
`/ifcOutput`	Specifies the name of the IFC file. If the name is a directory, the compiler generates a name based on the IFC file name or the header unit name.
`/ifcOnly`	Specifies that the compiler only produces an IFC file.

Modules compiler options

Modules Compiler Options	Description
`/exportHeader`	Specifies that the compiler creates a header unit from the input file.
`/headerName`	Specifies that the input file is a header file.
`/headerUnit <header name>=<ifc file name>`	Imports a header unit.
`/translateInclude`	Specifies that the compiler to perform `#include -> import` translation if the header name is an importable header.
`/showResolvedHeader`	Shows the fully resolved path to the header unit after compilation.
`/validateIfcChecksum[-]`	Specifies an extra security check using the stored content hash in the IFC. Off by default.

Additionally, the following general compiler options are often required.

Common `cl.exe` Compiler Options

`cl.exe` compiler options

Compiler Options	Description
`/EHsc`	Specifies the C++ standard exception handling model.
`/TP`	Specifies that all source files are C++ source files.
`/std:c++latest`	Use the latest C++ standard.

I use various compiler options for the module and the ifc file in the following command lines.

- Use the module `math.cppm` to create the obj and ifc file.

Creates the obj and ifc file

```
cl.exe /c /std:c++latest /interface /TP math.cppm
```

- Use the module `math.cppm` to create only the ifc file‘.

Creates only the ifc file

```
cl.exe /c /std:c++latest /ifcOnly /interface /TP math.cppm
```

- Use the module `math.cppm` to create the obj file `math.obj` and the ifc file `mathematic.ifc`.

Creates the ifc file `mathematic.ifc`

```
cl.exe /c /std:c++latest /interface /TP math.cppm /ifcOutput mathematic.ifc
```

- Creates the executable `client.exe` and explicitly use the ifc file `math.inter`.

Use the ifc file `math.inter`

```
cl.exe /std:c++latest client.cpp math.obj /reference math.inter
```

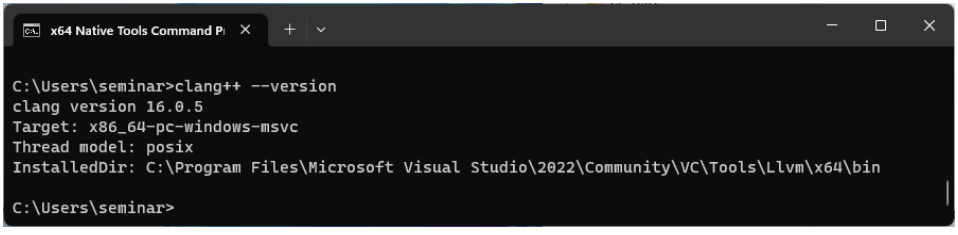
- Creates the executable `client.exe` and explicitly use the ifc file `math.inter` that is in the directory `ifcFiles`.

Use the ifc file `math.inter`

```
cl.exe /std:c++latest client.cpp math.obj /ifcSearchDir ifcFiles /reference math.inter
```

4.2.3.2.3 Clang Compiler

I use the **Clang 16.0.5** compiler.



Clang compiler for modules

With the clang compiler, the module declaration file should have a `cppm` extension. Consequently, I have to rename the `math.ixx` file to `math.cppm`.

A simple math module

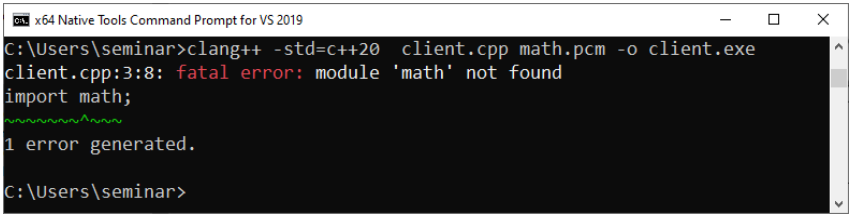
```
// math.cppm

export module math;

export int add(int fir, int sec){
    return fir + sec;
}
```

The client file `client.cpp` is unchanged. These are the necessary steps to create the executable.

Building the executable with the Clang compiler

```
1 clang++ -std=c++20 -c math.cppm --precompile -o math.pcm
2
3 clang++ -std=c++20 client.cpp -fprebuilt-module-path=. math.pcm -o client.exe
```

- Line 1 creates the module `math.pcm`. The suffix `pcm` stands for precompiled module and is equivalent to the ifc file of the Microsoft Visual Compiler. Additionally, the produced module already includes the module definition. Consequentially, the Clang compiler does not produce an object file `math.o`. The option ‘–precompile is necessary for creating the precompiled module.
- Line 3 creates the executable `client.exe`, which uses the module `math.pcm`. The Clang compiler requires that you specify the path to the module with the `-fprebuilt-module-path` flag. If not, the link process fails.



Missing path to the module

The Clang compiler provides various options for the creation of modules.

4.2.3.2.4 Module Options

Clang support three kinds of options for creating and using the module.

4.2.3.2.5 Creating the Module

A module can be created in two ways. From a named module or a header unit. The Clang compiler requires that you always specify the path to the module. The following table shows the options for handling modules.

Modules compiler options

Modules Compiler Options	Description
`--precompile`	Creates the module.
`-fmodule-output`	Creates the module in the working directory having the name of the input file with the extension `.pcm`.
`-fmodule-output=<ModuleName>`	Creates the module having the name `ModuleName`.
`-fmodule-header`	Enables the creation of the module from a header unit. Uses the user search path.
`-fmodule-header=user`	Enables the creation of the module from a header unit. Uses the user search path.
`-fmodule-header=system`	Enables the creation of the module from a header unit. Uses the system search path.
`-xc++-header`	Headers without suffixes can be marked as header.
`-xc++-user-header`	User headers without suffixes can be marked as header.
`-xc++-system-header`	System headers without suffixes can be marked as header.
`-x c++-module`	Enables you to use an importable module unit having not the suffix `.cppm`.
`-fprebuilt-module-path=<ModuleDirectory>`	The compiler looks up the module in the directory `ModuleDirectory`.
`fmodule-file=<ModuleName>=<ModulePath>`	The compiler looks up the module `ModuleName` in the path `ModulePath`. The option `-fprebuilt-module-path` has a higher priority.

An importable module is a module unit that can be imported. Valid suffixes for header units are `h` or `hh`.

For more details, refer to the official Standard C++ Modules[46] documentation. In the following command lines, I use the compiler options for the module, and the ifc file.

- Use the module declaration file `math.cppm` to create the pcm file (`math.pcm`).

[46] https://clang.llvm.org/docs/StandardCPlusPlusModules.html

Creates the pcm

```
clang++ -c -std=c++20 -fmodule-output math.cppm -o math.pcm
```

- Use the module with the extension ixx (`math.ixx`) to create the pcm file (`math.pcm`).

Creates the pcm file from the ixx file

```
clang++ -std=c++20 --precompile -x c++-module math.ixx -o math.pcm
```

- Creates the pcm file and use it

Compile the pcm file and use it

```
clang++ -std=c++20 -c math.pcm -o math.o
clang++ -std=c++20 -fprebuilt-module-path=. math.o client.cpp -o client.exe
```

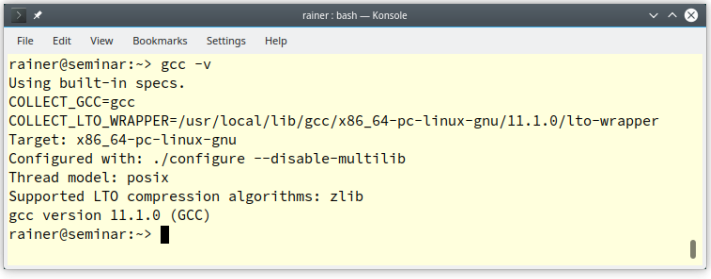
- Uses the pcm file `other.pcm` and compile it

Refering the module `math` in the file `other.pcm`

```
clang++ -std=c++20 -c client.cpp -fmodule-file=math=other.pcm -o client.o
```

4.2.3.2.6 GCC Compiler

The GCC compiler is the last one of the big three. I use the GCC **11.1.0** compiler.



GCC compiler for modules

The GCC Compiler neither supports Window's `*.ixx` nor Clang's `*.cppm` suffix. Consequently, I have to rename the `math.ixx` file into a cpp file: `math.cxx`.

A simple math module

```
// math.cxx

export module math;

export int add(int fir, int sec){
    return fir + sec;
}
```

The client file `client.cpp` is unchanged. These are the necessary steps to create the executable.

Building the executable with the GCC Compiler

```
1 g++ -c -std=c++20 -fmodules-ts math.cxx
2
3 g++ -std=c++20 -fmodules-ts client.cpp math.o -o client
```

- Line 1 creates the module `math.gcm` and the object file `math.o`. I have to specify `-fmodules-ts`. The extension `-fmodules-ts` irritates me because `ts` stands for technical specification. On the contrary, Clang names the same flag `-fmodules`. The module `math.gcm` is in the directory `gcm.cache`. `math.gcm` is the compiled module interface. Presumably, `gcm` stands for GCC compiled module.
- Line 3 creates the executable `client.exe`. It uses the module `math.gcm` implicitly.

GCC supports only a few module options.

4.2.3.2.7 Module Options

The following table shows the few GCC options.

Modules compiler options

Modules Compiler Options	Description
`-fmodules-ts`	Enables modules. Required for GCC.
`-fmodule-header`	Compiles the header units.
`-fmodule-mapper=VALUE`	Specifies the module mapper.
`-fno-module-lazy`	Disables lazy loading.

4.2.3.2.8 Used Compiler

I use mainly the cl.exe compiler from Microsoft in this book. Microsoft has currently (end of 2023) the best support for modules[47]. The Microsoft blog provides a few excellent articles to modules: Overview of modules in C++[48], C++ Modules conformance improvements with MSVC in Visual Studio 2019 16.5[49], and Using C++ Modules in MSVC from the Command Line Part 1: Primary Module Interfaces[50]. Neither Clang nor GCC provides similar introductions, making it quite difficult to use modules with those compilers.

I exemplify the usage of header units in the corresponding chapter.

4.2.3.3 Export

There are three ways to export names in a module interface unit: export specifier, export group, and export namespace.

4.2.3.4 Export Specifier

You can export each name explicitly.

Export specifier

```
export module math;

export int mult(int fir, int sec);

export void doTheMath();
```

4.2.3.5 Export Group

An export group exports all of its names.

[47] https://en.cppreference.com/w/cpp/compiler_support

[48] https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-160&viewFallbackFrom=vs-2019

[49] https://devblogs.microsoft.com/cppblog/c-modules-conformance-improvements-with-msvc-in-visual-studio-2019-16-5/

[50] https://devblogs.microsoft.com/cppblog/using-cpp-modules-in-msvc-from-the-command-line-part-1/

Export group

```
export module math;

export {

    int mult(int fir, int sec);
    void doTheMath();

}
```

4.2.3.6 Export Namespace

Instead of an exported group, you can use an export namespace.

Export namespace

```
export module math;

export namespace math {

    int mult(int fir, int sec);
    void doTheMath();

}
```

When clients use names from an export namespace, they have to qualify them.

Selectively Exporting Names in Namespaces

You can also use the export specifier, the export group, and the export namepace inside a namespace. In this case, only exported names are visible to a module consumer. The following example uses the three ways to export names inside a namespace.

Selectively exporting inside Namespaces

```
export module math;

namespace math {

    export int mult(int fir, int sec); // use with math::mult
    export {
        void doTheMath();              // use with math::doTheMath
    }
    export namespace mathDetails {     // use with math::mathDetails::add
       int add(int fir, int sec);
    }
    int div(int fir, int sec);         // no use outside the module

}
```

The function `div` cannot be used outside the module `math` and has to be fully qualified: `math::div(6, 2)`.

Only names that don't have an internal linkage can be exported.

4.2.3.7 Import

Thanks to `import`, you can import a module, a module partition, or a header unit.

The contextual keyword `import`

```
export module math;

import math.sin;
import math:cos;
import <vector>
```

`import` is a contextual keyword. This means `import` is an identifier and is only a keyword in certain contexts. Before you import an importable entity, you should compile it. If not, you may import an old version of the importable entity.

4.2.3.8 Guidelines for a Module Structure

Let's examine guidelines for how to structure a module.

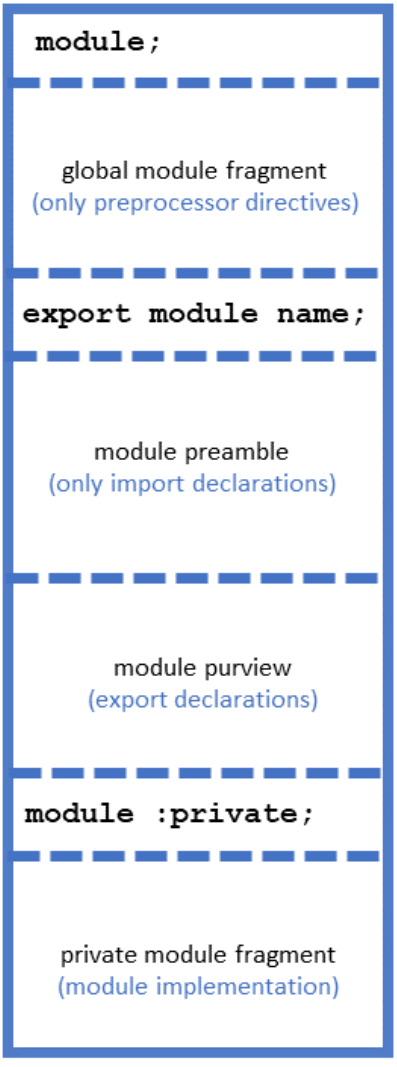
Guidelines for the structure of a module

```
module;                 // starts the global module fragment

#include <headers for libraries not modularized so far>

export module math;     // exporting module declaration; starts the module preamble

import <importing of other modules>

<non-exported declarations> // names only visible inside the module

export namespace math {

    <exported declarations> // exported names

}

module :private;        // not part of the interface

// part of the module implementation that does not cause a recompilation
```

This guideline serves one purpose: to give you a module structure and an idea of what I'm going to write about. So, what's new in this module structure?

- The *global module fragment* starting with the keyword `module` is optional. After it and preceding the module declaration, this is the right place to include headers. Only preprocessor directives are allowed here.
- The required exporting module declaration `export module math` starts the so-called *module preamble* followed by the *module purview* that ends at the end of the translation unit. The module preamble consists of import declarations, and the module purview mainly of export declarations.
- The module purview can have the `private module fragment`. The private module fragment is part of the module's implementation and can only be used in the primary module interface unit. Modifications in the private module fragment do not require the recompilation of the module.
- You can import modules at the beginning of the module purview. The imported modules have module linkage and are not visible outside the module. This observation also applies to the non-exported declarations.
- I put the exported names in namespace `math`, which has the same name as the module.
- The module has only declared names. Let's write about the separation of the interface and the implementation of a module.

Essentially, the module structure boils down to three sections.



Core Parts of the Module Structure

4.2.3.9 Module Interface Unit and Module Implementation Unit

When the module becomes bigger, you should structure it into a module interface unit and one or more module implementation units. Following the previously mentioned guidelines to structure a module, I will refactor the previous version of the `math` module.

4.2.3.9.1 Module Interface Unit

The module interface unit

```
1  // mathInterfaceUnit.ixx
2
3  module;
4
5  #include <vector>
6
7  export module math;
8
9  export namespace math {
10
11    int add(int fir, int sec);
12
13    int getProduct(const std::vector<int>& vec);
14
15 }
```

- The module interface unit contains the exporting module declaration: `export module math` (line 7).
- The names `add` and `getProduct` are exported (lines 11 and 13).
- A module can have only one module interface unit.

4.2.3.9.2 Module Implementation Unit

The module implementation unit

```
1  // mathImplementationUnit.cpp
2
3  module math;
4
5  #include <numeric>
6
7  namespace math {
8
9      int add(int fir, int sec) {
10         return fir + sec;
11     }
12
13     int getProduct(const std::vector<int>& vec) {
14         return std::accumulate(vec.begin(), vec.end(), 1, std::multiplies<int>());
15     }
16 }
```

- The module implementation unit contains non-exporting module declarations: `module math;` (line 3).
- A module can have more than one module implementation unit.

4.2.3.9.3 Main Program

The client uses module math

```
1  // client3.cpp
2
3  #include <iostream>
4  #include <vector>
5
6  import math;
7
8  int main() {
9
10     std::cout << '\n';
11
12     std::cout << "math::add(2000, 20): " << math::add(2000, 20) << '\n';
13
14     std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
15
16     std::cout << "math::getProduct(myVec): " << math::getProduct(myVec) << '\n';
17
18     std::cout << '\n';
19
20 }
```

From the user's perspective, the module `math` (line 6) is included, and the namespace `math` was added.

When my explanations become compiler-dependent, I put them in a separate tip box. This information is generally precious if you decide to try it out.

Building the Executable with the Microsoft Compiler

Manually building the executable includes a few steps.

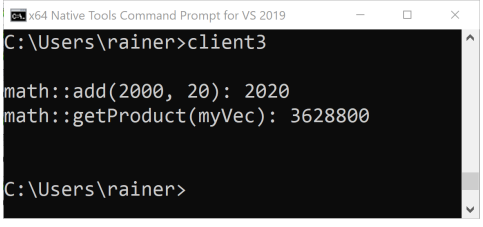
Building a module with a module interface unit and a module implementation unit

```
1  cl.exe /c /std:c++latest mathInterfaceUnit.ixx /EHsc
2  cl.exe /c /std:c++latest mathImplementationUnit.cpp /EHsc
3  cl.exe /c /std:c++latest client3.cpp /EHsc
4  cl.exe client3.obj mathInterfaceUnit.obj mathImplementationUnit.obj
```

- Line 1 creates the object file `mathInterfaceUnit.obj` and the module interface file `math.ifc`.
- Line 2 creates the object file `mathImplementationUnit.obj`.
- Line 3 creates the object file `client3.obj`.
- Line 4 creates the executable `client3.exe`.

For the Microsoft compiler, specify the exception handling model (`/EHsc`), and use the latest C++ standard: `/std:latest`.

Finally, here is the output of the program:



Execution of the program `client2.exe`

4.2.3.10 `Private` Module Fragment

One of the significat advantages of structuring modules into a module interface unit and one or more module implementation units is that modifications in the module implementation units do not affect the module implementation unit and, therefore, requires no recompilation of the importer of the module. Thanks to a `private` module fragment, you can implement a module in one file and declare its last part as its implementation using `module :private;`. Consequently, modifying the `private` module fragment does not cause recompilation of the importer of the module. The following module declaration file `mathInterfaceUnit2.ixx` refactors the module interface unit `mathInterfaceUnit.ixx` and the module implementation unit `mathImplementationUnit.cpp` into one file.

The module declaration file with a `private` module fragment

```
1  // mathInterfaceUnit2.ixx
2
3  module;
4
5  #include <numeric>
6  #include <vector>
7
8  export module math;
9
10 export namespace math {
11
12    int add(int fir, int sec);
13
14    int getProduct(const std::vector<int>& vec);
15
16 }
17
18 module :private;
19
20 int add(int fir, int sec) {
21     return fir + sec;
22 }
23
24 int getProduct(const std::vector<int>& vec) {
25     return std::accumulate(vec.begin(), vec.end(), 1, std::multiplies<int>());
26 }
```

`module: private;` in line 18 denotes the start of the `private` module fragment. Modifying this optional last part of a module declaration file does not cause recompilation of the importer of the module.

4.2.3.11 Submodules and Module Partitions

When your module grows, you want to divide its functionality into manageable components. C++20 modules offer two approaches: submodules and partitions.

4.2.3.11.1 Submodules

A module can import modules and then re-export them.

In the following example, module `math` imports the submodules `math.math1` and `math.math2`.

The module math

```
// mathModule.ixx

export module math;

export import math.math1;
export import math.math2;
```

The expression `export import math.math1` imports module `math.math1` and re-exports it as part of the module `math`.

For completeness, here are the modules `math.math1` and `math.math2`. I used a period to separate the module `math` from its submodules. This period is not necessary.

The submodule math.math1

```
// mathModule1.ixx

export module math.math1;

export int add(int fir, int sec) {
    return fir + sec;
}
```

The submodule math.math2

```
// mathModule2.ixx

export module math.math2;

export {
    int mul(int fir, int sec) {
        return fir * sec;
    }
}
```

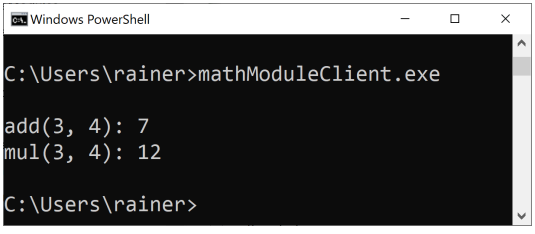
If you look carefully, you recognize a slight difference in the `export` statements in the modules `math`. While `math.math1` uses an export specifier, `math.math2` uses an export group or export block.

Using the `math` module is straightforward from the client's perspective.

The main program

```
// mathModuleClient.cpp

#include <iostream>

import math;

int main() {

    std::cout << '\n';

    std::cout << "add(3, 4): " << add(3, 4) << '\n';
    std::cout << "mul(3, 4): " << mul(3, 4) << '\n';

}
```

Compiling and executing the program gives the expected behavior.



The usage of function modules and submodules

Compilation of the Module and its Submodules with the Microsoft Compiler

Building the executable out of the modules and its submodules

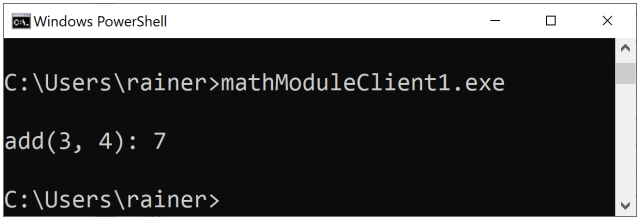
```
cl.exe /c /std:c++latest mathModule1.ixx /EHsc
cl.exe /c /std:c++latest mathModule2.ixx /EHsc
cl.exe /c /std:c++latest mathModule.ixx /EHsc
cl.exe /c /std:c++latest mathModuleClient.cpp /EHsc
cl.exe mathModuleClient.obj mathModule1.obj mathModule2.obj mathModule.obj /EHsc
```

Each compilation process of the three modules creates two artifacts: The IFC file (interface file) `*.ifc`, which is used implicitly in the last line, and the `*.obj file`, which is used explicitly in the last line.

I already mentioned that a submodule is also a module. Each submodule has a module declaration. Consequently, I can create a second client that is interested only in the `math.math1` module.

The main program uses only submodule math.math1

```
// mathModuleClient1.cpp

#include <iostream>

import math.math1;

int main() {

    std::cout << '\n';

    std::cout << "add(3, 4): " << add(3, 4) << '\n';

}
```



The usage of function modules and submodules

The division of modules into modules and submodules is a means for the module designer to give the user of the module the possibility to import fine-grained parts of the module. This observation does not apply to module partitions.

4.2.3.11.2 Module Partitions

A module can be divided into partitions. Each partition consists of a module interface unit (partition interface file) and zero or more module implementation units (see Module Interface Unit and Module Implementation Unit). The interface partition must be exported. The names that the partitions export are imported and re-exported by the primary module interface unit (primary interface file). The name of a partition must begin with the name of the module. The partitions cannot exist standalone. You cannot split a partition into sub-partitions.

The description of module partitions is more challenging to understand than its implementation. In the following lines, I rewrite the `math` module and its submodules `math.math1` and `math.math2` (see Submodules) to module partitions. In this straightforward process, I refer to the shortly introduced terms of module partitions.

Primary interface file

```
1 // mathPartition.ixx
2
3 export module math;
4
5 export import :math1;
6 export import :math2;
```

The primary interface file consists of the exporting module declaration (line 3). It imports and re-exports the partitions `math1` and `math2` using colons (lines 5 and 6). The name of the partitions must begin with the name of the module. Consequently, you don't have to specify them.

First module partition

```
1 // mathPartition1.ixx
2
3 export module math:math1;
4
5 export int add(int fir, int sec) {
6     return fir + sec;
7 }
```

Second module partition

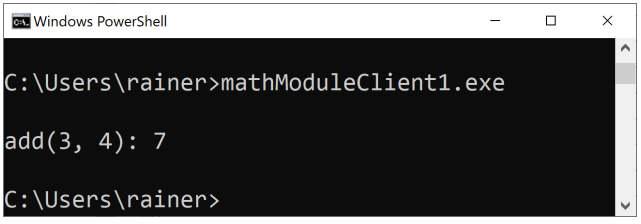
```
1 // mathPartition2.ixx
2
3 export module math:math2;
4
5 export {
6     int mul(int fir, int sec) {
7         return fir * sec;
8     }
9 }
```

Similar to the module declaration, the expressions `export module math:math1` and `export module math:math2` (line 3) declare a module interface partition. A module interface partition is also a module interface unit. `math` stands for the module and `math1` or `math2` for the partition.

Import the module partition

```
// mathModuleClient.cpp

import math;

int main() {

    std::cout << '\n';

    std::cout << "add(3, 4): " << add(3, 4) << '\n';
    std::cout << "mul(3, 4): " << mul(3, 4) << '\n';

}
```

You may have already assumed it: The client program is identical to the one I previously used with submodules. The same observation holds for the creation of the executable and the execution of the program:



The usage of function modules and submodules

4.2.3.12 Reachability versus Visibility

With modules, you have to distinguish between reachability and visibility. When a module exports some entity, an importing client can see and use it. Non-exported entities are not visible but may be reachable.

The module `bar`

```
1  // bar.cppm
2
3  module;
4
5  #include <iostream>
6
7  export module bar;
8
9  struct Foo {
```

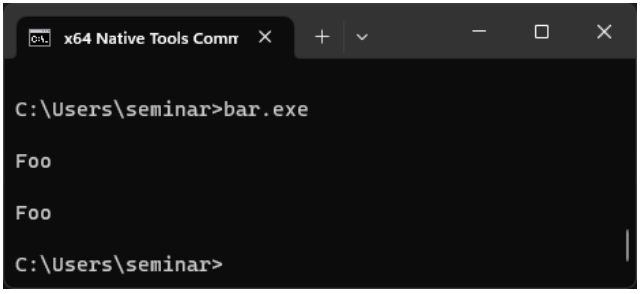
```
10      void writeName() {
11          std::cout << "\nFoo\n";
12      }
13
14  };
15
16  export struct Bar {
17      Foo getFoo() {
18          return Foo{};
19      }
20  };
```

The module `bar` exports the class `Bar`. `Bar` is visible and reachable. On the contrary, `Foo` is not visible.

Using the module `bar`

```
1   #include <utility>
2
3   import bar;
4
5   int main() {
6
7       Bar b;
8       // Foo f;
9       auto f = b.getFoo();
10      f.writeName();
11
12      using FooAlias = decltype(std::declval<Bar>().getFoo());
13      FooAlias f2;
14      f2.writeName();
15
16  }
```

The class `Foo` is not exported and, therefore, not visible. Its usage in line 6 would cause a linker error. On the contrary, `Foo` is reachable because the member function `getFoo` (line 18 in `bar.cppm`) returns it. Consequentially, the function `writeName` (line 8) can be invoked. Furthermore, I can create a type alias to `Foo` (line 12), use it to instantiate `Foo` (line 13), and invoke `writeName` (line 14) on it. The expression `std::declval<Bar>().getFoo()` in line 12 returns the object that a call `Bar.getFoo()` would return. Finally, `decltype` returns the type of this hypothetical object.



Using the module `bar`

4.2.3.13 Module Linkage

Until C++20, C++ supported two kinds of linkage: internal linkage and external linkage.

- **Internal linkage**: Names with internal linkage are not accessible outside the translation unit. Internal linkage includes mainly namespace-scope names declared `static` and members of anonymous namespaces.
- **External linkage**: Names with external linkage are accessible outside the translation unit. External linkage includes names declared not as `static`, class types, and their members, variables, and templates.

Language Linkage

External linkage implies language linkage. Language linkage provides linkage between different programming languages. `C++` is the default language linkage, but you can specify other language linkages, such as `C` linkage.

Language linkage

```
extern "C" {
    int openFile(const char* path);         // C function declaration
}
int main() {
   int fileHandle = openFile("grimm.txt");  // invokes a C function from C++
}
```

The `C` function declaration `openFile` has `C` language linkage and can be called from a C++ program. It supports `C` calling conventions and name mangling.

Modules introduce module linkage:

- **Module linkage**: Names with module linkage are only accessible inside the module. Names have module linkage if they don't have external linkage and they are not exported.

A slight variation of the previous module declaration `mathModuleTemplate.ixx` makes my point. Imagine that I want to return to the user of my function template `sum` not only the result of the addition but also the return type the compiler deduces.

An improved definition of the function template sum

```
1  // mathModuleTemplate1.ixx
2
3  module;
4
5  #include <iostream>
6  #include <typeinfo>
7  #include <utility>
8
9  export module math;
10
11 template <typename T>
12 auto showType(T&& t) {
13     return typeid(std::forward<T>(t)).name();
14 }
15
16 export namespace math {
17
18     template <typename T, typename T2>
19     auto sum(T fir, T2 sec) {
20         auto res = fir + sec;
21         return std::make_pair(res, showType(res));
22     }
23
24 }
```

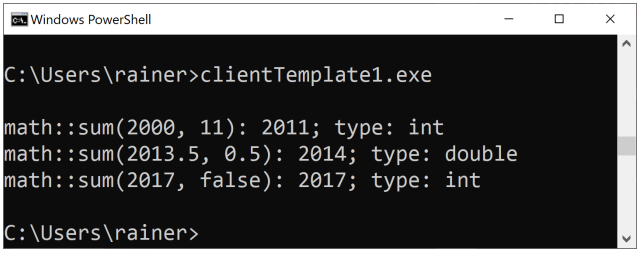
Instead of the sum of the numbers, the function template `sum` returns a `std::pair`[51] (line 21) consisting of the sum and a string representation of the type of the value `res`. Note that I put the function template `showType` (line 11) outside the exported namespace `math` (line 16). Consequently, invoking it from outside the module `math` is impossible. Function template `showType` uses perfect forwarding[52] to preserve the function argument `t` value category. The `typeid`[53] operator queries information about the type at run time (run time type identification (RTTI)[54]).

[51] https://en.cppreference.com/w/cpp/utility/pair
[52] https://www.modernescpp.com/index.php/perfect-forwarding
[53] https://en.cppreference.com/w/cpp/language/typeid
[54] https://en.cppreference.com/w/cpp/types

Use of the improved function template `sum`

```
// clientTemplate1.cpp

#include <iostream>
import math;

int main() {

    std::cout << '\n';

    auto [val, message] = math::sum(2000, 11);
    std::cout << "math::sum(2000, 11): " << val << "; type: " << message << '\n';

    auto [val1, message1] =  math::sum(2013.5, 0.5);
    std::cout << "math::sum(2013.5, 0.5): " << val1 << "; type: " << message1
              << '\n';

    auto [val2, message2] =  math::sum(2017, false);
    std::cout << "math::sum(2017, false): " << val2 << "; type: " << message2
              << '\n';

}
```

Now, the program displays the value of the summation and a string representation of the automatically deduced type.



Use of the improved function template sum

4.2.3.14 Header Units

Header units are a binary representation of header files and conveniently transition from headers to modules. You must replace the `#include` directive with the new `import` statement and add a semicolon (`;`).

Replacing `#include` directives with `import` statement

```
#include <vector>       => import <vector>;
#include "myHeader.h"   => import "myHeader.h";
```

First, `import` respects the same lookup rules as `include`. It means, in the case of the quotes (`"myHeader.h"`), that the lookup first searches in the local directory before it continues with the system search path.

Second, this is way more than text replacement. In this case, the compiler generates something module-like from the `import` directive and treats the result as a module. The importing module statement gets all exportable names from the header. The exported names include macros. Importing these synthesized header units is faster than including header files and comparable in speed and functionality to precompiled headers[55]. You should use header units instead of precompiled headers.

Finally, macros defined before the imported header are not visible inside the auto-generated module.

Modules versus Precompiled Headers

Precompiled headers are a non-standardized way to compile headers in an intermediate form that is faster to process for the compiler. The Microsoft compiler uses the extension `.pch` and the GCC compiler `.gch` for precompiled headers. The main difference between precompiled headers and modules is that modules can selectively export names. Only in a module exported names are visible outside the module.

After this theory, let me try it out.

4.2.3.14.1 Use of Header Units

The following example consists of three files. The header file `head.h` declares the function `hello`, its implementation file `head.cpp` defines the function `hello`, and the client file `helloWorld3.cpp` uses the function `hello`.

The header file `head.h`

```
// head.h

#include <iostream>

void hello();
```

Only the implementation file `head.cpp` and the client file `helloWorld3.cpp` are special. They import the header file `head.h`: `import "head.h";`.

[55] https://en.wikipedia.org/wiki/Precompiled_header

The source file `head.cpp` importing the header unit

```
// head.cpp

import "head.h";

void hello() {

    std::cout << '\n';

    std::cout << "Hello World: header units\n";

    std::cout << '\n';

}
```

The main program `helloWorld3.cpp` using the module

```
// helloWorld3.cpp

import "head.h";

int main() {

    hello();

}
```

I will create and use a header from the header file `head.h` for the Microsoft Visual Compiler and the GCC Compiler. In contrast to the official documentation Standard C++ Modules[56], I could not master header units with the Clang Compiler.

4.2.3.14.2 Microsoft Visual Compiler

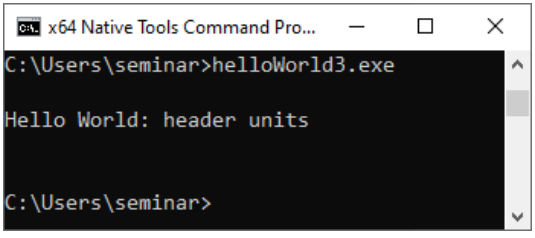
These are the necessary steps to use header units.

[56] https://clang.llvm.org/docs/StandardCPlusPlusModules.html#header-units

Create the module `head.h.ifc` and use it

```
cl.exe /std:c++latest /EHsc /exportHeader head.h
cl.exe /c /std:c++latest /EHsc /headerUnit head.h=head.h.ifc head.cpp
cl.exe /std:c++latest /EHsc /headerUnit head.h=head.h.ifc helloWorld3.cpp head.obj
```

- The flag `/exportHeader` in line 1 causes the creation of the ifc file `head.h.ifc` from the header file `head.h`.
- The implementation file `head.cpp` (line 2) and the client file `helloWorld3.cpp` (line 3) use the header unit. The flag `/headerUnit head.h=head.h.ifc` imports the header and tells the compiler/linker the name of the ifc file for the specified header.



Use the module `head.h.ifc`

4.2.3.14.3 GCC Compiler

Creating and using the module consists of three steps.

Create the module `head.gcm` and use it

```
g++ -fmodules-ts -fmodule-header head.h -std=c++20
g++ -fmodules-ts -c  -std=c++20 head.cpp
g++ -fmodules-ts  -std=c++20 head.o helloWorld3.cpp -o helloWorld3
```

- Line 1 creates the module `head.gcm`. The flag `-fmodule-header` specifies that `head.h` is a header unit.
- The following line creates the object file `head.o`.
- Finally, line 3 creates the executable that implicitly refers to the module `head.gcm`.

4.2.3.14.4 One Drawback

There is one drawback with header units. Not all headers are importable. Which headers are importable is implementation-defined[57], but the C++ standard guarantees all standard library headers are importable headers. The ability to import excludes C headers. They are wrapped in the `std` namespace. For example, `<cstring>` is the C++ wrapper for `<string.h>`. You can quickly identify the wrapped C header because the pattern is: `xxx.h` becomes `cxxx`.

[57] https://en.cppreference.com/w/cpp/language/ub

4.2.4 Further Aspects

4.2.4.1 Macros

Header units support macros, but modules ignore them. The following program consists of a module `macro`, a header `macro.h` used as header unit, and the main program `macroMain.cpp`.

The main program `macroMain.cpp`

```
1  // macroMain.cpp
2
3  #include <iostream>
4
5  import macro;
6  import "macro.h";
7
8  int main() {
9
10     std::cout << '\n';
11
12     std::cout << MACRO_HEADER_UNIT << '\n';
13     std::cout << MACRO_MODULE << '\n';
14
15     std::cout << '\n';
16
17 }
```

The program `macroMain.cpp` uses the two macros `MACRO_HEADER_UNIT` (line 12), and `MACRO_MODULE` (line 13). `MACRO_MODULE_UNIT` is defined in the header unit used header `macro.h`, and `MACRO_MODULE` in the module `macro`.

The as header unit used header `macro.h`

```
// macro.h

#define MACRO_HEADER_UNIT "macro header unit"
```

The module `macro.ixx`

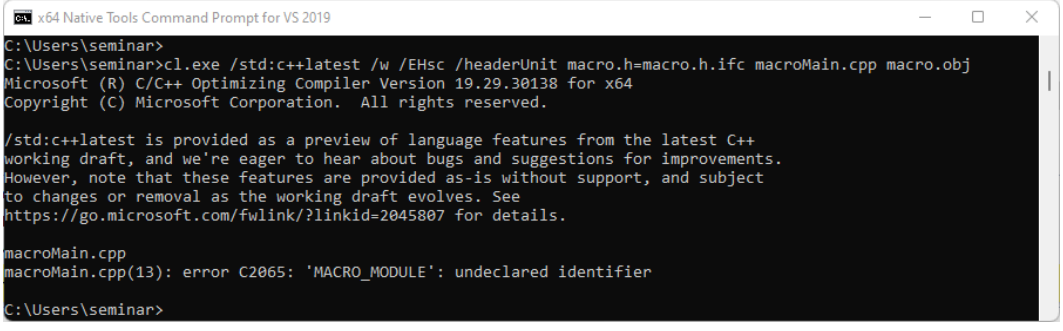
```
// macro.ixx

module;

#define MACRO_HEADER_UNIT "macro module"

export module macro;
```

The compilation of the program consists of the following three steps:

Importing of a header unit and a macro

```
1  cl.exe /std:c++latest /EHsc /exportHeader macro.h
2  cl.exe /std:c++latest /EHsc /c macro.ixx
3  cl.exe /std:c++latest /EHsc /headerUnit macro.h=macro.h.ifc macroMain.cpp macro.obj
```

Line 1 creates the header unit, line 2 the macro, and the last uses both. As expected, the final compilation step fails because the macro `MACRO_MODULE` (line 13 in `macroMain.cpp`) is not visible in the main program.



The macro `MACRO_MODULE` is not visible

You cannot export a macro from a module but include a header into the module. This header can have macros. The global module fragment is the right place to insert a header and, thus, a macro.

```
// macro.ixx

module;

#include "macro.h"

export module macro;
```

4.2.4.2 Templates in Modules

I often hear the question: How do modules export templates? When you instantiate a template, its definition must be available. For this reason, template definitions are hosted in headers. Conceptually, the usage of a template has the following structure.

4.2.4.2.1 Without Modules

- `templateSum.h`

Definition of the function template sum

```
// templateSum.h

template <typename T, typename T2>
auto sum(T fir, T2 sec) {
    return fir + sec;
}
```

- `sumMain.cpp`

Use of the template sum

```
// sumMain.cpp

#include <templateSum.h>

int main() {

    sum(1, 1.5);

}
```

The `main` program includes the header `templateSum.h`. The call `sum(1, 1.5)` triggers the template instantiation. In this case, the compiler generates out of the function template `sum` the concrete function `sum`, which takes an `int` and a `double` as arguments. If you want to visualize this process, use the example on C++ Insights[58].

[58] https://cppinsights.io/

4.2.4.2.2 With Modules

With C++20, templates can and should be in modules. Modules have a unique internal representation that is neither source code nor assembly. This representation is a kind of abstract syntax tree[59] (AST). Thanks to this AST, the template definition is available during template instantiation.

I define the function template `sum` in module `math` in the following example.

- `mathModuleTemplate.ixx`

Definition of the function template sum

```
// mathModuleTemplate.ixx

export module math;

export namespace math {

    template <typename T, typename T2>
    auto sum(T fir, T2 sec) {
        return fir + sec;
    }

}
```

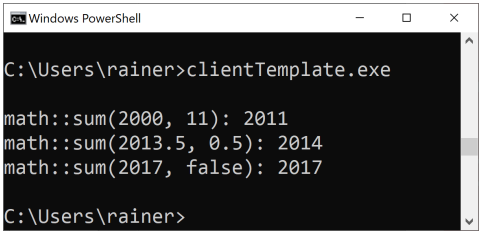
- `clientTemplate.cpp`

Use of the function template `sum`

```
// clientTemplate.cpp

#include <iostream>
import math;

int main() {

    std::cout << '\n';

    std::cout << "math::sum(2000, 11): " << math::sum(2000, 11) << '\n';

    std::cout << "math::sum(2013.5, 0.5): " << math::sum(2013.5, 0.5) << '\n';

    std::cout << "math::sum(2017, false): " << math::sum(2017, false) << '\n';

}
```

[59] https://en.wikipedia.org/wiki/Abstract_syntax_tree

The command line to compile the program is not different from the previous ones. Consequently, I skip it and present the output of the program directly:



Use of the function template sum

With modules, we get a new kind of linkage.

4.2.4.3 Migrating from Headers to Modules

Broadly speaking, there are two options when migrating from headers to modules. You can use header units instead of headers, or reimplement your header in one module or in a module partition. From the implementers perspective and the users perspective, both options are very convenient.

For convenience, I refer to modules and module partitions as modules for the rest of this section.

Header units are a no-brainer for the implementer. Headers are a good starting point for modules because they already provide the necessary modularization of the system. The user of the header units has to replace the `#include` directive with the new `import` statement and add a semicolon (`;`).

Replacing headers with a header units

```
#include <vector>  => import <vector>;
#include "math.h"  => import "math.h";
```

Using a module makes no significant difference for the user. Importing a module is quite similar to including a header.

Replacing headers with modules

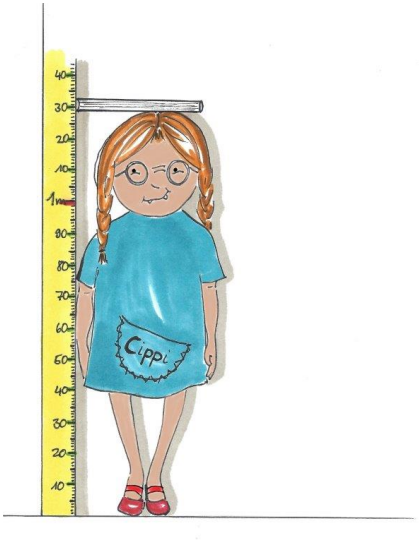
```
#include <vector>  => import vector;
#include "math.h"  => import math;
```

The burden of modules lies on their implementer, but the implementer can do this migration successively, thanks to header units. Therefore, a sound migration strategy is to start your migration with header units. Only headers that are not importable must be implemented as modules. Additionally, new functionality should be implemented as a named module.

Distilled Information

- Modules overcome the deficiencies of headers and macros. Their import is literally for free, and in contrast to macros, the sequence you import does not matter. Additionally, they overcome name collisions.
- A module consists of one module interface unit and arbitrarily many module implementation unit. The module interface must have the exporting module declaration and the module implementation units must have the non-exporting module declaration. Names that are not exported in the module interface have module linkage and cannot be used outside the module.
- Modules can have headers or import and re-export other modules.
- The standard library in C++20 is not modularized. With C++20, Building your modules is a challenging task.
- To structure large software systems, modules provide two ways: submodules and partitions. In contrast to a partition, a submodule can live on its own.
- Thanks to header units, you can replace an include statement with an import statement, and the compiler autogenerates a module.
- Header units do support macros, but modules not.
- A sound migration strategy from headers to modules or module partitions is to start your migration with header units. Only headers that are not importable must be implemented as modules. Additionally, new functionality should be implemented as a named module.

4.3 Equality Comparison and Three-Way Comparison



Cippi measures how big she is

C++20 empowers your to define or autogenerate the equality operator. The equality operator determines for two values A and B, whether A == B , or A != B. Additionally, if your values A and B should support ordering, use the three-way comparison operator `<=>`. The three-way comparison operator is often called the spaceship operator. The spaceship operator determines for two values A and B, whether A < B, A == B, or A > B. As with the equality operator, you can define the spaceship operator, or the compiler can autogenerate it for you.

To appreciate the advantages of the three-way comparison operator, let me start with the classical way of doing it.

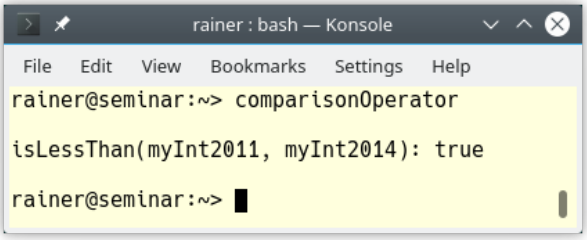
4.3.1 Comparison before C++20

I implemented a simple `int` wrapper `MyInt`. Of course, I want to compare `MyInt`. Here is my solution using the function template `isLessThan`.

MyInt supports less than comparisons

```
// comparisonOperator.cpp

#include <iostream>

struct MyInt {
    int value;
    explicit constexpr MyInt(int val): value{val} { }
    bool operator < (const MyInt& rhs) const {
        return value < rhs.value;
    }
};

template <typename T>
constexpr bool isLessThan(const T& lhs, const T& rhs) {
    return lhs < rhs;
}

int main() {

    std::cout << std::boolalpha << '\n';

    MyInt myInt2011(2011);
    MyInt myInt2014(2014);

    std::cout << "isLessThan(myInt2011, myInt2014): "
              << isLessThan(myInt2011, myInt2014) << '\n';

   std::cout << '\n';

}
```

The program works as expected:



Use of the less than operator

Honestly, `MyInt` is an unintuitive type. When you define one of the six ordering relations, you should define all of them. Intuitive types should be at least semiregular. Now, I have to write a lot of boilerplate code. Here are the missing five operators.

The five missing comparison operators

```
bool operator == (const MyInt& rhs) const {
    return value == rhs.value;
}
bool operator != (const MyInt& rhs) const {
    return !(*this == rhs);
}
bool operator <= (const MyInt& rhs) const {
    return !(rhs < *this);
}
bool operator > (const MyInt& rhs)  const {
    return rhs < *this;
}
bool operator >= (const MyInt& rhs) const {
    return !(*this < rhs);
}
```

Now, let's jump to C++20 and the equality operator and three-way comparison operator.

4.3.2 Comparison since C++20

You can define the comparison operator or the three-way comparison operator or request it from the compiler with `= default`. Let me start with the equality operator.

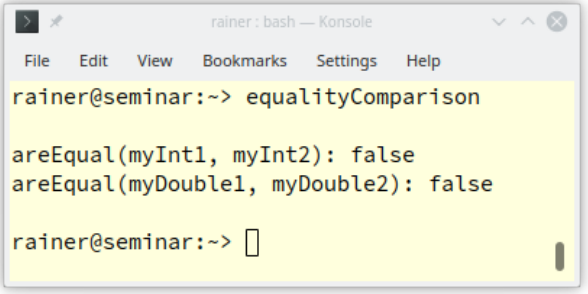
4.3.2.1 Equality Operator

When you define or request the equality operator from the compiler with `= default`, you automatically get the equality and inequality operators: `==`, and `!=`.

Implement or request the equality operator

```
// equalityComparison.cpp

#include <iostream>

struct MyInt {
    int value;
    explicit constexpr MyInt(int val): value{val} { }
    bool operator==(const MyInt& rhs) const {
        return value == rhs.value;
    }
};

struct MyDouble {
    double value;
    explicit constexpr MyDouble(double val): value{val} { }
    bool operator==(const MyDouble&) const = default;
};

template <typename T>
constexpr bool areEqual(const T& lhs, const T& rhs) {
    return lhs == rhs;
}

int main() {

    std::cout << std::boolalpha << '\n';

    MyInt myInt1(2011);
    MyInt myInt2(2014);

    std::cout << "areEqual(myInt1, myInt2): "
              << areEqual(myInt1, myInt2) << '\n';

    MyDouble myDouble1(2011);
    MyDouble myDouble2(2014);

    std::cout << "areEqual(myDouble1, myDouble2): "
              << areEqual(myDouble1, myDouble2) << '\n';

    std::cout << '\n';

}
```

The user-defined (line 8) and the compiler-generated (line 16) equality operators work as expected. Both return a boolean.



Use of the user-defined and compiler-generated equality operator

4.3.2.2 Three-Way Comparison Operator

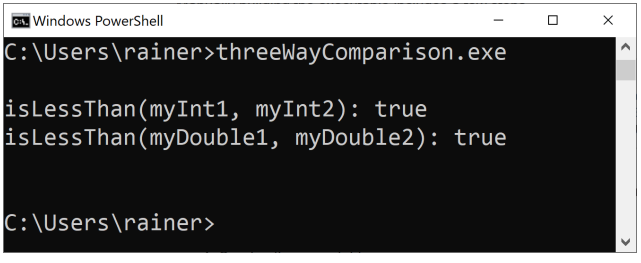
When you define or request the three-way operator from the compiler with `= default`, you automatically get all six comparison operators: `==`, `!=`, `<`, `<=`, `>`, and `>=`. The member function must be `const` and the parameter a `const` lvalue reference.

Implement or request the three-way comparison operator

```
1  // threeWayComparison.cpp
2
3  #include <compare>
4  #include <iostream>
5
6  struct MyInt {
7      int value;
8      explicit MyInt(int val): value{val} { }
9      auto operator<=>(const MyInt& rhs) const {
10         return value <=> rhs.value;
11     }
12 };
13
14 struct MyDouble {
15     double value;
16     explicit constexpr MyDouble(double val): value{val} { }
17     auto operator<=>(const MyDouble&) const = default;
18 };
19
20 template <typename T>
21 constexpr bool isLessThan(const T& lhs, const T& rhs) {
```

```
22      return lhs < rhs;
23  }
24
25  int main() {
26
27      std::cout << std::boolalpha << '\n';
28
29      MyInt myInt1(2011);
30      MyInt myInt2(2014);
31
32      std::cout << "isLessThan(myInt1, myInt2): "
33                << isLessThan(myInt1, myInt2) << '\n';
34
35      MyDouble myDouble1(2011);
36      MyDouble myDouble2(2014);
37
38      std::cout << "isLessThan(myDouble1, myDouble2): "
39                << isLessThan(myDouble1, myDouble2) << '\n';
40
41      std::cout << '\n';
42
43  }
```

The user-defined (line 9) and the compiler-generated (line 17) three-way comparison operators work as expected.



Use of the user-defined and compiler-generated spaceship operator

In this case there are a few subtle differences between the user-defined and the compiler-generated three-way comparison operator. The compiler-deduced return type for `MyInt` (line 9) supports strong ordering, and the compiler-deduced return type of `MyDouble` (line 17) supports partial ordering. Additionally, the three-way comparison operator requires the header `<compare>`.

Automatic Comparison of Pointers

The compiler-generated comparison operator compares the pointers but not the referenced objects.

Automatic Comparison of Pointers

```
1  // spaceshipPoiner.cpp
2
3  #include <iostream>
4  #include <compare>
5  #include <vector>
6
7  struct A {
8     std::vector<int>* pointerToVector;
9     auto operator <=> (const A&) const = default;
10 };
11
12 int main() {
13
14    std::cout << '\n';
15
16    std::cout << std::boolalpha;
17
18    A a1{new std::vector<int>()};
19    A a2{new std::vector<int>()};
20
21    std::cout << "(a1 == a2): " << (a1 == a2) << "\n\n";
22
23 }
```

Astonighly, the result of `a1 == a2` (line 21) is `false` and not `true` because the adresses of `std::vector<int>*` are compared.

```
(a1 == a2): false
```

Comparison of pointers

There are three comparison categories.

4.3.3 Comparison Categories

The names of the three comparison categories are strong ordering (`std::strong_ordering`), also known as total ordering, weak ordering (`std::weak_ordering`), and partial ordering (`std::partial_ordering`).

Strong ordering is also called total ordering.

For a type `T` the three following properties distinguish the three comparison categories.

1. `T` supports all six relational operators: `==`, `!=`, `<`, `<=`, `>`, and `>=` (short: Relational Operator)
2. All equivalent values are indistinguishable: (short: Equivalence)
3. All values of `T` are comparable: For arbitrary values `a` and `b` of `T`, one of the three relations `a < b`, `a == b` , and `a > b` must be true (short: Comparable)

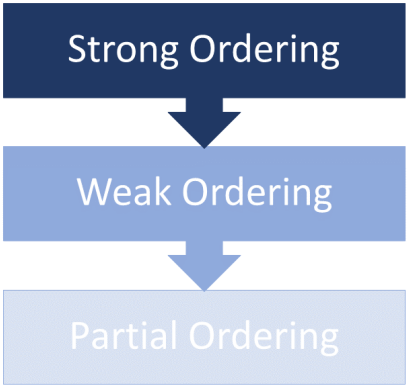
Integral types or strings are typical examples of strong ordering. The ordering is called weak when comparing the absolute value of signed integral types or strings case-insensitive. Strong ordering determines the identity of values, but weak ordering the equivalence of values. Additionally, two arbitrary floating-point values need not to be comparable: for `a = 5.5` and `b = NaN` (Not a Number) neither of the following expressions returns `true`: `a < Nan`, `a == Nan` , or `a > Nan`. This means floating-point values support partial ordering.

Based on the three properties, distinguishing the three comparison category is straightforward:

Strong, weak, and partial ordering

Comparison Category	Relational Operator	Equivalence	Comparable
Strong Ordering	yes	yes	yes
Weak Ordering	yes		yes
Partial Ordering	yes		

A type supporting strong ordering supports implicitly weak and partial ordering. The same holds for weak ordering. A type supporting weak ordering also supports partial ordering. The other directions do not apply.



Strong, weak, and partial ordering

Suppose the declared return type of the three-way comparison operator is `auto`. In that case, the actual return type is the common comparison category of the base and member subobject and the member array elements to be compared.

Let me give you an example for this rule:

Implement or request the three-way comparison operator

```
// strongWeakPartial.cpp

#include <compare>

struct Strong {
    std::strong_ordering operator <=> (const Strong&)  const = default;
};

struct Weak {
    std::weak_ordering operator <=> (const Weak&)  const = default;
};

struct Partial {
    std::partial_ordering operator <=> (const Partial&)  const = default;
};

struct StrongWeakPartial {

    Strong s;
    Weak w;
    Partial p;

    auto operator <=> (const StrongWeakPartial&)  const = default;
```

```
24
25      // FINE
26      // std::partial_ordering operator <=> (const StrongWeakPartial&) const = default;
27
28      // ERROR
29      // std::strong_ordering operator <=> (const StrongWeakPartial&) const = default;
30      // std::weak_ordering operator <=> (const StrongWeakPartial&) const = default;
31
32  };
33
34  int main() {
35
36      StrongWeakPartial a1, a2;
37
38      a1 < a2;
39
40  }
```

The type `StrongWeakPartial` has subtypes supporting strong (line 6), weak (line 10), and partial ordering (line 14). The common comparison category for the type `StrongWeakPartial` (line 17) is, therefore, `std::partial_ordering`. Using a more powerful comparison category, such as strong ordering (line 29) or weak ordering (line 30), would result in a compile-time error.

4.3.3.1 Values of the Comparision Categories

Each of the three comparison categories `std::strong_ordering`, `std::weak_ordering`, and `std::partial_ordering` has there values for denoting less, equal, or greater.

4.3.3.1.1 `std::strong_ordering`

```
std::strong_ordering::less
std::strong_ordering::equal, or std::strong_ordering::equivalent
std::strong_ordering::greater
```

4.3.3.1.2 `std::weak_ordering`

```
std::weak_ordering::less
std::weak_ordering::equivalent
std::weak_ordering::greater
```

4.3.3.1.3 `std::partial_ordering`

```
std::partial_ordering::less
std::partial_ordering::equivalent
std::partial_ordering::greater
std::partial_ordering::unordered
```

Equality of values support `std::strong_ordering` can either be expressed with `std::strong_ordering::equal` or `std::strong_ordering::equivalent`. `std::partial_ordering::unordered` represent values supporting `std::weak_ordering`, which neither support less, equal, or greater.

You can use the comparison categories and their values to explicitly define the three-way comparison operator. The following code snippet does it for the simple type `MyInt`.

Explicit definition of the three-way comparision operator

```
struct MyInt {
    int value;
    explicit MyInt(int val): value{val} { }
    std::strong_ordering operator<=>(const MyInt& rhs) const {
        return value == rhs.value ? std::strong_ordering::equal :
               value < rhs.value ? std::strong_ordering::less :
                                   std::strong_ordering::greater;
    }
};
```

Now, I want to focus on the compiler-generated spaceship operator.

4.3.4 Compiler-Generated Equality and Spaceship Operator

The compiler-generated comparison operators are implicit `constexpr` and `noexcept`[60], and performs a lexicographical comparison. The compaision operator is defined for all fundamental types for which the relational operators are defined. Additionally, the compiler-generated three-way comparison operator needs the header `<compare>`.

You can even directly use the three-way comparison operator.

4.3.4.1 Direct Use of the Three-Way Comparison Operator

The program `spaceship.cpp` directly uses the spaceship operator.

[60] https://www.modernescpp.com/index.php/c-core-guidelines-the-noexcept-specifier-and-operator

Implement or request the three-way comparison operator

```
// spaceship.cpp

#include <compare>
#include <iostream>
#include <string>
#include <vector>

int main() {

    std::cout << '\n';

    int a(2011);
    int b(2014);
    auto res = a <=> b;
    if (res < 0) std::cout << "a < b" << '\n';
    else if (res == 0) std::cout << "a == b" << '\n';
    else if (res > 0) std::cout << "a > b" << '\n';

    std::string str1("2014");
    std::string str2("2011");
    auto res2 = str1 <=> str2;
    if (res2 < 0) std::cout << "str1 < str2" << '\n';
    else if (res2 == 0) std::cout << "str1 == str2" << '\n';
    else if (res2 > 0) std::cout << "str1 > str2" << '\n';

    std::vector<int> vec1{1, 2, 3};
    std::vector<int> vec2{1, 2, 3};
    auto res3 = vec1 <=> vec2;
    if (res3 < 0) std::cout << "vec1 < vec2" << '\n';
    else if (res3 == 0) std::cout << "vec1 == vec2" << '\n';
    else if (res3 > 0) std::cout << "vec1 > vec2" << '\n';

    std::cout << '\n';

}
```

The program uses the spaceship operator for `int` (line 14), `string` (line 21), and `vector` (line 28). Here is the output of the program.

```
a < b
str1 > str2
vec1 == vec2
```

Direct use of the spaceship operator

As already mentioned, these comparisons are `constexpr` and could be performed at compile time.

4.3.4.2 Comparison at Compile Time

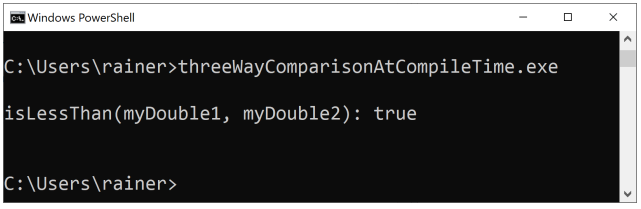
The three-way comparison operator is implicit `constexpr`. Consequently, I can simplify the previous program `threeWayComparison.cpp` and compare `MyDouble` in the following program at compile time.

A compiler-generated constexpr three-way comparison operator

```
// threeWayComparisonAtCompileTime.cpp

#include <compare>
#include <iostream>

struct MyDouble {
    double value;
    explicit constexpr MyDouble(double val): value{val} { }
    auto operator<=>(const MyDouble&) const = default;
};

template <typename T>
constexpr bool isLessThan(const T& lhs, const T& rhs) {
    return lhs < rhs;
}

int main() {

    std::cout << std::boolalpha << '\n';

    constexpr MyDouble myDouble1(2011);
    constexpr MyDouble myDouble2(2014);

    constexpr bool res = isLessThan(myDouble1, myDouble2);

    std::cout << "isLessThan(myDouble1, myDouble2): "
              << res << '\n';

```

```
29      std::cout << '\n';
30
31  }
```

I ask for the result of the comparison at compile time (line 24), and I get it.



Use of the `constexpr` compiler-generated spaceship operator

4.3.4.3 Lexicographical Comparison

The compiler-generated comparison operator performs the lexicographical comparison. Lexicographical comparison, in this case, means that all base classes are compared left to right and all non-static members of the class in their declaration order. I have to qualify: for performance reasons, the compiler-generated equality operator behaves differently in C++20. I will write about this exception in the section for the optimized == and != operators.

The post "Simplify Your Code With Rocket Science: C++20's Spaceship Operator"[61] from the Microsoft C++ Team Blog provides an impressive example of lexicographical comparison. For readability, I added a few comments.

Lexicographical comparison

```
1   struct Basics {
2     int i;
3     char c;
4     float f;
5     double d;
6     auto operator<=>(const Basics&) const = default;
7   };
8
9   struct Arrays {
10    int ai[1];
11    char ac[2];
12    float af[3];
13    double ad[2][2];
14    auto operator<=>(const Arrays&) const = default;
15  };
```

[61] https://devblogs.microsoft.com/cppblog/simplify-your-code-with-rocket-science-c20s-spaceship-operator/

```
16
17 struct Bases : Basics, Arrays {
18   auto operator<=>(const Bases&) const = default;
19 };
20
21 int main() {
22   constexpr Bases a = { { 0, 'c', 1.f, 1. },                      // Basics
23                         { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, // Arrays
24                         { { 1., 2. }, { 3., 4. } } } };
25   constexpr Bases b = { { 0, 'c', 1.f, 1. },                      // Basics
26                         { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, // Arrays
27                         { { 1., 2. }, { 3., 4. } } } };
28   static_assert(a == b);
29   static_assert(!(a != b));
30   static_assert(!(a < b));
31   static_assert(a <= b);
32   static_assert(!(a > b));
33   static_assert(a >= b);
34 }
```

I assume the most challenging aspect of the program is not the spaceship operator but the initialization of `Bases` via aggregate initialization (lines 22 and 25). Aggregate initialization enables us to directly initialize the members of a class type (`class`, `struct`, `union`) when the members are all `public`. In this case, you can use brace initialization. Aggregate initialization is discussed in more detail in the section on designated initializers in C++20.

Optimized == and != Operators

There is an optimization potential for string-like or vector-like types. In this case, a == and != may be faster than the compiler-generated comparison operator. The == and != operators can stop if the two values compared have different lengths. Otherwise, if one value were a prefix of the other, lexicographical comparison would compare all elements until the end of the shorter value. Consequently, the compiler-generated == and != operators compare, in the case of a string-like or a vector-like type, first their lengths and then their content if necessary. The standardization committee was aware of this performance issue and fixed it with the paper P1185R2[62].

Now, it's time for something new in C++. C++20 introduces the concept of rewriting expressions.

4.3.5 Rewriting Expressions

When the compiler sees something such as `a < b`, it rewrites it to `(a <=> b) < 0` using the spaceship operator.

[62] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1185r2.html

Of course, the rule applies to all six comparison operators:

`a OP b` becomes `(a <=> b) OP 0`. It's even better. If there is no conversion of the type(a) to type(b), the compiler generates the new expression `0 OP (b <=> a)`.

For example, this means for the less-than operator, if `(a <=> b) < 0` does not work, the compiler generates `0 < (b <=> a)`. In essence, the compiler takes care of the symmetry of the comparison operators.

Here are a few examples of rewriting expressions:

Rewriting expressions with MyInt

```
// rewritingExpressions.cpp

#include <compare>
#include <iostream>

class MyInt {
 public:
    constexpr MyInt(int val): value{val} { }
    auto operator<=>(const MyInt& rhs) const = default;
 private:
    int value;
};

int main() {

    std::cout << '\n';

    constexpr MyInt myInt2011(2011);
    constexpr MyInt myInt2014(2014);

    constexpr int int2011(2011);
    constexpr int int2014(2014);

    if (myInt2011 < myInt2014) std::cout << "myInt2011 < myInt2014" << '\n';
    if ((myInt2011 <=> myInt2014) < 0) std::cout << "myInt2011 < myInt2014" << '\n';

    std::cout << '\n';

    if (myInt2011 < int2014) std:: cout << "myInt2011 < int2014" << '\n';
    if ((myInt2011 <=> int2014) < 0) std:: cout << "myInt2011 < int2014" << '\n';

    std::cout << '\n';

    if (int2011 < myInt2014) std::cout << "int2011 < myInt2014" << '\n';
```

```
35      if (0 < (myInt2014 <=> int2011)) std:: cout << "int2011 < myInt2014" << '\n';
36
37      std::cout << '\n';
38
39  }
```

I used in line 24, line 29, and line 34 the less-than operator and the corresponding spaceship expression. Line 35 is the most interesting one. It exemplifies how the comparison `(int2011 < myInt2014)` triggers the generation of the spaceship expression `(0 < (myInt2014 <=> int2011)`.

```
myInt2011 < myInt2014
myInt2011 < myInt2014

myInt2011 < int2014
myInt2011 < int2014

int2011 < myInt2014
int2011 < myInt2014
```

Rewriting expressions

Honestly, `MyInt` has an issue: its constructor taking one argument should be declared `explicit`. Constructors taking one argument as `MyInt(int val)` (line 8) are conversion constructors. This means that an instance from `MyInt` can be generated from any integral or floating-point value because each integral or floating-point value can implicitly be converted to an `int`.

Let me fix this issue and make the constructor `MyInt(int val)` explicit. To support the comparison of `MyInt` and `int`, `MyInt` needs an additional three-way comparison operator for `int`.

An additional three-way comparison operator for int

```
1   // threeWayComparisonForInt.cpp
2
3   #include <compare>
4   #include <iostream>
5
6   class MyInt {
7    public:
8       constexpr explicit MyInt(int val): value{val} { }
9
10      auto operator<=>(const MyInt& rhs) const = default;
11
12      constexpr auto operator<=>(const int& rhs) const {
13          return value <=> rhs;
14      }
```

```
15  private:
16     int value;
17  };
18
19  template <typename T, typename T2>
20  constexpr bool isLessThan(const T& lhs, const T2& rhs) {
21     return lhs < rhs;
22  }
23
24  int main() {
25
26     std::cout << std::boolalpha << '\n';
27
28     constexpr MyInt myInt2011(2011);
29     constexpr MyInt myInt2014(2014);
30
31     std::cout << "isLessThan(myInt2011, myInt2014): "
32               << isLessThan(myInt2011, myInt2014) << '\n';
33
34     std::cout << "isLessThan(int2011, myInt2014): "
35               << isLessThan(int2011, myInt2014) << '\n';
36
37     std::cout << "isLessThan(myInt2011, int2014): "
38               << isLessThan(myInt2011, int2014) << '\n';
39
40     constexpr auto res = isLessThan(myInt2011, int2014);
41
42    std::cout << '\n';
43
44  }
```

I defined in (line 10) the three-way comparison operator and declared it `constexpr`. User-defined comparison operators are not implicitly `constexpr`, unlike the compiler-generated comparison operators. The comparison of `MyInt` and `int` is possible in each combination (lines 34, 37, and 40).

```
isLessThan(myInt2011, myInt2014): true
isLessThan(int2011, myInt2014): true
isLessThan(myInt2011, int2014): true
```

Three-way comparison operator for `int`

Honestly, the implementation of the various three-way comparison operators is very elegant. The compiler auto-generates the comparison of `MyInt`, and the user defines the comparison with `int` explicitly. Additionally, you have to define only two operators to get 18 = 3 * 6 combinations of

comparison operators thanks to reordering. The three stands for the combinations `int OP MyInt`, `MyInt OP MyInt`, and `MyInt OP int` and the six for six comparison operators.

4.3.6 User-Defined and Auto-Generated Comparison Operators

When you define one of the six comparison operators and auto-generate all of them using the spaceship operator, there is one question: Which one has the higher priority? For example, this implementation `MyInt` has a user-defined less-than-and-equal-to operator and compiler-generated six comparison operators.

Let's see what happens.

The interplay of user-defined and auto-generated operators

```
// userDefinedAutoGeneratedOperators.cpp

#include <compare>
#include <iostream>

class MyInt {
 public:
    constexpr explicit MyInt(int val): value{val} { }
    bool operator == (const MyInt& rhs) const {
        std::cout << "==  " << '\n';
        return value == rhs.value;
    }
    bool operator < (const MyInt& rhs) const {
        std::cout << "<  " << '\n';
        return value < rhs.value;
    }

    auto operator<=>(const MyInt& rhs) const = default;

 private:
     int value;
};

int main() {

    MyInt myInt2011(2011);
    MyInt myInt2014(2014);

    myInt2011 == myInt2014;
    myInt2011 != myInt2014;
    myInt2011 < myInt2014;
    myInt2011 <= myInt2014;
```

```
33      myInt2011 > myInt2014;
34      myInt2011 >= myInt2014;
35
36  }
```

To see the user-defined == and < operator in action, I write a corresponding message to `std::cout`. Neither operator can be `constexpr` because `std::cout` is a run-time operation.

Let's see what happens:

```
==
==
<
```

User-defined and auto-generated operators

In this case, the compiler uses the user-defined == (lines 29 and 30) and < operators (line 31). Additionally, the compiler synthesizes the `!=` operator (line 30) from the == operator. On the other hand, the compiler does not synthesize the == operator out of the `!=` operator.

Similarity to Python

In Python 3, the compiler generates `!=` out of == if necessary but not the other way around. In Python 2, the so-called rich comparison (the user-defined six comparison operators) has a higher priority than Python's three-way comparison operator `__cmp__`. I have to say Python 2 because the three-way comparison operator `__cmp__` was removed in Python 3.

Distilled Information

- By defaulting the `operator ==`, the compiler autogenerates the equality and the inequality operator: ==, and `!=`.
- By defaulting the `operator <=>`, the compiler autogenerates the six comparison operators: ==, `!=`, <, `<=`, >, and `>=`.
- The compiler-generated comparison operators are `noexcept` and `constexpr`. They apply lexicographical comparison: all base classes are compared left to right, and all non-static members of the class in their declaration order.
- When auto-generated comparison operators and user-defined comparison operators are present, the user-defined comparison operators have a higher priority.
- The compiler rewrites expressions to take care of the symmetry of the comparison operators. For example if `(a <=> b) < 0` does not work, the compiler generates `0 < (b <=> a)`.

4.4 Designated Initialization

Cippi receives the divine touch

Designated initialization is a special case of aggregate initialization. Writing about designated initialization therefore means writing about aggregate initialization.

4.4.1 Aggregate Initialization

First: what is an aggregate? Aggregates are arrays or class types. A class type is a class, a struct, or a union.

With C++20, the following condition must hold for class types being aggregates and supporting, therefore, aggregate initialization:

- No private or protected non-static data members
- No user-declared or inherited constructors
- No virtual, private, or protected base classes
- No virtual member functions

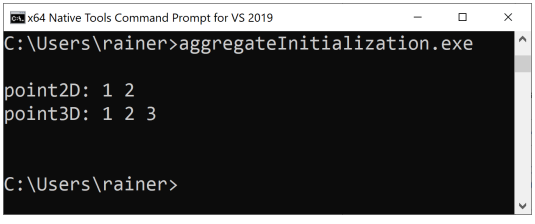
The following program exemplifies aggregate initialization.

Aggregate initialization

```
1  // aggregateInitialization.cpp
2
3  #include <iostream>
4
5  struct Point2D{
6      int x;
7      int y;
8  };
9
```

```
10  class Point3D{
11  public:
12      int x;
13      int y;
14      int z;
15  };
16
17  int main(){
18
19      std::cout << '\n';
20
21      Point2D point2D{1, 2};
22      Point3D point3D{1, 2, 3};
23
24      std::cout << "point2D: " << point2D.x << " " << point2D.y << '\n';
25      std::cout << "point3D: " << point3D.x << " " << point3D.y << " "
26                              << point3D.z << '\n';
27
28      std::cout << '\n';
29
30  }
```

Lines 21 and 22 directly initialize the aggregates using curly braces. The sequence of the initializers in the curly braces has to match the declaration order of the members. The setion on the three-way comparison operator has a more sophisticated example of aggregate initialization.



Aggregate initialization

Based on aggregate initialization in C++11, we get designed initializers in C++20.

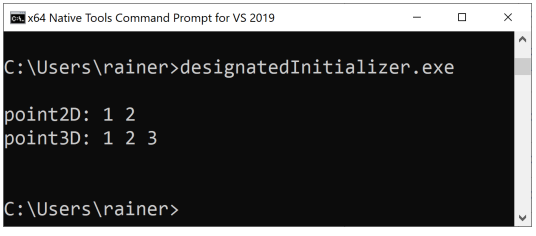
4.4.2 Named Initialization of Class Members

Designated initialization enables the direct initialization of members of a class type using their names. For a union, only one initializer can be provided. As for aggregate initialization, the sequence of initializers in the curly braces has to match the declaration order of the members.

Designated initialization

```
1  // designatedInitializer.cpp
2
3  #include <iostream>
4
5  struct Point2D{
6      int x;
7      int y;
8  };
9
10 class Point3D{
11 public:
12     int x;
13     int y;
14     int z;
15 };
16
17 int main(){
18
19     std::cout << '\n';
20
21     Point2D point2D{.x = 1, .y = 2};
22     Point3D point3D{.x = 1, .y = 2, .z = 3};
23
24     std::cout << "point2D: " << point2D.x << " " << point2D.y << '\n';
25     std::cout << "point3D: " << point3D.x << " " << point3D.y << " "
26                              << point3D.z << '\n';
27
28     std::cout << '\n';
29
30 }
```

Lines 21 and 22 use designated initializers to initialize the aggregates. The initializers as `.x` or `.y` are often called designators.



Designated Initializers

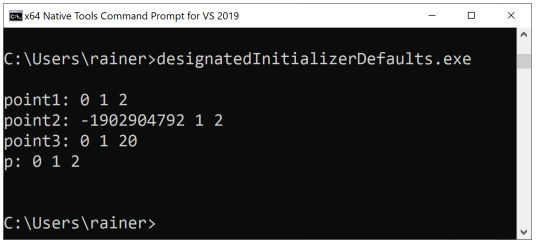
The members of the aggregate can already have a default value. This default value is used when the initializer is missing. This does not hold for a union.

Designated initializers with defaults

```
1  // designatedInitializersDefaults.cpp
2
3  #include <iostream>
4
5  class Point3D{
6  public:
7      int x;
8      int y = 1;
9      int z = 2;
10 };
11
12 void needPoint(Point3D p) {
13      std::cout << "p: " << p.x << " " << p.y << " " << p.z << '\n';
14 }
15
16 int main(){
17
18     std::cout << '\n';
19
20     Point3D point1{.x = 0, .y = 1, .z = 2};
21     std::cout << "point1: " << point1.x << " " << point1.y << " "
22                             << point1.z << '\n';
23
24     Point3D point2;
25     std::cout << "point2: " << point2.x << " " << point2.y << " "
26                             << point2.z << '\n';
27
28     Point3D point3{.x = 0, .z = 20};
29     std::cout << "point3: " << point3.x << " " << point3.y << " "
30                             << point3.z << '\n';
31
32     // Point3D point4{.z = 20, .y = 1}; ERROR
33
34     needPoint({.x = 0});
35
36     std::cout << '\n';
37
38 }
```

Line 20 initializes all members, but line 24 does not provide a value for the member `x`. Consequently,

x is not initialized. It is fine, if you only initialize the members that don't have a default value, such as in line 28 or 34. The expression in line 32 would not compile because z and y are in the wrong order.



Designated initializers with defaults

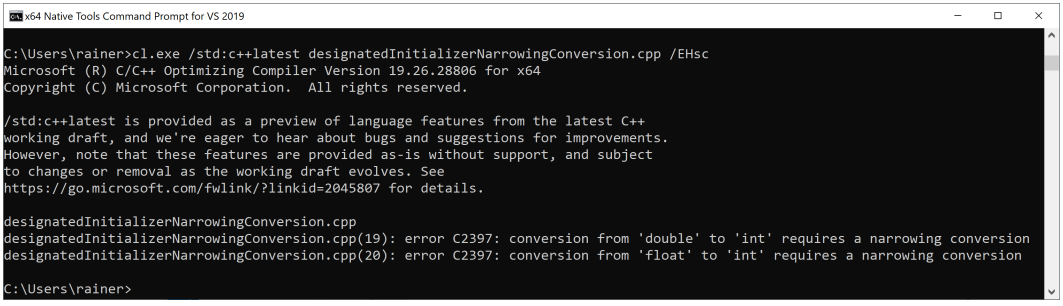
Designated initializers detect narrowing conversions. Narrowing conversion results in the loos of precision.

Designated initializers detect narrowing conversion

```
1  // designatedInitializerNarrowingConversion.cpp
2
3  #include <iostream>
4
5  struct Point2D{
6      int x;
7      int y;
8  };
9
10 class Point3D{
11 public:
12     int x;
13     int y;
14     int z;
15 };
16
17 int main(){
18
19     std::cout << '\n';
20
21     Point2D point2D{.x = 1, .y = 2.5};
22     Point3D point3D{.x = 1, .y = 2, .z = 3.5f};
23
24     std::cout << "point2D: " << point2D.x << " " << point2D.y << '\n';
25     std::cout << "point3D: " << point3D.x << " " << point3D.y << " "
26                              << point3D.z << '\n';
27
28     std::cout << '\n';
```

```
29
30  }
```

Line 21 and 22 produce compile-time errors, because the initialization `.y = 2.5` and `.z = 3.5f` would cause narrowing conversion to `int`.



Designated initializers detect narrowing conversion

Interestingly, designated initializers in C behave differently from designated initializers in C++.

Differences Between C and C++

C designated initializers support use cases that are not supported in C++. C allows

- initializing the members of the aggregate out-of-order
- initializing the members of a nested aggregate
- mixing designated initializers and regular initializers
- designated initialization of arrays

The proposal P0329R4[63] provides self-explanatory examples for these use cases:

Difference between C and C++

```
struct A { int x, y; };
struct B { struct A a; };
struct A a = {.y = 1, .x = 2}; // valid C, invalid C++ (out of order)
int arr[3] = {[1] = 5};        // valid C, invalid C++ (array)
struct B b = {.a.x = 0};       // valid C, invalid C++ (nested)
struct A a = {.x = 1, 2};      // valid C, invalid C++ (mixed)
```

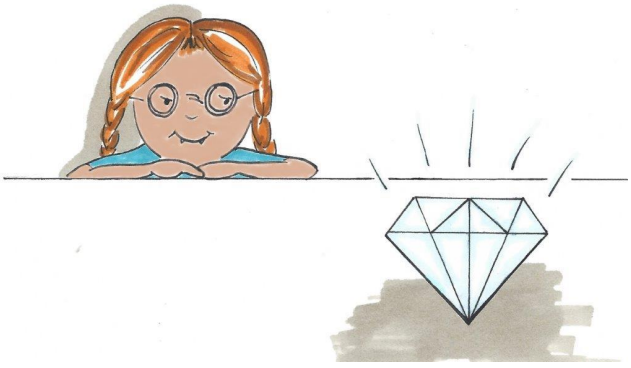
The rationale for this difference between C and C++ is also part of the proposal: "*In C++, members are destroyed in reverse construction order and the elements of an initializer list are evaluated in lexical order, so field initializers must be specified in order. Array designators conflict with lambda-expression syntax. Nested designators are seldom used.*" The paper continues to argue that only out-of-order initialization of an aggregate is commonly used.

[63] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0329r4.pdf

Distilled Information

- Designated initialization is a special case of aggregate initialization enabling it to initialize the class members using their names. The initialization order must match the declaration order.

4.5 consteval and constinit



Cippi admires the diamond

With C++20, we get two new keywords: `consteval` and `constinit`. Keyword `consteval` produces a function that is executed at compile time, and `constinit` guarantees that a variable with static storage duration or thread storage duration is initialized at compile time. Now, you may have the impression that both specifiers are quite similar to `constexpr`. To make it short, you are right. Before I compare the keywords `consteval`, `constinit`, `constexpr`, and good old `const`, I have to introduce the new specifiers `consteval` and `constinit`.

4.5.1 consteval

`consteval` creates a so-called immediate function.

A consteval function

```
consteval int sqr(int n) {
    return n * n;
}
```

Each invocation of an immediate function creates a compile-time constant. To say it more directly, a `consteval` (immediate) function is executed at compile time.

`consteval` cannot be applied to destructors or functions that allocate or deallocate. You can only use at most one of `consteval`, `constexpr`, or `constinit` specifier in a declaration. An immediate function (`consteval`) is implicitly inline and has to fulfill the requirements for a `constexpr` function.

The requirements of a `constexpr` function in C++14 and, therefore, a `consteval` function:

- A `consteval` (`constexpr`) can

  - have conditional jump instructions or loop instructions.
  - have more than one instruction.
  - invoke `constexpr` functions. A `consteval` function can only invoke a `constexpr` function but not the other way around.
  - use fundamental data types as variables that have to be initialized with a constant expression.

- A `consteval` (`constexpr`) function cannot
  - have `static` or `thread_local` data.
  - have a `try` block nor a `goto` instruction.
  - invoke or use non-`consteval` functions or non-`constexpr` data.

To make it short: all dependencies of a `consteval` function must be resolved at compile time.

The program `constevalSqr.cpp` applies the `consteval` function `sqr`.

A consteval function

```
1  // constevalSqr.cpp
2
3  #include <iostream>
4
5  consteval int sqr(int n) {
6      return n * n;
7  }
8
9  int main() {
10
11     std::cout << "sqr(5): " << sqr(5) << '\n';
12
13     const int a = 5;
14     std::cout << "sqr(a): " << sqr(a) << '\n';
15
16     int b = 5;
17     // std::cout << "sqr(b): " << sqr(b) << '\n'; ERROR
18
19 }
```

The number `5` is a constant expression and can be used as an argument for the function `sqr` (line 11). The same holds for the variable `a` (line 13). A constant variable such as `a` is usable in a constant expression when it is initialized with a constant expression. The variable `b` (line 16) is not a constant expression. Consequently, the invocation of `sqr(b)` (line 17) is not valid.

Here is the output of the program:

```
sqr(5): 25
sqr(a): 25
```

Use of a `consteval` function

Interestingly, you can have run-time functionality in your `consteval` function, but performing this run-time functionality gives a compile-time error. The following `consteval` function uses `std::cerr` that can only be performed at run time.

A consteval function using `std::cerr`

```
1  // constevalRuntime.cpp
2
3  #include <iostream>
4
5  consteval void validMonth(int n) {
6      if (n < 0 || n > 12) {
7          std::cerr << "Compile-time error if executed";
8      }
9  }
10
11 int main() {
12
13     validMonth(5);
14     validMonth(15);
15
16 }
```

Invoking `validMonth(5)` (line 13) is fine, but invoking `validMonth(15)` (line 14) gives a compile-time error because it causes the execution of the run-time function `std::cerr` (line 7).

```
<source>: In function 'int main()':
<source>:14:15:   in 'constexpr' expansion of 'validMonth(15)'
<source>:7:22: error: call to non-'constexpr' function 'std::basic_ostream<char, _Traits>&
std::operator<<(basic_ostream<char, _Traits>&, const char*) [with _Traits = char_traits<char>]'
    7 |         std::cerr << "Compile-time error if executed";
      |                      ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
In file included from /opt/compiler-explorer/gcc-13.2.0/include/c++/13.2.0/iostream:41,
                 from <source>:3:
/opt/compiler-explorer/gcc-13.2.0/include/c++/13.2.0/ostream:662:5: note: 'std::basic_ostream<char,
_Traits>& std::operator<<(basic_ostream<char, _Traits>&, const char*) [with _Traits =
char_traits<char>]' declared here
  662 |     operator<<(basic_ostream<char, _Traits>& __out, const char* __s)
      |     ^~~~~~~~
Compiler returned: 1
```

Use of `std::cerr` in a `consteval` function

4.5.2 constinit

constinit can be applied to variables with static storage duration or thread storage duration.

- Global (namespace) variables, static variables, or static class members have **static storage duration**. These objects are allocated when the program starts and are deallocated when the program ends.
- thread_local variables have **thread storage duration**. Thread-local data is created for each thread that uses this data. thread_local data exclusively belongs to the thread. They are created at their first usage and their lifetime is bound to the lifetime of the thread it belongs to. Often thread-local data is called thread-local storage.

constinit ensures that this kind of variable (static storage duration or thread storage duration) it is initialized at compile time. constinit does not imply constness. This has two interesting consequencies: A constinit variable requires a constant compile time value and cannot initialize another constinit variable.

Initialization with constinit

```
// constinitSqr.cpp

#include <iostream>

consteval int sqr(int n) {
    return n * n;
}

constexpr auto res1 = sqr(5);
constinit auto res2 = sqr(5);

int main() {

    std::cout << "sqr(5): " << res1 << '\n';
    std::cout << "sqr(5): " << res2 << '\n';

    constinit thread_local auto res3 = sqr(5);
    std::cout << "sqr(5): " << res3 << '\n';

}
```

res1 and res2 have static storage duration. res3 has thread storage duration.

```
sqr(5): 25
sqr(5): 25
sqr(5): 25
```

Use of `constinit` initialization

4.5.3 Comparison of `const`, `constexpr`, `consteval`, and `constinit`

Now it's time to write about the differences between `const`, `constexpr`, `consteval`, and `constinit`. First, I discuss function execution and then variable initialization.

4.5.3.1 Function Execution

The following program `consteval.cpp` has three versions of a `square` function.

Three versions of a square function

```
// consteval.cpp

#include <iostream>

int sqrRunTime(int n) {
    return n * n;
}

consteval int sqrCompileTime(int n) {
    return n * n;
}

constexpr int sqrRunOrCompileTime(int n) {
    return n * n;
}

int main() {

    // constexpr int prod1 = sqrRunTime(100); ERROR
    constexpr int prod2 = sqrCompileTime(100);
    constexpr int prod3 = sqrRunOrCompileTime(100);

    int x = 100;

    int prod4 = sqrRunTime(x);
    // int prod5 = sqrCompileTime(x); ERROR
    int prod6 = sqrRunOrCompileTime(x);
```

```
28
29  }
```

As the name suggests: the ordinary function `sqrRunTime` (line 5) runs at run time, the `consteval` function `sqrCompileTime` runs at compile time (line 9) the `constexpr` function `sqrRunOrCompileTime` can run at compile time or run time. Consequently, asking for the result at compile time with `sqrRunTime` (line 19) is an error, so, using a non-constant expression as an argument for `sqrCompileTime` (line 26) is also an error.

The difference between the `constexpr` function `sqrRunOrCompileTime` and the `consteval` function `sqrCompileTime` is that `sqrRunOrCompileTime` must be executed at compile time when the context requires compile-time evaluation.

Compile-time and run-time execution

```
1   static_assert(sqrRunOrCompileTime(10) == 100);                    // compile time
2   int arrayNewWithConstExpressiomFunction[sqrRunOrCompileTime(100)]; // compile time
3   constexpr int prod = sqrRunOrCompileTime(100);                    // compile time
4
5   int a = 100;
6   int runTime = sqrRunOrCompileTime(a);                  // run time
7
8   int runTimeOrCompiletime = sqrRunOrCompileTime(100);  // run time or compile time
9
10  int alwaysCompileTime = sqrCompileTime(100);          // compile time
```

Lines 1 - 3 require compile-time evaluation. Line 6 can only be evaluated at run time because `a` is not a constant expression. The critical line is line 8. The function can be executed at compile time or run time. Whether it is executed at compile time or run time may depend on the compiler or on the optimization level. This observation does not hold for line 10. A `consteval` function is always executed at compile time.

4.5.3.2 Variable Initialization

The program `constexprConstinit.cpp` compares `const`, `constexpr`, and `constinit`.

Comparison of `const`, `constexpr`, and `constinit`

```
1  // constexprConstinit.cpp
2
3  #include <iostream>
4
5  constexpr int constexprVal = 1000;
6  constinit int constinitVal = 1000;
7
8  int incrementMe(int val){ return ++val;}
9
10 int main() {
11
12     auto val = 1000;
13     const auto res = incrementMe(val);
14     std::cout << "res: " << res << '\n';
15
16     // std::cout << "res: " << ++res << '\n';                    ERROR
17     // std::cout << "++constexprVal: " << ++constexprVal << '\n'; ERROR
18     std::cout << "++constinitVal: " << ++constinitVal << '\n';
19
20     constexpr auto localConstexpr = 1000;
21     // constinit auto localConstinit = 1000; ERROR
22
23 }
```

Only the `const` variable (line 13) is initialized at run time. The `constexpr` and `constinit` variables are initialized at compile time.

The `constinit` (line 18) does not imply constness, as do `const` (line 16) or `constexpr` (line 17). A `constexpr` (line 20) or `const` (line 13) declared variable can be created as a local, but not a `constinit` declared variable (line 21).

```
res: 1001
++constinitVal: 1001
```

`const`, `constexpr`, and `constinit` declared variables

Initialization of a Local Non-Const Variable at Compile Time

After the previous program `constexprConstinit.cpp`, you may have the impression that you cannot initialize a local (automatic storage duration) non-const variable at compile time:

- `constinit` enables the initialization at compile time only for objects with static storage duration.
- `constexpr` implies that the variable is constant.

Thanks to `consteval`, a local having automatic storage duration can be initialized at compile time at modified afterward.

```
1  // compileTimeInitializationLocal.cpp
2
3  consteval auto doubleMe(auto val) {
4     return 2 * val;
5  }
6
7  int main() {
8
9     auto res = doubleMe(1010);
10    ++res;           // 2021
11
12 }
```

The local `res` is initialized at compile time (line 9) and modified at run time (line 10).

4.5.4 Solving the Static Initialization Order Fiasco

According to the FAQ at isocpp.org[64], the static initialization order fiasco is "*a subtle way to crash your program*". The FAQ continues: "*The static initialization order problem is a very subtle and commonly misunderstood aspect of C++.*"

Before I continue, I want to make a short disclaimer. Dependencies on variables with static storage duration (short statics) in different translation units are, in general, a code smell and should be a reason for refactoring. Consequently, if you follow my advice to refactor, you can skip this section.

4.5.4.1 Static Initialization Order Fiasco

Static variables in one translation unit are initialized according to their definition order.

In contrast, the initialization of static variables between translation units has a severe issue. When one static variable `staticA` is defined in one translation unit and another static variable `staticB` is

[64] https://isocpp.org/wiki/faq/ctors#static-init-order

defined in another translation unit, and `staticB` needs `staticA` to initialize itself, you end up with the static initialization order fiasco. The program is ill-formed because you have no guarantee which static variable is initialized first at (dynamic) run time.

Before I write about the solution, let me show you the static initialization order fiasco in action.

4.5.4.1.1 A 50:50 Chance to get it Right

What is unique about the initialization of statics? The initialization-order of statics happens in two steps: static and dynamic.

When a static cannot be const-initialized during compile time, it is zero-initialized. At run time, the dynamic initialization happens for these statics that was zero-initialized.

The static initialization order fiasco

```
// sourceSIOF1.cpp

int square(int n) {
    return n * n;
}

auto staticA  = square(5);
```

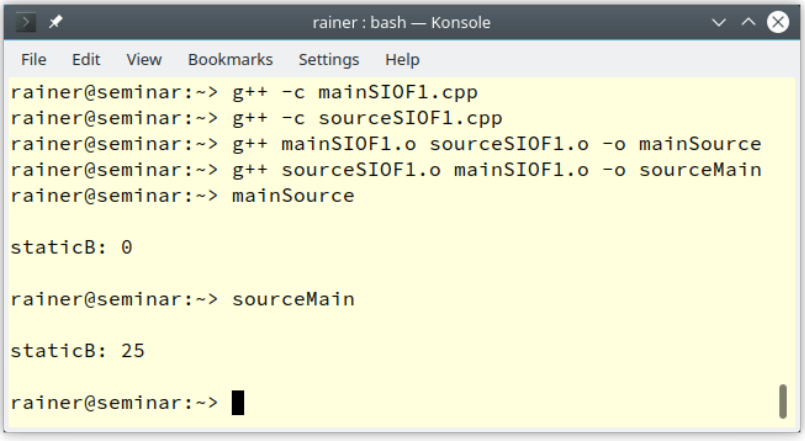
The static initialization order fiasco

```
1  // mainSOIF1.cpp
2
3  #include <iostream>
4
5  extern int staticA;
6  auto staticB = staticA;
7
8  int main() {
9
10     std::cout << '\n';
11
12     std::cout << "staticB: " << staticB << '\n';
13
14     std::cout << '\n';
15
16 }
```

Line 5 declares the static variable `staticA`. The initialization of `staticB` depends on the initialization of `staticA`. But `staticB` is zero-initialized at compile time and dynamically initialized at run time. The

issue is that there is no guarantee in which order `staticA` or `staticB` are initialized because `staticA` and `staticB` belong to different translation units. You have a 50:50 chance that `staticB` is 0 or 25.

To demonstrate this problem, I can change the link order of the object files. This also changes the value for `staticB`!



The static initializaion order fiasco caught in action

What a fiasco! The result of the executable depends on the link order of the object files. What can we do when we don't have C++20 at our disposal?

4.5.4.1.2 Lazy initialization of a `static` with a Local Scope

Static variables with local scope are created when they are used the first time. Local scope essentially means that the static variable is surrounded in some way by curly braces. This lazy creation is a guarantee that C++98 provides. With C++11, static variables with local scope are also initialized in a thread-safe way. The thread-safe Meyers[65] singleton is based on this additional guarantee.

The lazy initialization can also be used to overcome the static initialization order fiasco.

[65] https://en.wikipedia.org/wiki/Scott_Meyers

Lazy initialization of a static with local scope

```
// sourceSIOF2.cpp

int square(int n) {
    return n * n;
}

int& staticA() {

    static auto staticA  = square(5);
    return staticA;

}
```

Lazy initialization of a static with local scope

```
// mainSOIF2.cpp

#include <iostream>

int& staticA();

auto staticB = staticA();

int main() {

    std::cout << '\n';

    std::cout << "staticB: " << staticB << '\n';

    std::cout << '\n';

}
```

`staticA` (line 9 in file `sourceSIOF2.cpp`) is, in this case, a static in a local scope. Line 5 in file `mainSOIF2.cpp` declares the function `staticA`, which is used to initialize in the following line `staticB`. This local scope of `staticA` guarantees that `staticA` is created and initialized during run time when it is the first time used. Changing the link order can, in this case, not change the value of `staticB`.

Solving the static initialization order fiasco with local statics

In the last step, I solve the static initialization order fiasco using C++20.

4.5.4.1.3 Compile-Time Initialization of a static

Let me apply `constinit` to `staticA`. The `constinit` guarantees that `staticA` is initialized during compile time.

Compile-time initialization of a static

```
// sourceSIOF3.cpp

constexpr int square(int n) {
    return n * n;
}

constinit auto staticA = square(5);
```

Compile-time initialization of a static

```
// mainSOIF3.cpp

#include <iostream>

extern constinit int staticA;

auto staticB = staticA;

int main() {
```

```
10
11      std::cout << '\n';
12
13      std::cout << "staticB: " << staticB << '\n';
14
15      std::cout << '\n';
16
17  }
```

Line 5 in file `mainSOIF3.cpp` declares the variable `staticA`, which is initialized (line 7 in file `sourceSIOF3.cpp`) at compile time. By the way, using `constexpr` (line 5 in file `mainSOIF3.cpp`) instead of `constinit` would not be valid, because `constexpr` requires a definition and not just a declaration.

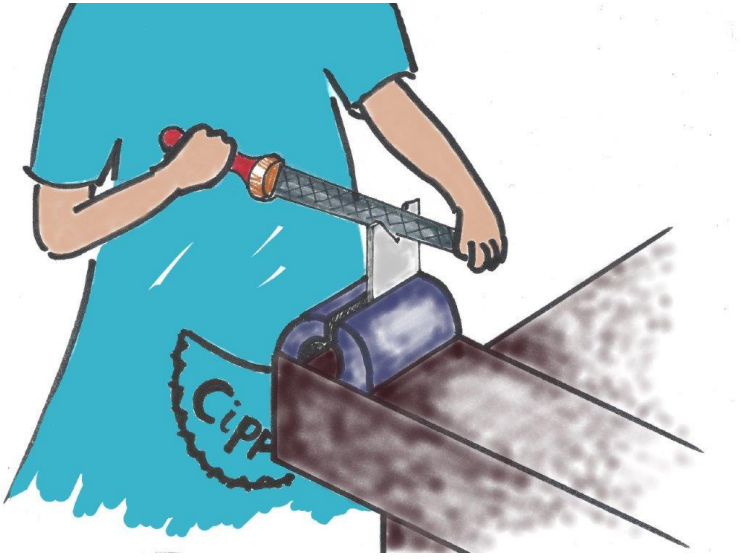
Solving the static initializaion order fiasco with `constinit`

As in the case of the lazy initialization with a local static, `staticB` has the value 25.

Distilled Information

- With C++20, we get two new keywords: `consteval` and `constinit`. `consteval` produces a function that is executed at compile time, and `constinit` guarantees that the variable is initialized at compile time.
- In contrast to `constexpr` in C++11, `consteval` guarantees that the function is executed at compile time.
- There are subtle differences between `const`, `constexpr`, and `constinit`. `const` and `constexpr` create constant variables. `constexpr` and `constinit` are executed at compile time.

4.6 Template Improvements



Cippi uses her new tools

The improvements to templates make C++20 more consistent and, therefore, less error-prone when writing generic programs.

4.6.1 Conditionally Explicit Constructor

Sometimes you need a class that should have constructors accepting different types. For example, you have a class `VariantWrapper`, that holds a `std::variant` accepting various types.

A class VariantWrapper holding an attribute std::variant

```
class VariantWrapper {

    std::variant<bool, char, int, double, float, std::string> myVariant;

};
```

To initialize a `VariantWrapper` with `bool`, `char`, `int`, `double`, `float`, or `std::string`, the class `VariantWrapper` needs constructors for each listed type. Laziness is a virtue – at least for programmers – , therefore, you decide to make the constructor generic.

The class `Implicit` shows a generic constructor.

A generic constructor

```
1  // implicitExplicitGenericConstructor.cpp
2
3  #include <iostream>
4  #include <string>
5
6  struct Implicit {
7      template <typename T>
8      Implicit(T t) {
9          std::cout << t << '\n';
10     }
11 };
12
13 struct Explicit {
14     template <typename T>
15     explicit Explicit(T t) {
16         std::cout << t << '\n';
17     }
18 };
19
20 int main() {
21
22     std::cout << '\n';
23
24     Implicit imp1 = "implicit";
25     Implicit imp2("explicit");
26     Implicit imp3 = 1998;
27     Implicit imp4(1998);
28
29     std::cout << '\n';
30
31     // Explicit exp1 = "implicit";
32     Explicit exp2{"explicit"};
33     // Explicit exp3 = 2011;
34     Explicit exp4{2011};
35
36     std::cout << '\n';
37
38 }
```

Now, you have an issue. A generic constructor (line 7) is a catch-all constructor because you can invoke it with any type. The constructor is way too greedy. By putting an `explicit` in front of the constructor (line 14), implicit conversions (lines 31 and 33) are not valid anymore. Only the explicit calls (lines 32 and 34) are valid.

```
implicit
explicit
1998
1998

explicit
2011
```

Implicit and explicit generic constructors

In C++20, `explicit` is even more useful. Imagine you have a type `MyBool` that should only support the implicit conversion from bool, but no other implicit conversion. In this case, `explicit` can be used conditionally.

A generic constructor that allows implicit conversions from bool

```
// conditionallyConstructor.cpp

#include <iostream>
#include <type_traits>
#include <typeinfo>

struct MyBool {
    template <typename T>
    explicit(!std::is_same<T, bool>::value) MyBool(T t) {
        std::cout << typeid(t).name() << '\n';
    }
};

void needBool(MyBool b){ }

int main() {

    MyBool myBool1(true);
    MyBool myBool2 = false;

    needBool(myBool1);
    needBool(true);
    // needBool(5);
    // needBool("true");

}
```

The `explicit(!std::is_same<T, bool>::value)` expression guarantees that `MyBool` can only be implicitly created from a `bool` value. The function `std::is_same` is a compile-time predicate from

the type_traits library[66]. A compile-time predicate, such as `std::is_same` is evaluated at compile time and returns a boolean. Consequently, the implicit conversions from `bool` (lines 19 and 22) are possible, but not the commented-out conversions from `int` and `C-string` (lines 23 and 24).

4.6.2 Non-Type Template Parameters (NTTP)

C++ supports non-types as template parameters. Essentially non-types could be

- integers and enumerators
- pointers to objects, to functions and to attributes of a class
- lvalue references
- `std::nullptr_t`

Typical Non-Type Template Parameter

When I ask the students in my class if they ever used a non-type as template parameter they say: No! Of course, I answer my tricky question and show an often-used example for non-type template parameters:

Defining a std::array

```
std::array<int, 5> myVec;
```

Constant 5 is a non-type used as a template argument.

Since the first C++-standard C++98, there has been an ongoing discussion in the C++ community about supporting floating-point template parameters. Now, we have them and more: C++20 supports floating-points, literal types, and string literals as non-types.

4.6.2.1 Floating-Point Typs

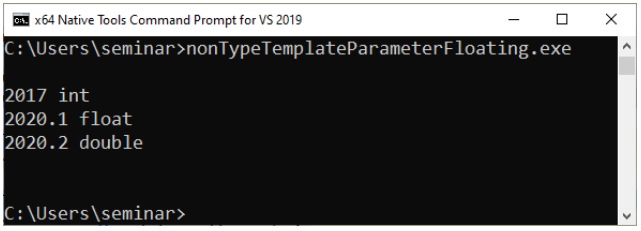
The following program uses floating-point types as non-type template parameters.

[66] https://en.cppreference.com/w/cpp/header/type_traits

Floating-point types as non-type template parameters

```
1  // nonTypeTemplateParameterFloating.cpp
2
3  #include <iostream>
4  #include <typeinfo>
5
6  template <double d>
7  auto getDouble() {
8      return d;
9  }
10
11 template <auto NonType>
12 auto getNonType() {
13     return NonType;
14 }
15
16 int main() {
17
18     std::cout << '\n';
19
20     auto d1 = getDouble<5.5>();
21     auto d2 = getDouble<6.5>();
22
23     auto i = getNonType<2017>();
24     std::cout << i << " " << typeid(i).name() << '\n';
25
26     auto f = getNonType<2020.1f>();
27     std::cout << f << " " << typeid(f).name() << '\n';
28
29     auto d = getNonType<2020.2>();
30     std::cout << d << " " << typeid(d).name() << '\n';
31
32     std::cout << '\n';
33
34 }
```

The function template `getDouble` (line 6) only accepts `double` values. I want to emphasize that each call of the function template `getDouble` (lines 21 and 22) creates a new function `getDouble`. This function is a full specialization for the given `double` value. Since C++17, you can use a `auto` as non-type template parameter. Consequently, line 23 is valid with C++17. With C++20, you can also use `auto` for floating-point types. The following program visualizes the type deduction of the C++20 compiler. The compiler deduces the type `int` (line 24), `float` (line 27), and `double` (line 30) for the non-type template parameter.



Floating-point types as non-type template parameters

4.6.2.2 Literal Typs

Literal Types with the following two properties:

- all base classes and non-static data members are public and non-mutable
- the types of all base classes and non-static data members are structural types or arrays of these

A literal type must have a `constexpr` constructor.

Literal types as non-type template parameters

```
// nonTypeTemplateParameterLiteral.cpp

struct ClassType {
    constexpr ClassType(int) {}
};

template <ClassType c1>
auto getClassType() {
    return c1;
}

int main() {

    auto c1 = getClassType<ClassType(2020)>();

}
```

Since C++20, strings can be used as non-type template arguments.

4.6.2.3 String Literals

The class `StringLiteral` has a `constexpr` constructor.

String literals as non-type template parameters

```
1  // nonTypeTemplateParameterString.cpp
2
3  #include <algorithm>
4  #include <iostream>
5
6  template <int N>
7  class StringLiteral {
8   public:
9      constexpr StringLiteral(char const (&str)[N]) {
10         std::copy(str, str + N, data);
11     }
12     char data[N];
13 };
14
15 template <StringLiteral str>
16 class ClassTemplate {};
17
18 template <StringLiteral str>
19 void FunctionTemplate() {
20     std::cout << str.data << '\n';
21 }
22
23 int main() {
24
25     std::cout << '\n';
26
27     ClassTemplate<"string literal"> cls;
28     FunctionTemplate<"string literal">();
29
30     std::cout << '\n';
31
32 }
```

`StringLiteral` is a literal type and, therefore, can be used as non-type template parameter for `ClassTemplate` (line 15) and `FunctionTemplate` (line 18). The `constexpr` constructor (line 9) takes a C-string as an argument.

```
string literal
```

String literals as non-type template parameters

You may wonder why we need string literals as non-type template parameter?

Compile-Time Regular Expressions

A very impressive use-case for string literals is compile-time parsing of regular expressions[67]. There is already a proposal for C++23 in the pipeline: P1433R0: Compile-Time Regular Expressions[68]. Hana Dusíková as the author of the proposal motivates compile-time regular expressions in C++: "*The current `std::regex` design and implementation [regular expression library[69]] are slow, mostly because the RE [regular expression] pattern is parsed and compiled at run time. Users often don't need a runtime RE [regular expression] parser engine as the pattern is known during compilation in many common use cases. I think this breaks C++'s promise of 'don't pay for what you don't use'.*

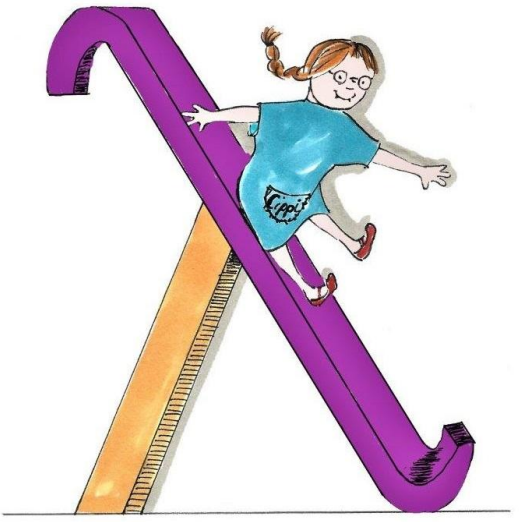
If the RE [regular expression] is known at compile time, the pattern should be checked during the compilation. The design of `std::regex` doesn't allow for this[compile-time evaluation,] as the RE input is a run-time string and syntax errors are reported as exceptions.".

Distilled Information

- A conditionally explicit constructor allows it to control explicitly for a generic constructor which types can be used in a constructor.
- C++20 supports further floating-point types, literal types, and string literals as non-type template parameters.

[67] https://github.com/hanickadot/compile-time-regular-expressions
[68] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1433r0.pdf
[69] https://en.cppreference.com/w/cpp/regex

4.7 Lambda Improvements



Cippi slides down the slide

With C++20, lambda expressions support template parameters and hence concepts can be default-constructed and support copy assignment when they have no state. Furthermore, a syntactical restriction is gone: pack expansion in init-capture. Additionally, lambda expressions can be used in unevaluated contexts. With C++20, they detect when you implicitly copy the `this` pointer. That means a significant cause of undefined behavior with lambdas is gone.

Let's start with template parameters for lambdas.

4.7.1 Template Parameter for Lambdas

Admittedly, the differences between typed lambdas (C++11), generic lambdas (C++14), and template lambdas (template parameter for lambdas) in C++20 are subtle.

Typed lambdas, generic lambdas, and template lambdas

```
// templateLambda.cpp

#include <iostream>
#include <string>
#include <vector>

auto sumInt = [](int fir, int sec) { return fir + sec; };
auto sumGen = [](auto fir, auto sec) { return fir + sec; };
auto sumDec = [](auto fir, decltype(fir) sec) { return fir + sec; };
auto sumTem = []<typename T>(T fir, T sec) { return fir + sec; };

int main() {

    std::cout << '\n';

    std::cout << "sumInt(2000, 11): " << sumInt(2000, 11) << '\n';
    std::cout << "sumGen(2000, 11): " << sumGen(2000, 11) << '\n';
    std::cout << "sumDec(2000, 11): " << sumDec(2000, 11) << '\n';
    std::cout << "sumTem(2000, 11): " << sumTem(2000, 11) << '\n';

    std::cout << '\n';

    std::string hello = "Hello ";
    std::string world = "world";
    // std::cout << "sumInt(hello, world): " << sumInt(hello, world) << '\n';
    std::cout << "sumGen(hello, world): " << sumGen(hello, world) << '\n';
    std::cout << "sumDec(hello, world): " << sumDec(hello, world) << '\n';
    std::cout << "sumTem(hello, world): " << sumTem(hello, world) << '\n';


    std::cout << '\n';

    std::cout << "sumInt(true, 2010): " << sumInt(true, 2010) << '\n';
    std::cout << "sumGen(true, 2010): " << sumGen(true, 2010) << '\n';
    std::cout << "sumDec(true, 2010): " << sumDec(true, 2010) << '\n';
    // std::cout << "sumTem(true, 2010): " << sumTem(true, 2010) << '\n';

    std::cout << '\n';

}
```

Before I show the presumably astonishing output of the program, I want to compare the four lambdas.

- `sumInt`

  - C++11
  - Typed lambda
  - Accepts only types convertible to `int`
- `sumGen`
  - C++14
  - Generic lambda
  - Accepts all types
- `sumDec`
  - C++14
  - Generic lambda
  - The second type must be convertible to the first type
- `sumTem`
  - C++20
  - Template lambda
  - The first type and the second type must be identical

What does this mean for template arguments with different types? Of course, each lambda accepts `int` (lines 16 - 19), and the typed lambda `sumInt` does not accept strings (line 25).

Invoking the lambdas with the `bool true` and the `int 2010` may be surprising (lines 33 - 36).

- `sumInt` returns 2011 because `true` is an integral, promoted to `int`.
- `sumGen` returns 2011 because true is an integral, promoted to `int`. There is a subtle difference between `sumInt` and `sumGen`, which I will present in a few lines.
- `sumDec` returns `2`. Why? The type of the second parameter `sec` becomes the type of the first parameter `fir`: thanks to `decltype(fir) sec`, the compiler deduces the type of `fir` and makes it the type of `sec`. Consequently, 2010 is converted to `true`. In the expression `fir + sec`, `fir` is integral promoted to `1`. Finally, the result is `2`.
- `sumTem` is not valid.

```
sumInt(2000, 11): 2011
sumGen(2000, 11): 2011
sumDec(2000, 11): 2011
sumTem(2000, 11): 2011

sumGen(hello, world): Hello world
sumDec(hello, world): Hello world
sumTem(hello, world): Hello world

sumInt(true, 2010): 2011
sumGen(true, 2010): 2011
sumDec(true, 2010): 2
```

The subtle differences between typed lambdas, generic lambdas, and template lambdas

A more typical use case for template lambdas is using of containers in lambdas. The following program presents three lambdas accepting a container. Each lambda returns the size of the container.

Three lambdas accepting a container

```
// templateLambdaVector.cpp

#include <concepts>
#include <deque>
#include <iostream>
#include <string>
#include <vector>

auto lambdaGeneric = [](const auto& container) { return container.size(); };
auto lambdaVector = []<typename T>(const std::vector<T>& vec) { return vec.size(); };
auto lambdaVectorIntegral = []<std::integral T>(const std::vector<T>& vec) {
    return vec.size();
};

int main() {


    std::cout << '\n';

    std::deque deq{1, 2, 3};
    std::vector vecDouble{1.1, 2.2, 3.3, 4.4};
    std::vector vecInt{1, 2, 3, 4, 5};

    std::cout << "lambdaGeneric(deq): " << lambdaGeneric(deq) << '\n';
    // std::cout << "lambdaVector(deq): " << lambdaVector(deq) << '\n';
    // std::cout << "lambdaVectorIntegral(deq): "
    //           << lambdaVectorIntegral(deq) << '\n';

    std::cout << '\n';

    std::cout << "lambdaGeneric(vecDouble): " << lambdaGeneric(vecDouble) << '\n';
    std::cout << "lambdaVector(vecDouble): " << lambdaVector(vecDouble) << '\n';
    // std::cout << "lambdaVectorIntegral(vecDouble): "
    //           << lambdaVectorIntegral(vecDouble) << '\n';

    std::cout << '\n';

    std::cout << "lambdaGeneric(vecInt): " << lambdaGeneric(vecInt) << '\n';
    std::cout << "lambdaVector(vecInt): " << lambdaVector(vecInt) << '\n';
    std::cout << "lambdaVectorIntegral(vecInt): "
              << lambdaVectorIntegral(vecInt) << '\n';

    std::cout << '\n';
```

```
44
45 }
```

Function `lambdaGeneric` (line 9) can be invoked with any data type having a member function `size()`. Function `lambdaVector` (line 10) is more specific: it only accepts a `std::vector`. Function `lambdaVectorIntegral` (line 11) uses the C++20 concept `std::integral`. Consequently, it only accepts a `std::vector` using integral types such as `int`. To use the concept `std::integral`, I have to include the header `<concepts>`. I assume the small program is self-explanatory.

```
lambdaGeneric(deq): 3

lambdaGeneric(vecDouble): 4
lambdaVector(vecDouble): 4

lambdaGeneric(vecInt): 5
lambdaVector(vecInt): 5
lambdaVectorIntegral(vecInt): 5
```

Lambdas, accepting a container and a std::vector

Class Template Argument Deduction

There is one feature in the program `templateLambdaVector.cpp` that you have probably missed. Since C++17, the compiler can deduce the type of a class template from its arguments (lines 20 - 22). Consequently, instead of the verbose `std::vector<int> myVec{1, 2, 3}`, you can simply write `std::vector myVec{1, 2, 3}`.

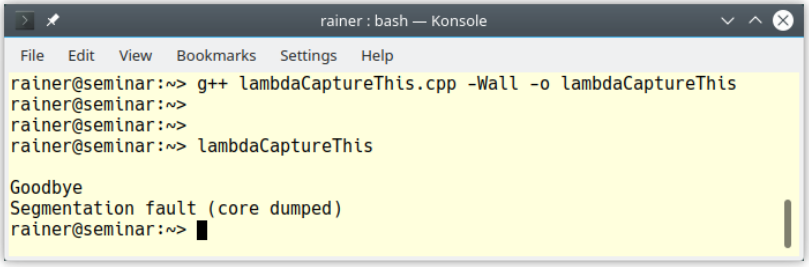
4.7.2 Detection of the Implicit Copy of the `this` Pointer

The C++20 compiler detects when you implicitly copy the `this` pointer. Implicitly capturing the `this` pointer by copy can cause undefined behavior. Undefined behavior essentially means that there are no guarantees about the program's behavior, such as for the following:

Implicitly capturing the `this` pointer by copy

```
// lambdaCaptureThis.cpp

#include <iostream>
#include <string>

struct LambdaFactory {
    auto foo() const {
        return [=] { std::cout << s << '\n'; };
    }
    std::string s = "LambdaFactory";
     ~LambdaFactory() {
        std::cout << "Goodbye" << '\n';
    }
};

auto makeLambda() {
    LambdaFactory lambdaFactory;                                                    \

    return lambdaFactory.foo();
}


int main() {

    std::cout << '\n';

    auto lam = makeLambda();
    lam();

    std::cout << '\n';

}
```

The compilation of the program works as expected, but this does not hold for the execution of the program.



Segmentation fault due to undefined behavior

Do you spot the issue in the program `lambdaCaptureThis.cpp`? The member function `foo` (line 7) returns the `lambda [=] { std::cout << s << '\n'; }` having an implicit copy of the `this` pointer. This implicit copy is no issue in (line 17), but it becomes an issue with the end of the scope. The end of the scope means the end of the lifetime of the local lambda (line 19). Consequently, the call `lam()` (line 28) triggers undefined behavior.

A C++20 compiler must, in this case, issue a warning.

```
<source>:8:16: warning: implicit capture of 'this' via '[=]' is deprecated in C++20 [-Wdeprecated]
    8 |         return [=] { std::cout << s << std::endl; };
      |                ^
<source>:8:16: note: add explicit 'this' or '*this' capture
Execution build compiler returned: 0
Program returned: 139

  Goodbye
```

C++20 diagnoses a warning

The last two lambdas features of C++20 are quite handy when you combine them: Lambdas in C++20 can be default-constructed and support copy-assignment when they have no state. Additionally, lambdas can be used in unevaluated contexts.

4.7.3 Lambdas in an Unevaluated Context and Stateless Lambdas can be Default-Constructed and Copy-Assigned

Admittedly, the title of this section contains two terms that may be new to you: unevaluated context and stateless lambda. Let me start with unevaluated context.

4.7.3.1 Unevaluated Context

The following code snippet has a function declaration and a function definition.

Declaration and definition of a function

```
int add1(int, int);                       // declaration
int add2(int a, int b) { return a + b; }  // definition
```

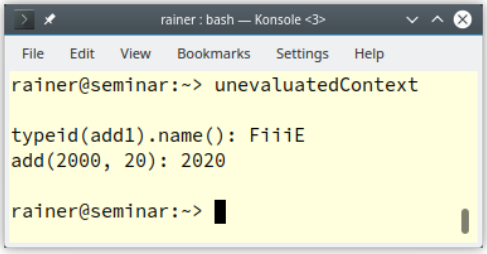
Function `add1` is declared, while `add2` is defined. That means, if you use `add1` in an evaluated context, for example, by invoking it, you get a link-time error. The key observation is that you can use `add1` in unevaluated contexts, such as `typeid`[70] or `decltype`[71]. Both operators accept unevaluated operands.

Unevaluated context

```
1  // unevaluatedContext.cpp
2
3  #include <iostream>
4  #include <typeinfo>  // typeid
5
6  int add1(int, int);                       // declaration
7  int add2(int a, int b) { return a + b; }  // definition
8
9  int main() {
10
11     std::cout << '\n';
12
13     std::cout << "typeid(add1).name(): " << typeid(add1).name() << '\n';
14
15     decltype(*add1) add = add2;
16
17     std::cout << "add(2000, 20): " << add(2000, 20) << '\n';
18
19     std::cout << '\n';
20
21 }
```

`typeid(add1).name()` (line 13) returns a string representation of the type and `decltype` (line 15) deduces the type of its argument.

[70] https://en.cppreference.com/w/cpp/language/typeid
[71] https://en.cppreference.com/w/cpp/language/decltype



Use of an unevaluated context

4.7.3.2 Stateless Lambda

A stateless lambda is a lambda that captures nothing from its environment. Or, to put it another way, a stateless lambda is a lambda where the initial brackets `[]` in the lambda definition are empty. For example, the lambda expression `auto add = [ ](int a, int b) { return a + b; };` is stateless.

4.7.3.3 Adapting Associative Containers of the Standard Template Library

Before I show you the example, I must add a few remarks. Container `std::set` and all other ordered associative containers from the Standard Template Library (`std::map`, `std::multiset`, and `std::multimap`) use the function object `std::less` to sort the keys. `std::less` sorts all keys lexicographically in ascending order by default. The declaration of `std::set`[72] shows the implicit usage of `std::less`.

Declaration of std::set

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

Now, let me play with the ordering.

[72] https://en.cppreference.com/w/cpp/container/set

Lambdas used in an unevaluated context

```cpp
// lambdaUnevaluatedContext.cpp

#include <cmath>
#include <iostream>
#include <memory>
#include <set>
#include <string>

template <typename Cont>
void printContainer(const Cont& cont) {
    for (const auto& c: cont) std::cout << c << "  ";
    std::cout << "\n";
}

int main() {

    std::cout << '\n';

    std::set<std::string> set1 = {"scott", "Bjarne", "Herb", "Dave", "michael"};
    printContainer(set1);

    using SetDecreasing = std::set<std::string,
                                   decltype([](const auto& l, const auto& r) {
                                       return l > r;
                                   })>;
    SetDecreasing set2 = {"scott", "Bjarne", "Herb", "Dave", "michael"};
    printContainer(set2);

    using SetLength = std::set<std::string,
                               decltype([](const auto& l, const auto& r) {
                                   return l.size() < r.size();
                               })>;
    SetLength set3 = {"scott", "Bjarne", "Herb", "Dave", "michael"};
    printContainer(set3);

    std::cout << '\n';

    std::set<int> set4 = {-10, 5, 3, 100, 0, -25};
    printContainer(set4);

    using setAbsolute = std::set<int, decltype([](const auto& l, const auto& r) {
                                                   return std::abs(l)< std::abs(r);
                                               })>;
    setAbsolute set5 = {-10, 5, 3, 100, 0, -25};
```

```
45      printContainer(set5);
46
47      std::cout << "\n\n";
48
49  }
```

`set1` (line 19) and `set4` (line 38) sort their keys in ascending order. Each of `set2` (line 26), `set3` (line 33), and `set5` (line 44) sorts its keys in an uniquely manner, using a lambda in an unevaluated context. The `using` keyword (line 22) declares a type alias, which is used in the following line (line 26) to define the sets. Creating the `std::set` causes the call of the default constructor of the stateless lambda.

Here is the output of the program.

```
Bjarne  Dave  Herb  michael  scott
scott  michael  Herb  Dave  Bjarne
Herb  scott  Bjarne  michael

-25  -10  0  3  5  100
0  3  5  -10  -25  100
```

Use of a lambda in an unevaluated context

When you study the output of the program, you may be surprised. The special `set3`, which uses the lambda `[](const auto& l, const auto& r){ return l.size() < r.size(); }` as a predicate, ignores the name `Dave`. The reason is simple. `Dave` has the same size as `Herb`, which was added first. `std::set` supports unique keys, and the keys in this case are identical using the special predicate. If I had used `std::multiset`, this wouldn't have happened.

4.7.4 `consteval` Lambdas

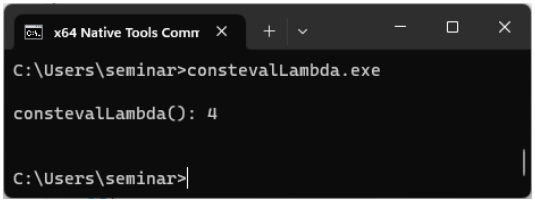
C++20 support `consteval` in C++20. `consteval` lambdas means that the lambda is an immediate function executed at compile time.

A `consteval` lambda

```
1   // constevalLambda.cpp
2
3   #include <algorithm>
4   #include <iostream>
5   #include <vector>
6
7   int main() {
8
9       std::cout <<  '\n';
10
```

```
11      auto constevalLambda = [] () consteval {
12          std::vector myVec = {1, 2, 4, 3};
13          std::sort(myVec.begin(), myVec.end()) ;
14          return myVec.back();
15      };
16
17      std::cout << "constevalLambda(): " << constevalLambda() << '\n';
18
19      std::cout << '\n';
20
21  }
```

The lambda `constevalLambda` (line 11) is declared as `consteval`. Consequentially, the lambda call (line 17) is performed at compile time.



A `consteval` lambda

Empty Parameter Clause

When you declare the lambda as `consteval`, the redundant empty parameter clause `[] () consteval` is still in C++20 required. With C++23, the restriction is gone and you can define the `consteval` lambda without the parameter clause:

A `consteval` lambda with empty parameter lause

```
auto constevalLambda = [] consteval {
   std::vector myVec = {1, 2, 4, 3};
   std::sort(myVec.begin(), myVec.end()) ;
   return myVec.back();
};
```

4.7.5 Pack Expansion in Init-Capture

C++20 fixes a syntax restriction of lambda expressions: pack expansions in init-capture. Lambda expression in C++ supports parameter packs in the capture clause. Since C++14, generalized captures allow it to init-capture variables from the surrounding scope and use them in a lambda.

The following short code examples exemplify both features.

Parameter Packs in the Capture Clause

```
// parameterPacksLambda.cpp

void hello(int, double, bool) { }

template<typename... Args>
void func(Args... args) {
    auto newFunc = [args...] { return hello(args...); };
    newFunc();
}

int main() {

    func(5, 5.5, true);

}
```

Generalized Captures

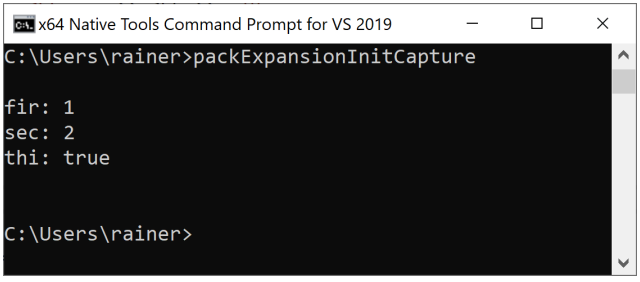
```
// generalizedCaptures.cpp

#include <memory>
#include <utility>

int main() {

    auto uniq = std::make_unique<int>(5);

    auto lamb = [newUniq  = std::move(std::move(uniq))] {
        int val = *newUniq;
    };

    lamb();

}
```

Combining both features was not possible before C++20. Since C++20, the asymmetry is gone.

Pack Expansion in Init-Capture

```
1  // packExpansionInitCapture.cpp
2
3  #include <iostream>
4  #include <memory>
5  #include <utility>
6
7  template<typename Callable, typename ... Args>
8  auto packExpansion(Callable call, Args ... args) {
9      return [call, ...args = std::move(args)] {
10         return call(args...);
11     };
12 }
13
14 void func(int fir, double sec, bool thi) {
15     std::cout << std::boolalpha;
16     std::cout << "fir: "  << fir << '\n';
17     std::cout << "sec: "  << sec << '\n';
18     std::cout << "thi: "  << thi << '\n';
19 }
20
21 int main() {
22
23     std::cout << '\n';
24
25     auto lamb1 = packExpansion(func, 1, 2, true);
26     lamb1();
27
28     std::cout << '\n';
29
30 }
```

In line 10, I apply pack expansion in init-capture. Due to syntactical restrictions, the ellipsis is before `args`: `[call, ...args = std::move(args)]`.

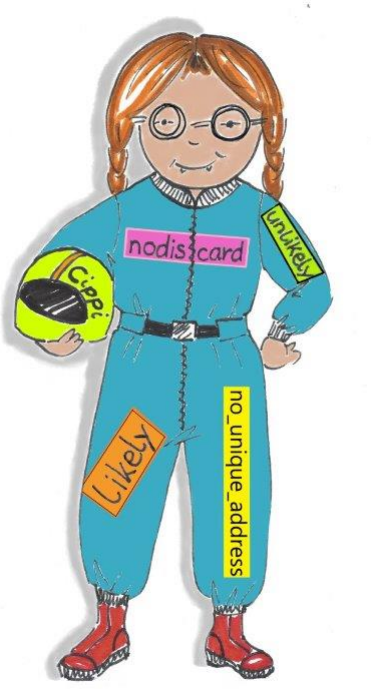


Pack expansion in init-capture

Distilled Information

- With C++20, lambdas can have template parameters. Therefore, a significant cause of undefined behavior with lambdas is gone.
- Lambdas detect when the `this` pointer is implicitly referenced.
- You can use lambda expressions in unevaluated contexts.
- Lambdas can be `consteval` and allow pack expension in the init-capture.

4.8 New Attributes



Cippi is ready for the race

With C++20, we get new and improved attributes such as `[[nodiscard("reason")]]`, `[[likely]]`, `[[unlikely]]`, and `[[no_unique_address]]`. In particular, `[[nodiscard("reason")]]` can be used to explicitly express the intent of our interface.

Attributes

Attributes allow the programmer to express additional constraints on the source code or give the compiler additional optimization possibilities. You can use attributes for types, variables, functions, names, and code blocks. When you use more than one attribute, you can apply each one after the other (`func1`) or all together in one attribute, separated by commas (`func2`):

Use of attributes

```
1  [[attribute1]]  [[attribute2]] [[attribute3]]
2  int func1();
3
4  [[attribute1, attribute2, attribute3]]
5  int func2();
```

Attributes can be implementation-defined language extensions or standard attributes, such as the following list of attributes C++11 - C++17 already have.

- `[[noreturn]]` (C++11): indicates that the function does not return
- `[[carries_dependency]]` (C++11): indicates a dependency chain in release-consume ordering[73]
- `[[deprecated]]` (C++14): indicates that you should not use a name
- `[[fallthrough]]` (C++17): indicates that a fallthrough in a case branch is intentional
- `[[maybe_unused]]` (C++17): suppresses compiler warning about used names

4.8.1 `[[nodiscard("reason")]]`

C++17 introduced the new attribute `[[nodiscard]]` without a reason. C++20 added the possibility to add a message to the attribute.

Discarding objects and error codes

```
1  // withoutNodiscard.cpp
2
3  #include <utility>
4
5  struct MyType {
6
7      MyType(int, bool) {}
8
9  };
10
```

[73] https://en.cppreference.com/w/cpp/atomic/memory_order#Release-Consume_ordering

```
11 template <typename T, typename ... Args>
12 T* create(Args&& ... args) {
13   return new T(std::forward<Args>(args)...);
14 }
15
16 enum class ErrorCode {
17     Okay,
18     Warning,
19     Critical,
20     Fatal
21 };
22
23 ErrorCode errorProneFunction() { return ErrorCode::Fatal; }
24
25 int main() {
26
27     int* val = create<int>(5);
28     delete val;
29
30     create<int>(5);
31
32     errorProneFunction();
33
34     MyType(5, true);
35
36 }
```

Thanks to perfect forwarding and parameter packs, the factory function `create` (line 11) can call any constructor and return a heap-allocated object.

The program has many issues. First, line 30 has a memory leak, because the `int` created on the heap is never deleted. Second, the error code of the function `errorProneFunction` (line 32) is not checked. Lastly, the constructor call `MyType(5, true)` (line 34) creates a temporary, which is created and immediately destroyed. Thast is at least a waste of resources. Now, `[[nodiscard]]` comes into play.

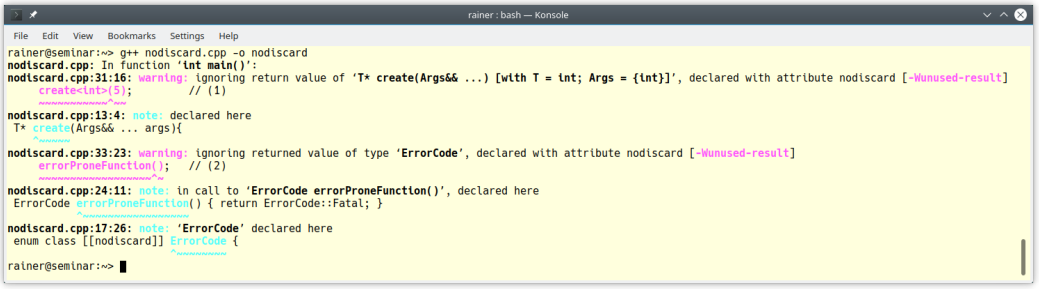
`[[nodiscard]]` can be used in a function declaration, enumeration declaration, or class declaration. If you discard the return value from a function declared as `[[nodiscard]]`, the compiler should issue a warning. The same holds for a function returning by copy an enumeration or a class declared as `[[nodiscard]]`. If you still want to ignore the return value, you can cast it to `void`.

Let us see what this means. In the following example, I use the C++17 syntax of the attribute `[[nodiscard]]`.

Use of the attribute [[nodiscard]] in C++17

```
1  // nodiscard.cpp
2
3  #include <utility>
4
5  struct MyType {
6
7       MyType(int, bool) {}
8
9  };
10
11 template <typename T, typename ... Args>
12 [[nodiscard]]
13 T* create(Args&& ... args){
14   return new T(std::forward<Args>(args)...);
15 }
16
17 enum class [[nodiscard]] ErrorCode {
18     Okay,
19     Warning,
20     Critical,
21     Fatal
22 };
23
24 ErrorCode errorProneFunction() { return ErrorCode::Fatal; }
25
26 int main() {
27
28     int* val = create<int>(5);
29     delete val;
30
31     create<int>(5);
32
33     errorProneFunction();
34
35     MyType(5, true);
36
37 }
```

The factory function `create` (line 13) and the `enum ErrorCode` (line 17) are declared as `[[nodiscard]]`. Consequently, the calls in lines 31 and 33 create warnings.



A C++17 compiler complains about a discarded object and a discarded error code

Way better, but the program still has a few issues. `[[nodiscard]]` cannot be used for functions such as a constructor returning nothing. Therefore, the temporary `MyType(5, true)` (line 35) is still created without a warning. Second, the error messages are too general. As a user of the functions, I want to have a reason why discarding the result is an issue.

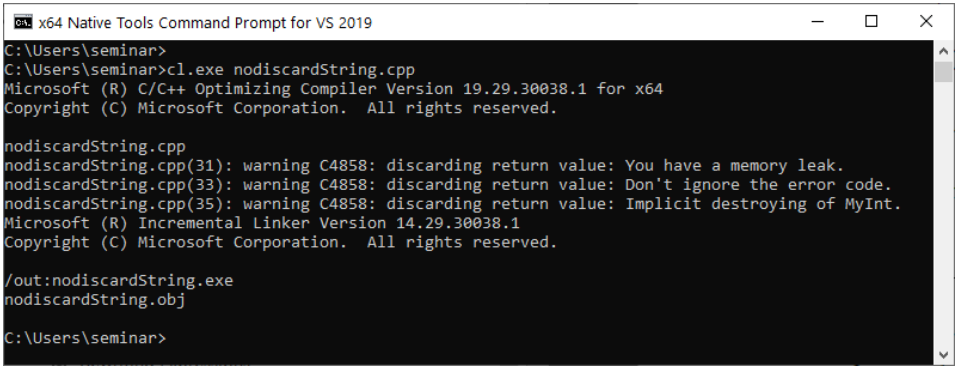
Both issues can be solved with C++20. Constructors can be declared as `[[nodiscard]]`, and the warning can have additional information.

Use of the attribute [[nodiscard]] in C++20

```
1  // nodiscardString.cpp
2
3  #include <utility>
4
5  struct MyType {
6
7       [[nodiscard("Implicit destroying of temporary MyInt.")]] MyType(int, bool) {}
8
9  };
10
11 template <typename T, typename ... Args>
12 [[nodiscard("You have a memory leak.")]]
13 T* create(Args&& ... args){
14   return new T(std::forward<Args>(args)...);
15 }
16
17 enum class [[nodiscard("Don't ignore the error code.")]] ErrorCode {
18     Okay,
19     Warning,
20     Critical,
21     Fatal
22 };
23
24 ErrorCode errorProneFunction() { return ErrorCode::Fatal; }
```

```
25
26  int main() {
27
28      int* val = create<int>(5);
29      delete val;
30
31      create<int>(5);
32
33      errorProneFunction();
34
35      MyType(5, true);
36
37  }
```

Now, the user of the functions gets specific messages. Here is the output of the Microsoft compiler.



A C++20 compiler complains about discarded objects and error codes

The issue with `std::async`

Many existing functions in C++ could benefit from the `[[nodiscard]]` attribute. An ideal candidate is the function `std::async`. When you don't use the return value of `std::asnyc`, what you intended as an asynchronous `std::async` call implicitly becomes synchronous. What should have run in a separate thread behaves instead as a blocking function call. Read more about the counterintuitive behavior of `std::async` in my post "The Special Futures"[74].

While studying the `[[nodiscard]]` syntax on cppreference.com/nodiscard[75], I noticed that the declarations of `std::async`[76] changed with C++20. Here is one:

`std::async` uses in C++20 the attribute `[[nodiscard]]`

```
template<class Function, class... Args>
[[nodiscard]]
std::future<std::invoke_result_t<std::decay_t<Function>,
                                 std::decay_t<Args>...>>
    async( Function&& f, Args&&... args );
```

The return-type of promise `std::async`, is declared as `[[nodiscard]]` in C++20.

The next two attributes `[[likely]]` and `[[unlikely]]` are about optimization.

4.8.2 `[[likely]]` and `[[unlikely]]`

Proposal P0479R5[77] for the attributes `[[likely]]` and `[[unlikely]]` is the shortest proposal I know of. To give you an idea, this is an interesting note to the proposal. "*The use of the likely attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more likely than any alternative path of execution that does not include such an attribute on a statement or label. The use of the unlikely attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more unlikely than any alternative path of execution that does not include such an attribute on a statement or label. A path of execution includes a label if and only if it contains a jump to that label. Excessive usage of either of these attributes is liable to result in performance degradation.*"

In summary, both attributes allow for giving the optimizer a hint regarding the path of execution expected to be more or less likely.

[74] https://www.modernescpp.com/index.php/the-special-futures
[75] https://en.cppreference.com/w/cpp/language/attributes/nodiscard
[76] https://en.cppreference.com/w/cpp/thread/async
[77] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0479r5.html

Give the optimizer a hint with [[likely]]

```
for(size_t i=0; i < v.size(); ++i){
    if (v[i] < 0) [[likely]] sum -= sqrt(-v[i]);
    else sum += sqrt(v[i]);
}
```

The story of optimization goes on with the new attribute `[[no_unique_address]]`. This time the optimization addresses space instead of execution time.

4.8.3 `[[no_unique_address]]`

`[[no_unique_address]]` expresses that this data member of a class need not have an address distinct from all other non-static data members of its class. Consequently, if the member has an empty type, the compiler can optimize it to occupy no memory.

The following program exemplifies the usage of the new attribute.

Use of the attribute [[no_unique_address]]

```
1  // uniqueAddress.cpp
2
3  #include <iostream>
4
5  struct Empty {};
6
7  struct NoUniqueAddress {
8      int d{};
9      [[no_unique_address]] Empty e{};
10 };
11
12 struct UniqueAddress {
13     int d{};
14     Empty e{};
15 };
16
17 int main() {
18
19     std::cout << '\n';
20
21     std::cout << std::boolalpha;
22
23     std::cout << "sizeof(int) == sizeof(NoUniqueAddress): "
24               << (sizeof(int) == sizeof(NoUniqueAddress)) << '\n';
25
```

```
26      std::cout << "sizeof(int) == sizeof(UniqueAddress): "
27                << (sizeof(int) == sizeof(UniqueAddress)) << '\n';
28
29      std::cout << '\n';
30
31      NoUniqueAddress NoUnique;
32
33      std::cout << "&NoUnique.d: " <<  &NoUnique.d << '\n';
34      std::cout << "&NoUnique.e: " <<  &NoUnique.e << '\n';
35
36      std::cout << '\n';
37
38      UniqueAddress unique;
39
40      std::cout << "&unique.d: " <<  &unique.d << '\n';
41      std::cout << "&unique.e: " <<  &unique.e << '\n';
42
43      std::cout << '\n';
44
45  }
```

The class `NoUniqueAddress` has a size equal to `int` (line 7), but not the class `UniqueAddress` (line 12). The members `d` and `e` of `UniqueAddress` (lines 40 and 41) have different addresses but not the members of the class `UniqueAddress` (lines 33 and 34).

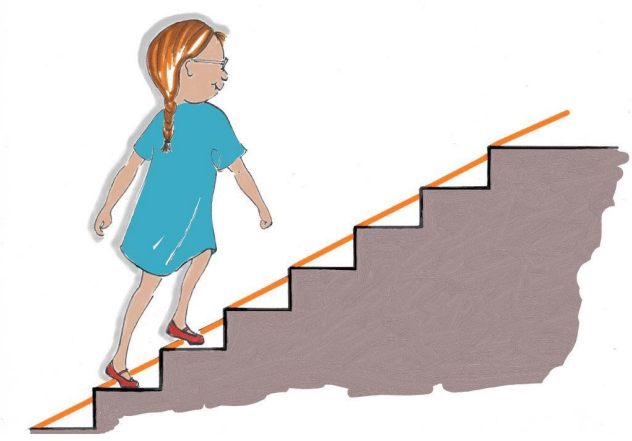
```
sizeof(int) == sizeof(NoUniqueAddress): true
sizeof(int) == sizeof(UniqueAddress): false

&NoUnique.d: 0x7fff44f8fd0c
&NoUnique.e: 0x7fff44f8fd0c

&unique.d: 0x7fff44f8fd04
&unique.e: 0x7fff44f8fd08
```

Use of the class `NoUniqueAddress` and `UniqueAddress`

Distilled Information

- C++20 supports a few new attributes. `[[nodiscard("reason")]]` can be used in various contexts to check if the return value of a function is ignored.
- `[[likely]]` and `[[unlikely]]` allows the programmer to give the compiler a hint which code path is more likely to be executed.
- Thanks to the attribute `[[no_unique_address]]`, data members of a class can have the same address.

4.9 Further Improvements



Cippi goes up

This section presents the remaining small improvements in the C++20 core language.

4.9.1 `volatile`

The abstract in the proposal P1152R0[78] gives a short description of the changes that `volatile` undergoes: "*The proposed deprecation preserves the useful parts of volatile, and removes the dubious / already broken ones. This paper aims at breaking at compile-time code which is today subtly broken at run time or through a compiler update.*"

Before I dive into `volatile`, I want to answer the crucial question: When should you use `volatile`? A note from the C++ standard says that "*volatile is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation.*" That means that for a single thread of execution, the compiler must perform load or store operations in the executable as often as they occur in the source code. `volatile` operations, therefore, cannot be eliminated or reordered. Consequently, you can use `volatile` objects for communication with a signal handler but not for communication with another thread of execution.

Before I show you what semantics of volatile are preserved, I want to start with the deprecated features:

1. Deprecate `volatile` compound assignment, and pre/post increment/decrement
2. Deprecate `volatile` qualification of function parameters or return types
3. Deprecate `volatile` qualifiers in a structured binding declaration

[78] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1152r0.html

If you want to know all the sophisticated details, I strongly suggest you watch the CppCon 2019 talk "Deprecating volatile"[79] from JF Bastien. Here are a few examples from his talk. Additionally, I fixed a few typos in the source code. The numbers in the following code snippets refer to the three deprecations listed earlier.

Deprecated use case for volatile

```
// (1)
int neck, tail;
volatile int brachiosaur;
brachiosaur = neck;    // OK, a volatile store
tail = brachiosaur;    // OK, a volatile load

// deprecated: does this access brachiosaur once or twice
tail = brachiosaur = neck;

// deprecated: does this access brachiosaur once or twice
brachiosaur += neck;

// OK, a volatile load, an addition, a volatile store
brachiosaur = brachiosaur + neck;

########################################
// (2)
// deprecated: a volatile return type has no meaning
volatile struct amber jurassic();

// deprecated: volatile parameters aren't meaningful to the
//             caller, volatile only applies within the function
void trex(volatile short left_arm, volatile short right_arm);

// OK, the pointer isn't volatile, the data it points to is
void fly(volatile struct pterosaur* pterandon);

########################################
(3)
struct linhenykus { volatile short forelimb; };
void park(linhenykus alvarezsauroid) {
    // deprecated: does the binding copy the forelimbs?
    auto [what_is_this] = alvarezsauroid;  // structured binding
    // ...
}
```

[79] https://www.youtube.com/watch?v=KJW_DLaVXIY

`volatile` and Multithreading Semantics

`volatile` is typically used to denote objects that can change independently of the regular program flow. These are, for example, objects in embedded programming that represent an external device (memory-mapped I/O). Because these objects can change independently of the regular program flow and their value is directly written to main memory, no optimized storing in caches takes place. **In other words, volatile avoids aggressive optimization and has no multithreading semantics.**

4.9.2 Range-based for loop with Initializers

With C++20, you can directly use a range-based for loop with an initializer.

Range-based for loop with initializer

```
// rangeBasedForLoopInitializer.cpp

#include <iostream>
#include <string>
#include <vector>

int main() {

    for (auto vec = std::vector{1, 2, 3}; auto v : vec) {
        std::cout << v << " ";
    }

    std::cout << "\n\n";

    for (auto initList = {1, 2, 3}; auto e : initList) {
        e *= e;
        std::cout << e << " ";
    }

    std::cout << "\n\n";

    using namespace std::string_literals;
    for (auto str = "Hello World"s; auto c: str) {
        std::cout << c << " ";
    }

    std::cout << '\n';

}
```

The range-based for loop uses in line 9 a `std::vector`, in line 15 a `std::initializer_list`, and line 23 a `std::string`. Furthermore, in line 9 and line 15 I apply automatically type deduction for class templates, which we had since C++17. Instead of `std::vector<int>`, I just write `std::vector`.

```
1 2 3

1 4 9

H e l l o   W o r l d
```

Use of a range-based for loop with initializers

4.9.3 Virtual `constexpr` function

A `constexpr` function has the potential to run at compile time but can also be executed at run time. Consequently, you can make a `constexpr` function with C++20 virtual. Both directions are possible. A virtual `constexpr` function can override a non-`constexpr` function, and a virtual non-`constexpr` function can override a virtual `constexpr` function. I want to emphasize that override implies that the relevant function of a base class is virtual.

Program `virtualConstexpr.cpp` shows both combinations:

Virtual constexpr functions

```
1  // virtualConstexpr.cpp
2
3  #include <iostream>
4
5  struct X1 {
6      virtual int f() const = 0;
7  };
8
9  struct X2: public X1 {
10     constexpr int f() const override { return 2; }
11 };
12
13 struct X3: public X2 {
14     int f() const override { return 3; }
15 };
16
17 struct X4: public X3 {
18     constexpr int f() const override { return 4; }
19 };
20
21 int main() {
22
```

```
23      X1* x1 = new X4;
24      std::cout << "x1->f(): " << x1->f() << '\n';
25
26      X4 x4;
27      X1& x2 = x4;
28      std::cout << "x2.f(): " << x2.f() << '\n';
29
30  }
```

Line 24 uses virtual dispatch (late binding) via a pointer, line 28 uses virtual dispatch via reference.

```
x1->f(): 4
x2.f(): 4
```

Use of virtual constexpr functions

4.9.4 The new Character Type of UTF-8 Strings: `char8_t`

In addition to the character types `char16_t` and `char32_t` from C++11, C++20 gets the new character type `char8_t`. Type `char8_t` is large enough to represent any UTF-8 code unit (8 bits). It has the same size, signedness, and alignment as an unsigned char, but is a distinct type.

`char` versus `char8_t`

A `char` has one byte. In contrast to a `char8_t`, the number of bits of a byte and hence of a `char` is not defined. Nearly all implementations use 8 bits for a byte. The `std::string` is an alias for a `std::basic_string` of `chars`.

std::string and a `std::string` literal

```
std::string std::basic_string<char>
"Hello World"s
```

Consequently, C++20 has a new typedef for the character type `char8_t` (line 1) and a new UTF-8 string literal (line 2).

A new char8_t character type and an UTF-8 string literal

```
std::u8string std::basic_string<char8_t>
u8"Hello World"
```

The program `char8Str.cpp` shows the straightforward usage of the new character type `char8_t`.

Intuitive usage for the new character type `char8_t`

```
// char8Str.cpp

#include <iostream>
#include <string>

int main() {

    const char8_t* char8Str = u8"Hello world";
    std::basic_string<char8_t> char8String = u8"helloWorld";
    std::u8string char8String2 = u8"helloWorld";

    char8String2 += u8".";

    std::cout << "char8String.size(): " << char8String.size() << '\n';
    std::cout << "char8String2.size(): " << char8String2.size() << '\n';

    char8String2.replace(0, 5, u8"Hello ");

    std::cout << "char8String2.size(): " << char8String2.size() << '\n';

}
```

Without further ado, here is the output of the program:

```
char8String.size(): 10
char8String2.size(): 11
char8String2.size(): 12
```

Use of the new character type `char8_t`

C++20 does not have the output operator for `char8_t` strings. Consequentially, a convenient way is to apply `reinterpret_cast<const char*>` to a `char8_t` string.

Output of a `char8_t` string

```
// char8StrOutput.cpp

#include <iostream>
#include <string>

int main() {

    std::cout << '\n';

```

```
10      const char8_t* char8Str = u8"Dollar: \u20AC";
11      std::basic_string<char8_t> char8String = u8"Euro: \u0024";
12      std::u8string char8String2 = u8"Pound: \u00A3";
13
14      std::cout << reinterpret_cast<const char*>(char8Str) << '\n';
15      std::cout << reinterpret_cast<const char*>(char8String.c_str()) << '\n';
16      std::cout << reinterpret_cast<const char*>(char8String2.c_str()) << '\n';
17
18      std::cout << '\n';
19
20  }
```

The program displays the Dollar (line 14), Euro (line 15), and Pound (line 16) sign.

Output of a `char8_t` string

4.9.5 `using enum` in Local Scopes

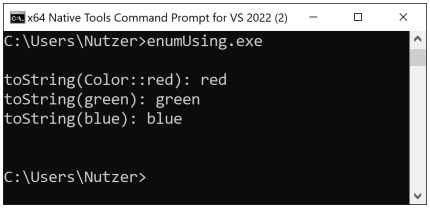
A `using enum` declaration introduces the enumerators of the named enumeration in the local scope.

Introducing enumerators in the local scope

```
1   // enumUsing.cpp
2
3   #include <iostream>
4   #include <string_view>
5
6   enum class Color {
7       red,
8       green,
9       blue
10  };
11
12  std::string_view toString(Color col) {
13    switch (col) {
14      using enum Color;
15      case red:   return "red";
16      case green: return "green";
17      case blue:  return "blue";
```

```
18    }
19    return "unknown";
20  }
21
22  int main() {
23
24      std::cout << '\n';
25
26      std::cout << "toString(Color::red): " << toString(Color::red) << '\n';
27
28      using Color::green;
29
30      std::cout << "toString(green): " << toString(green) << '\n';
31
32      using enum Color;
33
34      std::cout << "toString(blue): " << toString(blue) << '\n';
35
36      std::cout << '\n';
37
38  }
```

The `using enum` declaration (lines 14 and 32) introduces the enumerators of the scoped enumerations `Color` into the local scope. From that point on, the enumerators can be used unscoped (lines 15 - 17 and line 34). Additionally, you can also apply a using declaration for a specific enumeration value (line 28).



Application of `using enum`

4.9.6 Default Member Initializers for Bit Fields

First of all, what is a bit field? Here is the definition from Wikipedia[80]: "*A bit field is a data structure used in computer programming. It consists of a number of adjacent computer memory locations which have been allocated to hold a sequence of bits, stored so that any single bit or group of bits within the*

[80] https://en.wikipedia.org/wiki/Bit_field

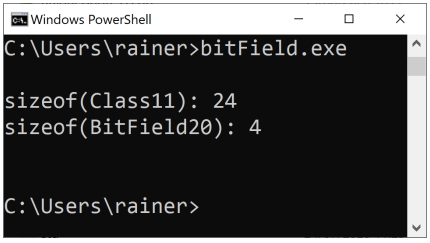
set can be addressed. A bit field is most commonly used to represent integral types of known, fixed bit-width."

With C++20, we can default-initialize the members of a bit field:

Default initializers for the members of a bit field

```
1  // bitField.cpp
2
3  #include <iostream>
4
5  struct Class11 {
6      int i = 1;
7      int j = 2;
8      int k = 3;
9      int l = 4;
10     int m = 5;
11     int n = 6;
12 };
13
14 struct BitField20 {
15     int i : 3 = 1;
16     int j : 4 = 2;
17     int k : 5 = 3;
18     int l : 6 = 4;
19     int m : 7 = 5;
20     int n : 7 = 6;
21 };
22
23 int main () {
24
25     std::cout << '\n';
26
27     std::cout << "sizeof(Class11): " << sizeof(Class11) << '\n';
28     std::cout << "sizeof(BitField20): " << sizeof(BitField20) << '\n';
29
30     std::cout << '\n';
31
32 }
```

According to the members of a class (lines 6 - 11) with C++11, the members of the bit field can have default initializers (lines 15 - 20) with C++20. When you sum up the numbers 3, 4, 5, 6, 7, and 7, you get 32. Hence, 32 bits, or 4 bytes is exactly the size of the `BitField20`:

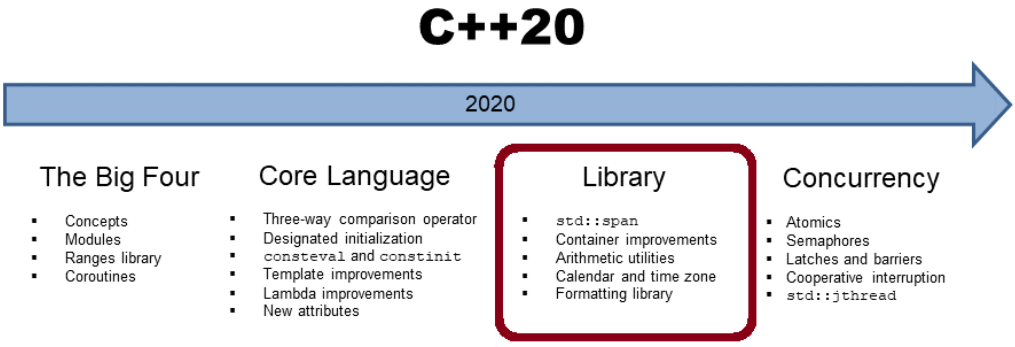


Size information to a bit field

Distilled Information

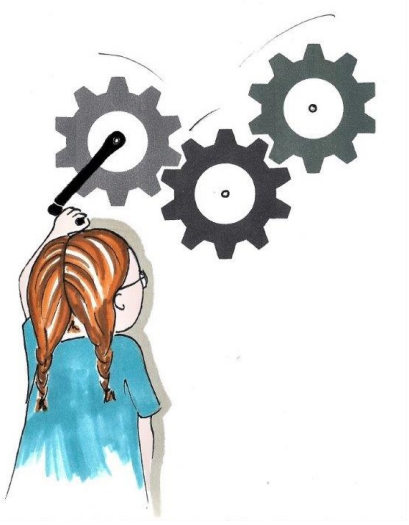
- The meaning of `volatile` is clarified in C++20. `volatile` has no multithreading semantics and should only be used to avoid aggressive optimization because an object may be changed independently of the regular program flow.
- Range-based for loops can use an initializer.
- The new character type `char8_t` is large enough to represent 8 bits.
- A `using enum` declaration introduces the enumerators of a named enumeration in the local scope.
- The members of a bit field can be default-initialized.
- A `constexpr` function can be virtual.

5. The Standard Library



In addition to the ranges library, the C++20 standard library has many new features to offer. A `std::span` as a non-owning reference to a contiguous memory area, improved string, and container implementations, and improved algorithms. Additionally, the chrono library of C++11 is extended with calendar and time-zone capabilities. Last but not least, text can be safely and powerfully formatted.

5.1 The Ranges Library



Cippi starts the pipeline job

Thanks to the ranges library in C++20, working with the Standard Template Library (STL) is much more comfortable and powerful. The algorithms of the ranges library are lazy, can work directly on containers, and are easy to assemble. To make it short: The comfort and the power of the ranges library are due to its functional ideas.

Before I dive into the details, here is a first example of the ranges library:

Combining the transform and filter functions

```
// rangesFilterTransform.cpp

#include <iostream>
#include <ranges>
#include <vector>

int main() {

    std::vector<int> numbers = {1, 2, 3, 4, 5, 6};

    auto results = numbers | std::views::filter([](int n){ return n % 2 == 0; })
                           | std::views::transform([](int n){ return n * 2; });

    for (auto v: results) std::cout << v << " ";     // 4 8 12

}
```

You have to read the expression from left to right. The pipe symbol stands for function composition: First, all numbers which are even can pass (`std::views::filter([](int n){ return n % 2 == 0; })`). After that, each remaining number is mapped to its double (`std::views::transform([](int n){ return n * 2; })`). The small example shows two new features of the ranges library: function composition being applied on the entire container.

Now you should be prepared for the details. Let's go back to square one: ranges and views are concepts.

5.1.1 Ranges

I already presented the concept range in the chapter on concepts. Consequently, here's a brief refresher.

A range that is provided by a begin iterator and an end sentinel specifies a group of items that you can iterate over.

The containers of the STL are ranges but not views.

The sentinel specifies the end of a range.

5.1.1.1 Sentinel

For the containers of the STL, the end iterator is the sentinel. With C++20, the type of the sentinel can be different from the type of the begin iterator. Depending on the range, a null terminator `\0` may end a string, an empty string `std::string{}` may end a list of words, a `std::nullptr` may end a linked list, or the number `-1` may end a list of non-negative numbers.

The following example uses sentinels for a C-string and a `std::vector<int>`.

Space and a negative number as sentinel

```
1  // sentinel.cpp
2
3  #include <iostream>
4  #include <algorithm>
5  #include <compare>
6  #include <vector>
7
8  struct Space {
9  bool operator== (auto pos) const {
10         return *pos == ' ';
11     }
12 };
13
14 struct NegativeNumber {
15     bool operator== (auto num) const {
```

```
16          return *num < 0;
17      }
18  };
19
20  struct Sum {
21      void operator()(auto n) { sum += n; }
22      int sum{0};
23  };
24
25  int main() {
26
27      std::cout << '\n';
28
29      const char* rainerGrimm = "Rainer Grimm";
30
31      std::ranges::for_each(rainerGrimm, Space{}, [] (char c) { std::cout << c; });
32      std::cout << '\n';
33      for (auto c: std::ranges::subrange{rainerGrimm, Space{}}) std::cout << c;
34      std::cout << '\n';
35
36      std::ranges::subrange rainer{rainerGrimm, Space{}};
37      std::ranges::for_each(rainer, [] (char c) { std::cout << c << ' '; });
38      std::cout << '\n';
39      for (auto c: rainer) std::cout << c << ' ';
40      std::cout << '\n';
41
42
43      std::cout << "\n";
44
45
46      std::vector<int> myVec{5, 10, 33, -5, 10};
47
48      for (auto v: myVec) std::cout << v << " ";
49      std::cout << '\n';
50
51      auto [tmp1, sum] = std::ranges::for_each(myVec, Sum{});
52      std::cout << "Sum: " << sum.sum << '\n';
53
54      auto [tmp2, sum2] = std::ranges::for_each(std::begin(myVec), NegativeNumber{},
55                                                Sum{} );
56      std::cout << "Sum: " << sum2.sum << '\n';
57
58      std::ranges::transform(std::begin(myVec), NegativeNumber{},
59                             std::begin(myVec), [](auto num) { return num * num; });
60      std::ranges::for_each(std::begin(myVec), NegativeNumber{},
```

```
61                          [](int num) { std::cout << num << " "; });
62      std::cout << '\n';
63      for (auto v: std::ranges::subrange{ std::begin(myVec), NegativeNumber{}}) {
64          std::cout << v << " ";
65      }
66
67      std::cout << "\n\n";
68
69  }
```

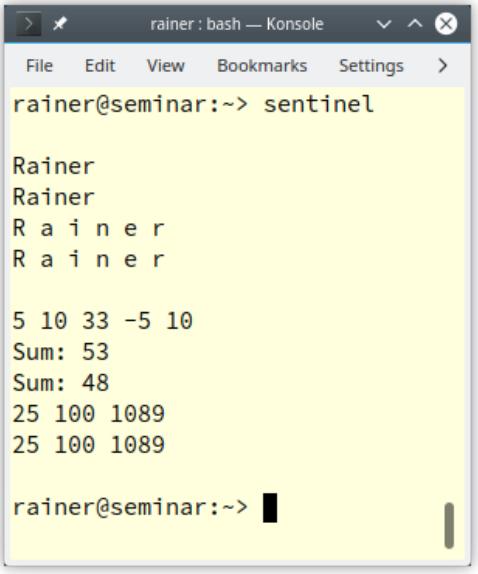
The program defines two sentinels: `Space` (line 8) and `NegativeNumber` (line 14). Both define the equal operator. Thanks to the `<compare>` header, the compiler auto-generates the non-equal operator. The non-equal operator is required when using algorithms such as `std::ranges_for_each` or `std::ranges::tranform` with a sentinel. Let me start with the sentinel `Space`.

Line 31 applies the sentinel `Space{}` directly onto the string `rainerGrimm`. Creating a `std::ranges::subrange` (line 33) allows it to use the sentinel in a range-based for-loop. You can also define a `std::ranges::subrange` and use it directly in the algorithm `std::ranges::for_each` (line 37) or a range-based for-loop (line 39).

My second example uses a `std::vector<int>`, filled with the values `{5, 10, 33, -5, 10}`. The sentinel `NegativeNumber` checks if a number is negative. First, I sum up all values using the function object `Sum` (lines 20 - 23). `std::ranges::for_each` returns a pair `(it, func)`. `it` is the successor of the sentinel and `func` the function object applied to the range. Thanks to structured binding[1], I can directly define the variables `sum` and `sum2` and display their values (lines 52 and 56). `std::ranges::for_each` uses the sentinel `NegativeNumber`. Consequently, `sum2` has the sum up to the sentinel. The call `std::ranges::transform` (line 58) transforms each element to its square: `[](auto num){ return num * num}`. The transformation stops with the sentinel `NegativeNumber`. Line 60 and line 63 display the transformed values.

Finally, here is the output of the program.

[1] https://en.cppreference.com/w/cpp/language/structured_binding



Use of sentinels

5.1.1.2 New Iterators and Sentinels

C++20 supports new iterators and special sentinels. You must include the header `<iterator>` to use them.

Let me start with the iterators.

New iterators

Iterator	Description
`std::counted_iterator(it, count)`	Uses `count` to specify the end of the range
`std::common_iterator(it, sent)`	Unifies an iterator/sentinel pair

Ranges also have three special sentinels:

Special sentinels

Iterator	Description
`std::default_sentinel`	For an iterator that knows the bound of its range
`std::unreachable_sentinel`	For an end iterator that can never be reached
`std::move_sentinel`	For an end iterator that maps copies to moves

The following example applies the new iterators and the special sentinels.

New iterators and sentinels

```
1  // iteratorSentinels.cpp
2
3  #include <iterator>
4  #include <algorithm>
5  #include <iostream>
6  #include <vector>
7
8  int main() {
9
10     std::cout << '\n';
11
12     std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
13
14     std::ranges::copy(vec.begin(),
15         vec.begin() + 4, std::ostream_iterator<int>{std::cout, " "});        // 1 2 3 4
16
17     std::cout << '\n';
18
19     std::ranges::copy(std::counted_iterator{vec.begin(), 4},                 // ERROR
20         vec.end(), std::ostream_iterator<int>{std::cout, " "});
21
22     std::cout << '\n';
23
24     std::ranges::copy(std::counted_iterator{vec.begin(), 4},
25         std::default_sentinel, std::ostream_iterator<int>{std::cout, " "});  // 1 2 3 4
26
27
28     std::cout << '\n';
29
30 }
```

The program uses two ways to display the first four elements of the `std::vector vec`. Line 14 uses

a begin and an end iterator. Line 24 applies a `std::counted_iterator` and a `std::default_sentinel`. Using a `std::counted_iterator` and end iterator does not compile (line 19).

5.1.2 Views

Views are lightweight ranges. A view does not own data, and its time complexity to copy, move, or assign is constant. The containers of the STL and `std::string` are ranges but not views. A view allows you to access ranges, iterate through ranges, or modify or filter elements of a range.

5.1.2.1 `std::ranges::view_interface`

Views derive from the class `std::ranges::view_interface<View>` which derives from the class `std::ranges::view_base`.

`std::ranges::view_interface` supports a few basic operations:

Operations of a `std::ranges::view_interface` `v`

Operation	Requirement	Description
`v.empty()`	At least a forward iterator	Returns if `v` is empty
`if (v)`	At least a forward iterator	Returns if `v` is not empty
`v.size()`		Returns the number of elements
`v.front()`	At least a forward iterator	Returns the first element
`v.back()`	At least bidirectional iterator `begin` and `end` have the same type	Returns the last element
`v[i]`	At least random-access iterator	Returns the `i`-th element
`v.data()`	Returns a raw pointer to the elements	Elements are in contiguous memory

As you may guess, a `view` is also a concept. For a more detailed discussion about the concept `view`, read the chapter on concepts: views.

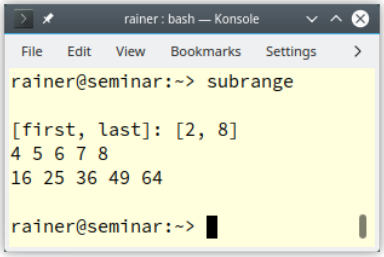
5.1.2.2 `std::ranges::subrange`

The class template `std::ranges::subrange` combines an iterator and a sentinel into a single view. Thanks to `std::ranges::subrange`, creating a view out of a begin iterator and a sentinel is straightforward.

Creating a subrange

```
// subrange.cpp

#include <iterator>
#include <algorithm>
#include <iostream>
#include <ranges>
#include <vector>

int main() {

    std::cout << '\n';

    std::vector vec{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    auto [first, last]  = std::ranges::subrange{vec.begin() + 2, vec.end() - 2};

    std::cout << "[first, last]: " << "[" <<  *first << ", " << *last << "]";

    std::cout << '\n';

    std::ranges::subrange sub1{std::ranges::find_if(vec, [](int i){ return i > 3; }),
                               std::ranges::find_if(vec, [](int i){ return i > 8; })};

    for (auto v: sub1) std::cout << v << " ";

    std::cout << '\n';

    auto transVec = sub1 | std::views::transform([](int i) { return i * i; });

    for (auto v: transVec) std::cout << v << " ";

    std::cout << "\n\n";

}
```

The call `std::ranges::subrange{vec.begin() + 2, vec.end() - 2}` (line 15) returns an iterator and a sentinel. Line 21 creates a subrange of elements bigger than 3 and smaller than 8 (line21). You can directly apply a range adaptor onto the subrange `sub1` (line 28).



Creating subranges

The created subrange models the concept std::range::sized_range, if the subrange was created with random-access iterators of the same type. If not, you can create a sized subrange by providing a third constructor argument.

Creating a sized subrange

```
std::list numbers = {1, 2, 3, 4, 5, 6};

std::ranges::subrange sub2{numbers.begin() + 1 , numbers.end() - 1, numbers.size() - 2};
```

`std::ranges::subrange` supports basic operations:

Operations of a `std::ranges::subrange sub`

Operation	Requirement	Description
`sub.begin()`		Returns the begin iterator
`sub.end()`		Returns the sentinel
`sub.empty()`		Returns if `sub` is empty
`sub.size()`	Available if sized	Returns the number of elements
`sub.front()`	Available if at least a forward iterator	Returns the first element
`sub.back()`	Available if at least a bidirectional iterator and common	Returns the last element
`sub[i]`	Available if at least a random-access iterator	Returns the `i`-th element
`sub.data()`	Available if a contiguous iterator	Returns a raw pointer to the elements

Operations of a `std::ranges::subrange sub`

Operation	Requirement	Description
`sub.next(n = 1)`		Returns a subrange starting with the `n`-th element
`sub.prev(n = 1)`		Returns a subrange starting with the `n`-th element before the first element
`sub.advance(n)`		Returns a subrange starting n-th elements later (`n`-th elements earlier if `n < 0`
`auto[beg, end] = sub`		Creates `beg` and `end` with begin and end of `sub`

The concept of a view is strongly related range adaptors.

5.1.3 Range Adaptors

A range adaptor transforms a range into a view.

The following code snippet shows range adaptors operating on a range.

Range adaptors operating on a range

```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6};

auto results = numbers | std::views::filter([](int n){ return n % 2 == 0; })
                       | std::views::transform([](int n){ return n * 2; });
```

In this code snippet, `numbers` is the range, and range adaptors `std::views::filter` and `std::views::transform` create the views. Additionally, `std::string_view`[2] and `std::span` are also views.

Thanks to range adaptors, C++20 allows programming in a functional style. Range adaptors can be combined, and the resulting views are lazy. I already presented two range adaptors, but C++20 offers more.

[2] https://en.cppreference.com/w/cpp/string/basic_string_view

Range adaptors in C++20

View	Description
`std::views::all_t` `std::views::all`	Converts a range into a view.
`std::ranges::ref_view`	Takes all elements of another range.
`std::ranges::owning_view`	Owns uniquely another range.
`std::ranges::iota_view` `std::views::iota`	Generates a view of incremented values.
`std::ranges::single_view` `std::views::single`	Owns a single value.
`std::ranges::empty_view` `std::views::empty`	A view with no elements.
`std::ranges::basic_istream_view` `std::ranges::istream_view`	Reads elements from a stream.
`std::ranges::filter_view` `std::views::filter`	Takes the elements that satisfy the predicate.
`std::ranges::transform_view` `std::views::transform`	Transforms each element.
`std::ranges::take_view` `std::views::take`	Takes the first `n` elements of another view.
`std::ranges::take_while_view` `std::views::take_while`	Takes the elements of another view as long as the predicate returns `true`.
`std::ranges::drop_view` `std::views::drop`	Skips the first `n` elements of another view.
`std::ranges::drop_while_view` `std::views::drop_while`	Skips the initial elements of another view until the predicate returns `false`.
`std::ranges::join_view` `std::views::join`	Joins a view of ranges.
`std::ranges::split_view` `std::views::split`	Splits a view by using a delimiter.

Range adaptors in C++20

View	Description
`std::ranges::lazy_split_view` `std::views::lazy_split`	Splits a view by using a delimiter.
`std::views::counted`	Creates a view from a begin iterator and a count.
`std::ranges::common_view` `std::views::common`	Converts a view into a `std::ranges::common_range`.
`std::ranges::reverse_view` `std::views::reverse`	Iterates in reverse order.
`std::ranges::elements_view` `std::views::elements`	Creates a view on the `n`-th element of tuples.
`std::ranges::keys_view` `std::views::keys`	Creates a view on the first element of pair-like values.
`std::ranges::values_view` `std::views::values`	Creates a view on the second element of pair-like values.

Strictly speaking, the functions in the namespace `std::views` like `std::views::values` are range adaptors, and the types in the namespace `std::ranges` like `std::ranges::values_view` are views. A range adaptor `range | std::views::values` operates on a range and returns a view but a view gets a range as argument: `std::ranges::values_view{range}`.

`std::ranges_view` stores a reference to the underlying range. It is a borrowed range and can refer to the the underlying range as long as it is valid. Reallocation of the underlying range does not invalidate the `std::ranges_view`.

`std::ranges::owning_view` takes ownership of the elements of another range. It can only be constructed from an rvalue and cannot be copied.

std::owning_view

```
1  std::vector<int> vec{1, 2, 3, 4, 5};
2
3  std::ranges::owning_view vec2{vec};            // ERROR
4  std::ranges::owning_view vec3{std::move(vec)}; // OK
5
6  std::ranges::owning_view vec4{vec2};           // ERROR
7  std::ranges::owning_view vec5{std::move(v2)};  // OK
```

Initializing a `std::ranges::owning_view` from a lvalue, such as in lines 3 or 6, is an error. In the first case, the lvalue is a `std::vector<int>`, and in the second case, a `std::ranges::owning_view`.

The subtile difference between `std::ranges::split_view` and `std::ranges::lazy_split_view` is that `std::ranges::split_view` cannot iterate over a constant view and requires a forward iterator. `std::ranges::split_view` does not know its size.

Splitting a string

```
1  // splitView.cpp
2
3  #include <iostream>
4  #include <string>
5  #include <ranges>
6
7  int main() {
8
9      std::string myString = "Hello:world!";
10
11     for (auto subRange: myString | std::views::split(':')) {
12         for (auto c: subRange) {
13             std::cout << c;
14         }
15         std::cout << ':';
16     }
17
18     std::cout << '\n';
19
20     myString = "ThisTESTisTESTaTESTtest.";
21
22     for (auto subRange: myString | std::views::split(std::string("TEST"))) {
23         for (auto c: subRange) {
24             std::cout << c;
25         }
26         std::cout << ':';
27     }
28
29 }
```

When you split a range, you get back a subrange. First, I split `myString` by space (line 11) and second by the string `TEST` (line 23). Additionally, I add a colon `:` to each resulting subrange.

```
Hello:world!:
This:is:a:test.:
```

Splitting a string

5.1.3.1 Three ways to use Range Adaptors

In general, you can use a range adaptor such as `std::views::drop` or the corresponding view `std::ranges::drop_view`. Consequently, you have to use the arguments of the function call differently:

Invocation of `std::view::drop` and `std::ranges::drop_view`

```
const auto numbers = {1, 2, 3, 4, 5};

auto firstThree = numbers | std::views::drop(3);
std::ranges::drop_view firstThree{numbers, 3};
auto firstThree = std::views::drop(3)(numbers);
```

More formally, here are the three syntactic forms of using range adaptors or its corresponding view. Its line numbers refer to the line number in the previous example using `std::views::drop`, and `std::ranges::drop_view`.

Invocation of `std::view::drop` and `std::ranges::drop_view`

```
Range | RangeAdaptor(args...) (line 3)
View(Range, args...)          (line 4)
RangeAdaptor(args...)(Range)  (line 5)
```

The range adaptor or view can accept an arbitrary number of arguments: `args...`.

There a three range adaptors to create views out of ranges or create a special view.

5.1.3.2 `std::views::all`

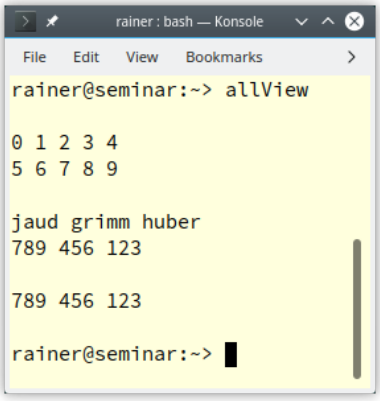
The range adaptor `std::view::all` is convenient to convert a range into a view.

Converting a range into a view

```
// allView.cpp

#include <iostream>
#include <ranges>
#include <vector>
#include <unordered_map>

int main() {

    std::cout << '\n';

    std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    for (auto v : std::views::all(myVec) | std::views::take(5)) {
```

```
14          std::cout << v << " ";
15      }
16
17      std::cout << '\n';
18
19      for (auto v : std::views::all(myVec) | std::views::drop(5)) {
20          std::cout << v << " ";
21      }
22
23      std::cout << "\n\n";;
24
25      std::unordered_map<std::string, int> myMap{{"huber", 123}, {"grimm", 456},
26                                                 {"jaud", 789}};
27      for (auto pa : std::views::all(myMap)) {
28          std::cout << pa.first << " ";
29      }
30
31      std::cout << '\n';
32
33      for (auto pa : std::views::all(myMap)) {
34          std::cout << pa.second << " ";
35      }
36
37      std::cout << "\n\n";
38
39      auto valuesView = std::views::all(myMap);
40
41      for (auto pa : std::views::all(valuesView)) {
42          std::cout << pa.second << " ";
43      }
44
45      std::cout << "\n\n";
46
47  }
```

`std::vector` (line 12) and `std::unordered_map` (line 25) are the ranges that are converted into views. `std::views::take(5)` takes the first five elements from the view and `std::view::drop(5)` drops the first view elements from the view. You can also apply `std::views::all` to a `std::unordered_map` (lines 27 and 33). In this case, you get a `std::pair pa` and can address its first element with `pa.first` (line 28) and the second with `pa.second` (line 34). A view is only a special range. Consequentially, you can directly apply a view onto a view (lines 39 and 41).



Converting a range into a view

Admittedly, lines 33 to 35 and 41 to 43 are too complicated. Just use the views `std::views::keys`, and `std::views::values` to get a view of the keys and values of the associative container.

Views on the keys and values of a associative container, lang=C++

```
for (auto k : std::views::keys(myMap)) {
    std::cout << k << " ";
}

for (auto v : std::views::values(myMap)) {
    std::cout << v << " ";
}
```

There is an easier way to get a view of the first five or last five elements of a range.

5.1.3.3 `std::views::counted`

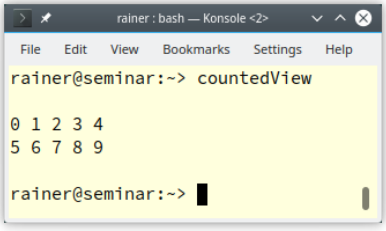
`std::views::counted` creates a view from a begin iterator and a count.

Create a view from a begin iterator and a count

```
// countedView.cpp

#include <iostream>
#include <ranges>
#include <vector>

int main() {

    std::cout << '\n';
```

```
10
11      std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
12      for (auto v : std::views::counted(std::begin(myVec), 5)) {
13          std::cout << v << " ";
14      }
15
16      std::cout << '\n';
17
18      for (auto v : std::views::counted(std::begin(myVec) + 5, 5)) {
19          std::cout << v << " ";
20      }
21
22      std::cout << "\n\n";;
23
24  }
```

Lines 12 and 28 directly create the view from the begin iterator and the count. If the count is too high, you get undefined behavior.



Create a view from a begin iterator and a count

5.1.3.4 `std::views::common`

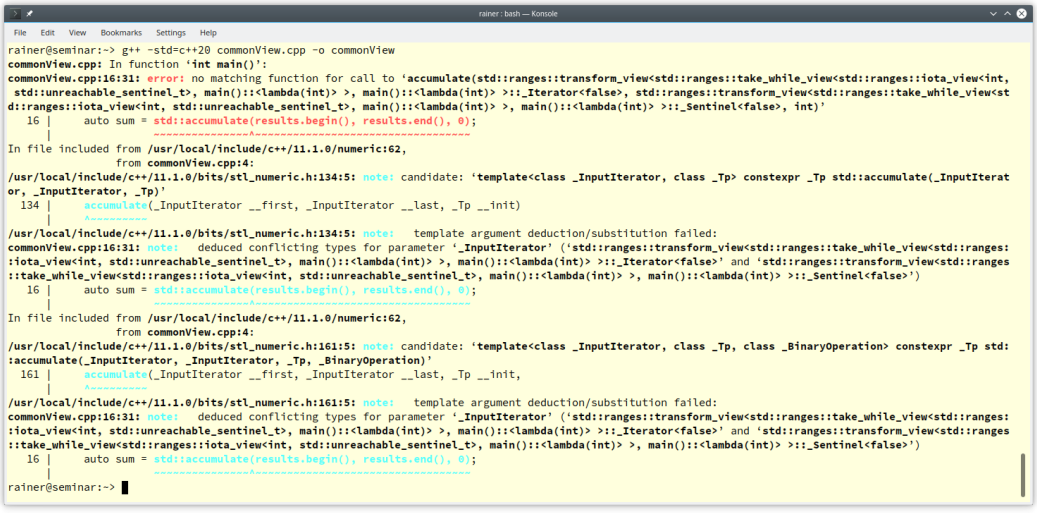
The function `std::views::common` is quite convenient when you have a view but need a `std::ranges::common_range`. In the following program, `commonView.cpp`, I sum up all squares of the numbers from 100 to 200.

Converting a view into a `std::views::common`

```
1  // commonView.cpp
2
3  #include <iostream>
4  #include <numeric>
5  #include <ranges>
6  #include <vector>
7
8  int main() {
9
10     std::cout << '\n';
11
12     auto results = std::views::iota(100) |
13                    std::views::take_while([](int i) { return i <= 200; }) |
14                    std::views::transform([](int i) { return i * i;});
15
16     /*
17     auto results = std::views::iota(100) |
18                    std::views::take_while([](int i) { return i <= 200; }) |
19                    std::views::transform([](int i) { return i * i;}) |
20                    std::views::common;
21
22     */
23
24     auto sum = std::accumulate(results.begin(), results.end(), 0);
25
26     std::cout << "sum: " << sum << '\n';
27
28     std::cout << '\n';
29
30 }
```

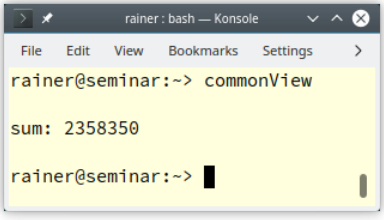
The ranges library provides only pendants to the algorithms of the algorithm[3] and the memory[4] library, but not to the algorithms of the numeric[5] library. Consequentially, I use `std::accumulate` from the numeric library, to sum up all squares (line 24). The compilation of the program fails with a cryptic error message:

[3] https://en.cppreference.com/w/cpp/header/algorithm
[4] https://en.cppreference.com/w/cpp/header/memory
[5] https://en.cppreference.com/w/cpp/numeric



Compilation error invoking `std::accumulate` with a view

The reason for the compilation error is straightforward but also surprising. The composition of range adapter algorithms (lines 12 - 14) returns a view. `std::accumulate` on the other hand requires harmonized types for the begin and end iterator. Converting the view into a `std::view::common` (lines 17 - 20) solves the issue. Now, I can directly feed `results` into `std::accumulate` (line 24).



Summing up all squares from 100 to 200

Views on Temporary Ranges

Views do not own data. Therefore, views do not extend the lifetime of their data. Consequently, views can only be created on lvalues. The compilation fails if you create a view on a temporary range.

Creating views on temporary ranges

```
1  // temporaryRange.cpp
2
3  #include <initializer_list>
4  #include <ranges>
5
6
7  int main() {
8
9     const auto numbers = {1, 2, 3, 4, 5};
10
11    auto firstThree = numbers | std::views::drop(3);
12    // auto firstThree = {1, 2, 3, 4, 5} | std::views::drop(3); ERROR
13
14    std::ranges::drop_view firstFour{numbers, 4};
15   // std::ranges::drop_view firstFour{{1, 2, 3, 4, 5}, 4}; ERROR
16
17 }
```

Lines 12 and 15 cause a compilation error. The use of the lvalue `numbers` in lines 11 and 14 is valid.

5.1.4 Direct on the Container

The algorithms of the Standard Template Library (STL) are sometimes a little inconvenient. They need a begin and an end iterator. This is often more than you want to write.

Algorithms of the STL need both begin and end iterators

```
// sortClassical.cpp

#include <algorithm>
#include <iostream>
#include <vector>

int main()  {

    std::vector<int> myVec{-3, 5, 0, 7, -4};
    std::sort(myVec.begin(), myVec.end());
```

```
    for (auto v: myVec) std::cout << v << " "; // -4, -3, 0, 5, 7

}
```

Wouldn't it be nice if `std::sort` could be executed on the entire container? Thanks to the ranges library, this is possible in C++20.

Algorithms of the ranges library operate directly on the container

```
// sortRanges.cpp

#include <algorithm>
#include <iostream>
#include <vector>

int main()  {

    std::vector<int> myVec{-3, 5, 0, 7, -4};
    std::ranges::sort(myVec);
    for (auto v: myVec) std::cout << v << " "; // -4, -3, 0, 5, 7

}
```

The algorithms of the algorithm library[6] and the memory libray[7] have ranges pendants. They start with the namespace `std::ranges`. The algorithms of the numeric library[8] have no ranges pendant.

When you study the overloads of `std::ranges::sort`, you notice that they support a projection.

5.1.4.1 Projection

`std::ranges::sort` has two overloads:

[6] https://en.cppreference.com/w/cpp/header/algorithm
[7] https://en.cppreference.com/w/cpp/header/memory
[8] https://en.cppreference.com/w/cpp/header/numeric

```
template <std::random_access_iterator I, std::sentinel_for<I> S,
          class Comp = ranges::less, class Proj = std::identity>
requires std::sortable<I, Comp, Proj>
constexpr I sort(I first, S last, Comp comp = {}, Proj proj = {});

template <ranges::random_access_range R, class Comp = ranges::less,
          class Proj = std::identity>
requires std::sortable<ranges::iterator_t<R>, Comp, Proj>
constexpr ranges::borrowed_iterator_t<R> sort(R&& r, Comp comp = {}, Proj proj = {});
```

When you study the second overload, you notice that it takes a sortable range `R`, a predicate `Comp`, and a projection `Proj`. The predicate `Comp` uses for default less, and the projection `Proj` the identity `std::identity`[9] that does return its arguments unchanged. A projection is a mapping of a set into a subset. A projection can be

- a callable such as a lambda
- a pointer to a member function or data member

Let me show you what that means:

Applying projections on data types

```
// rangeProjection.cpp

#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>

struct PhoneBookEntry{
    std::string name;
    int number;
};

void printPhoneBook(const std::vector<PhoneBookEntry>& phoneBook) {
    for (const auto& entry: phoneBook) std::cout << "(" << entry.name << ", "
                                                 << entry.number << ")";
    std::cout << "\n\n";
}

int main() {

    std::cout << '\n';

    std::vector<PhoneBookEntry> phoneBook{ {"Brown", 111}, {"Smith", 444},
```

[9] https://en.cppreference.com/w/cpp/utility/functional/identity

```
24      {"Grimm", 666}, {"Butcher", 222}, {"Taylor", 555}, {"Wilson", 333} };
25
26      std::ranges::sort(phoneBook, {}, &PhoneBookEntry::name);    // ascending by name
27      printPhoneBook(phoneBook);
28
29      std::ranges::sort(phoneBook, std::ranges::greater() ,
30                        &PhoneBookEntry::name);                   // descending by name
31      printPhoneBook(phoneBook);
32
33      std::ranges::sort(phoneBook, {}, &PhoneBookEntry::number); // ascending by number
34      printPhoneBook(phoneBook);
35
36       std::ranges::sort(phoneBook, std::ranges::greater(),
37                         &PhoneBookEntry::number);              // descending by number
38      printPhoneBook(phoneBook);
39
40  }
```

phoneBook (line 23) has structs of type PhoneBookEntry (line 8). A PhoneBookEntry consists of a name and a number. Thanks to projections, the phoneBook can be sorted in ascending order by name (line 26), descending order by name (line 29), ascending order by number (line 33), and descending order by number (line 36). The empty curly braces in the expression std::ranges::sort(phoneBook, {}, &PhoneBookEntry::name) cause the default construction of the sort criteria which is in this case is std::less.

```
(Brown, 111)(Butcher, 222)(Grimm, 666)(Smith, 444)(Taylor, 555)(Wilson, 333)

(Wilson, 333)(Taylor, 555)(Smith, 444)(Grimm, 666)(Butcher, 222)(Brown, 111)

(Brown, 111)(Butcher, 222)(Wilson, 333)(Smith, 444)(Taylor, 555)(Grimm, 666)

(Grimm, 666)(Taylor, 555)(Smith, 444)(Wilson, 333)(Butcher, 222)(Brown, 111)
```

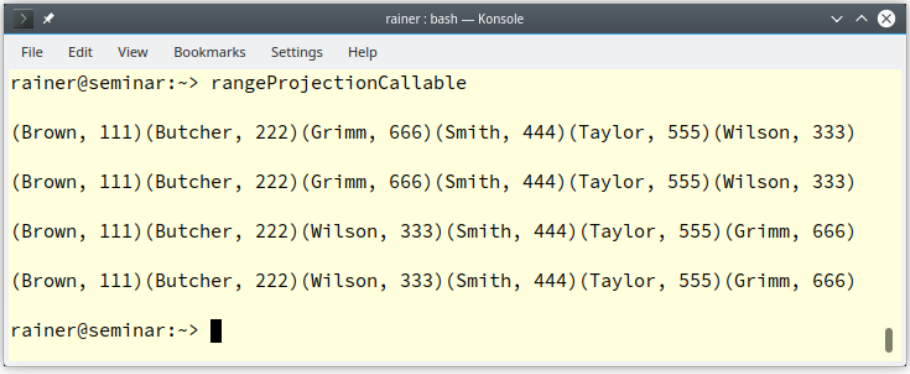
Applying projections on data types

When your projection is more demanding, you can use a callable such as a lambda expression.

Use of callables as projections

```
// rangeProjectionCallable.cpp

#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>

struct PhoneBookEntry{
    std::string name;
    int number;
};

void printPhoneBook(const std::vector<PhoneBookEntry>& phoneBook) {
    for (const auto& entry: phoneBook) std::cout << "(" << entry.name << ", "
                                                 << entry.number << ")";
    std::cout << "\n\n";
}

int main() {

    std::cout << '\n';

    std::vector<PhoneBookEntry> phoneBook{ {"Brown", 111}, {"Smith", 444},
    {"Grimm", 666}, {"Butcher", 222}, {"Taylor", 555}, {"Wilson", 333} };

    std::ranges::sort(phoneBook, {}, &PhoneBookEntry::name);
    printPhoneBook(phoneBook);

    std::ranges::sort(phoneBook, {}, [](auto p){ return p.name; } );
    printPhoneBook(phoneBook);

    std::ranges::sort(phoneBook, {}, [](auto p) {
        return std::to_string(p.number) + p.name;
    });
    printPhoneBook(phoneBook);

    std::ranges::sort(phoneBook, [](auto p, auto p2) {
        return std::to_string(p.number) + p.name <
               std::to_string(p2.number) + p2.name;
    });
    printPhoneBook(phoneBook);

}
```

`std::ranges::sort` in line 26 uses the attribute `PhoneBookEntry::name` as projection. Line 29 shows the equivalent lambda expression `[](auto p){ return p.name; }` as projection. The projection in line 32 is more demanding. It uses the stringified number concatenated with the `p.name`. Of course, you can use the concatenated stringified number and the name directly as sorting criteria. In this case, the algorithm call in line 32 is easier to read than the one in line 37.



Use of a callable as range projection

Most ranges algorithms support projections.

5.1.4.2 Direct Views on Keys and Values

Furthermore, you can create direct views on the keys (line 16) and the values (line 24) of a `std::unordered_map`.

Views on the keys and the values of a `std::unordered_map`

```
1  // rangesEntireContainer.cpp
2
3  #include <iostream>
4  #include <ranges>
5  #include <string>
6  #include <unordered_map>
7
8
9  int main() {
10
11   std::unordered_map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33},
12                                                  {"tale", 45}, {"dog", 4},
13                                                  {"cat", 34}, {"fish", 23} };
14
15   std::cout << "Keys:" << '\n';
```

```
16    auto names = std::views::keys(freqWord);
17    for (const auto& name : names){ std::cout << name << " "; }
18    std::cout << '\n';
19    for (const auto& name : std::views::keys(freqWord)){ std::cout << name << " "; }
20
21    std::cout << "\n\n";
22
23    std::cout << "Values: " << '\n';
24    auto values = std::views::values(freqWord);
25    for (const auto& value : values){ std::cout << value << " "; }
26    std::cout << '\n';
27    for (const auto& value : std::views::values(freqWord)) {
28                               std::cout << value << " ";
29                             }
30
31  }
```

Of course, the keys and values can be displayed directly (lines 19 and 27). The output is identical.

```
Keys:
fish cat dog tale wizard witch
fish cat dog tale wizard witch

Values:
23 34 4 45 33 25
23 34 4 45 33 25
```

Views on the keys and values of a `std::unordered_map`

Working directly on the container might be not so thrilling, but function composition and lazy evaluation are.

5.1.5 Function Composition

In the example `rangesComposition.cpp`, I use a `std::map` because the ordering of the keys is crucial.

Composition of views

```
1  // rangesComposition.cpp
2
3  #include <iostream>
4  #include <ranges>
5  #include <string>
6  #include <map>
7
8
9  int main() {
10
11   std::map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33},
12                                        {"tale", 45}, {"dog", 4},
13                                        {"cat", 34}, {"fish", 23} };
14
15   std::cout << "All words: ";
16   for (const auto& name : std::views::keys(freqWord)) { std::cout << name << " "; }
17
18   std::cout << '\n';
19
20   std::cout << "All words, reverses: ";
21   for (const auto& name : std::views::keys(freqWord)
22                         | std::views::reverse) { std::cout << name << " "; }
23
24   std::cout << '\n';
25
26   std::cout << "The first 4 words: ";
27   for (const auto& name : std::views::keys(freqWord)
28                         | std::views::take(4)) { std::cout << name << " "; }
29
30   std::cout << '\n';
31
32   std::cout << "All words starting with w: ";
33   auto firstw = [](const std::string& name){ return name[0] == 'w'; };
34   for (const auto& name : std::views::keys(freqWord)
35                         | std::views::filter(firstw)) { std::cout << name << " "; }
36
37   std::cout << '\n';
38
39 }
```

I'm only interested in the keys. I display all of them (line 15), all of them reversed (line 20), the first four (line 26), and the keys starting with the letter 'w' (line 32).

Finally, here is the output of the program.

```
All words: cat dog fish tale witch wizard
All words, reversed: wizard witch tale fish dog cat
The first 4 words: cat dog fish tale
All words starting with w: witch wizard
```

Composition of views

The pipe symbol `|` is syntactic sugar[10] for function composition. Instead of `C(R)`, you can write `R | C`. Consequently, the next three lines are equivalent.

Three syntactic forms of function composition

```
auto rev1 = std::views::reverse(std::views::keys(freqWord));
auto rev2 = std::views::keys(freqWord) | std::views::reverse;
auto rev3 = freqWord | std::views::keys | std::views::reverse;
```

5.1.6 Lazy Evaluation

`std::views::iota` is a range factory for creating a sequence of elements by successively incrementing an initial value. This sequence can be finite or infinite. The program `rangesIota.cpp` fills a `std::vector` with 10 `int`'s, starting with 0.

Using `std::views::iota` to fill a `std::vector`

```
1  // rangesIota.cpp
2
3  #include <iostream>
4  #include <numeric>
5  #include <ranges>
6  #include <vector>
7
8  int main() {
9
10     std::cout << std::boolalpha;
11
12     std::vector<int> vec;
13     std::vector<int> vec2;
14
15     for (int i: std::views::iota(0, 10)) vec.push_back(i);
16
17     for (int i: std::views::iota(0) | std::views::take(10)) vec2.push_back(i);
18
```

[10] https://en.wikipedia.org/wiki/Syntactic_sugar

```
19      std::cout << "vec == vec2: " << (vec == vec2) << '\n';
20
21      for (int i: vec) std::cout << i << " ";
22
23  }
```

The first `iota` call (line 15) creates all numbers from 0 to 9, incremented by 1. The second iota call (line 17) creates an infinite data stream, starting with 0, incremented by 1. `std::views::iota(0)` is lazy. I only get a new value if I ask for it. I ask for it ten times. Consequently, both vectors are identical.

```
vec == vec2: true
0 1 2 3 4 5 6 7 8 9
```

Using std::views::iota to fill a `std::vector`

Now, I want to solve a small challenge: finding the first 20 prime numbers starting with 1,000,000.

The first 20 prime numbers starting with 1'000'000

```
1   // rangesLazy.cpp
2
3   #include <iostream>
4   #include <ranges>
5
6
7   bool isPrime(int i) {
8       for (int j=2; j*j <= i; ++j){
9           if (i % j == 0) return false;
10      }
11      return true;
12  }
13
14  int main() {
15
16      std::cout << "Numbers from 1'000'000 to 1'001'000 (displayed each 100th): "
17                << '\n';
18      for (int i: std::views::iota(1'000'000, 1'001'000)) {
19          if (i % 100 == 0) std::cout << i << " ";
20      }
21
22      std::cout << "\n\n";
23
24      auto odd = [](int i){ return i % 2 == 1; };
25      std::cout << "Odd numbers from 1'000'000 to 1'001'000 (displayed each 100th): "
26                << '\n';
```

```
27      for (int i: std::views::iota(1'000'000, 1'001'000) | std::views::filter(odd)) {
28           if (i % 100 == 1) std::cout << i << " ";
29      }
30
31      std::cout << "\n\n";
32
33      std::cout << "Prime numbers from 1'000'000 to 1'001'000: " << '\n';
34      for (int i: std::views::iota(1'000'000, 1'001'000) | std::views::filter(odd)
35                                                         | std::views::filter(isPrime)) {
36          std::cout << i << " ";
37      }
38
39      std::cout << "\n\n";
40
41      std::cout << "20 prime numbers starting with 1'000'000: " << '\n';
42      for (int i: std::views::iota(1'000'000) | std::views::filter(odd)
43                                              | std::views::filter(isPrime)
44                                              | std::views::take(20)) {
45          std::cout << i << " ";
46      }
47
48      std::cout << '\n';
49
50  }
```

This is my iterative strategy:

- **line 18**: Of course, I don't know when I have 20 primes greater than 1000000. To be on the safe side, I create 1000 numbers. For obvious reasons, I displayed only each 100th.
- **line 27**: I'm only interested in the odd numbers; therefore, I remove the even numbers.
- **line 34**: Now, it's time to apply the next filter. The predicate `isPrime` (line 7) returns if a number is prime. As you can see in the following screenshot, I was too eager. I got 75 primes.
- **line 42**: Laziness is a virtue. I use `std::iota` as an infinite number factory, starting with 1000000 and ask precisely for 20 primes.

```
Numbers from 1'000'000 to 1'001'000 (displayed each 100th):
1000000 1000100 1000200 1000300 1000400 1000500 1000600 1000700 1000800 1000900

Odd numbers from 1'000'000 to 1'001'000 (displayed each 100th):
1000001 1000101 1000201 1000301 1000401 1000501 1000601 1000701 1000801 1000901

Prime numbers from 1'000'000 to 1'001'000:
1000003 1000033 1000037 1000039 1000081 1000099 1000117 1000121 1000133 1000151
1000159 1000171 1000183 1000187 1000193 1000199 1000211 1000213 1000231 1000249
1000253 1000273 1000289 1000291 1000303 1000313 1000333 1000357 1000367 1000381
1000393 1000397 1000403 1000409 1000423 1000427 1000429 1000453 1000457 1000507
1000537 1000541 1000547 1000577 1000579 1000589 1000609 1000619 1000621 1000639
1000651 1000667 1000669 1000679 1000691 1000697 1000721 1000723 1000763 1000777
1000793 1000829 1000847 1000849 1000859 1000861 1000889 1000907 1000919 1000921
1000931 1000969 1000973 1000981 1000999

20 prime numbers starting with 1'000'000:
1000003 1000033 1000037 1000039 1000081 1000099 1000117 1000121 1000133 1000151
1000159 1000171 1000183 1000187 1000193 1000199 1000211 1000213 1000231 1000249
```

The first 20 prime numbers, starting with 1,000,000

Pull Pipelines

Combining views using the `|` operator enables you to create robust pipelines. Due to the lazy evaluation of the ranges library, the pipelines operate in pull mode.

Lazy Pipelines in Pull Mode

```
for (int i: std::views::iota(1'000'000) | std::views::filter(odd)
                                        | std::views::filter(isPrime)
                                        | std::views::take(20)) {
      std::cout << i << " ";
   }
```

Pull mode means in the concrete case that the data sink `std::views::take(20)` ask for the next value. This request for a new value is delegated to `std::view::filter(isPrime)`, `std::view::filter(isPrime)` delegates it to `std::views::filter(odd)`, and `std::views::filter(odd)` delegates it to `std::views::iota(1'000'000)`. Finally, the data source `std::views::iota(1'000'000)` produces the next value and puts it into the pipeline.

Conceptually, the workflow of the pipeline could also be started by the data source `std::views::iota(1'000'000)`. Such a pipeline models a push mode and applies eager evaluation. Eager evaluation has a disastrous outcome if applied to an infinite data source such as `std::views::iota(1'000'000)`. Before the subsequent view `std::view::filter(odd)` gets its first value, the data source eagerly produces all values.

5.1.7 Define a View

You can define your view.

5.1.7.1 `std::ranges::view_interface`

Thanks to the `std::ranges::view_interface` helper class, defining a view is easy. To fulfill the concept view, your view needs at least a default constructor, and member functions `begin()` and `end()`:

Your own view

```
class MyView : public std::ranges::view_interface<MyView> {
public:
    auto begin() const { /*...*/ }
    auto end() const { /*...*/ }
};
```

By deriving `MyView` public from the helper class `std::ranges::view_interface` using itself as a template parameter, `MyView` becomes a view. This technique of class template having itself as a template parameter is called Curiously Recurring Template Pattern[11] (short CRTP).

I use this technique in the next example to create a view out of a container of the Standard Template Library.

5.1.7.2 A Container View

The view `ContainerView` creates a view on an arbitrary container.

Creating a view from a container

```
1  // containerView.cpp
2
3  #include <iostream>
4  #include <ranges>
5  #include <string>
6  #include <vector>
7
8  template<std::ranges::input_range Range>
9  requires std::ranges::view<Range>
10 class ContainerView : public std::ranges::view_interface<ContainerView<Range>> {
11 private:
12    Range range_{};
13    std::ranges::iterator_t<Range> begin_{ std::begin(range_) };
14    std::ranges::iterator_t<Range> end_{ std::end(range_) };
```

[11] https://www.modernescpp.com/index.php/c-is-still-lazy

```
15
16  public:
17     ContainerView() = default;
18
19     constexpr ContainerView(Range r): range_(std::move(r)) ,
20                                       begin_(std::begin(r)), end_(std::end(r)) {}
21
22     constexpr auto begin() const {
23        return begin_;
24     }
25     constexpr auto end() const {
26        return end_;
27     }
28  };
29
30  template<typename Range>
31  ContainerView(Range&& range) -> ContainerView<std::ranges::views::all_t<Range>>;
32
33  int main() {
34
35     std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
36
37     auto myContainerView = ContainerView(myVec);
38     for (auto c : myContainerView) std::cout << c << " ";
39     std::cout << '\n';
40
41     for (auto i : std::views::reverse(ContainerView(myVec))) std::cout  << i <<  ' ';
42     std::cout << '\n';
43
44     for (auto i : ContainerView(myVec) | std::views::reverse) std::cout  << i <<  ' ';
45     std::cout << '\n';
46
47     std::cout << '\n';
48
49     std::string myStr = "Only for testing purpose.";
50
51     auto myContainerView2 = ContainerView(myStr);
52     for (auto c: myContainerView2) std::cout << c << " ";
53     std::cout << '\n';
54
55     for (auto i : std::views::reverse(ContainerView(myStr))) std::cout << i << ' ';
56     std::cout << '\n';
57
58     for (auto i : ContainerView(myStr) | std::views::reverse) std::cout  << i <<  ' ';
59     std::cout << '\n';
```

```
60
61  }
```

The class template `ContainerView` (line 8) derives from the helper class `std::ranges::view_interface` and requires that the container support the concept `std::ranges::view` (line 9). The remaining, minimal implementation is straightforward. `ContainerView` has a default constructor (line 17). Due to a change in the C++20 standard for a view, this default constructor is not necessary anymore, but many compilers still require it. On the contrary, the two member functions `begin()` (line 22) and `end()` (line 25) are required. Initially, a view must be default constructible. This requirement was removed, but many compilers still require a default constructor. Therefore, `ContainerView` has one. For convenience, I added a user-defined deduction guide for class template argument deduction (line 32).

In the `main` function, I apply the `ContainerView` on a `std::vector` (line 37) and a `std::string` (line 49) and iterate through them forwards and backward.

```
1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1

O n l y  f o r  t e s t i n g  p u r p o s e .
. e s o p r u p  g n i t s e t  r o f  y l n O
. e s o p r u p  g n i t s e t  r o f  y l n O
```

Creating a view from a container

Let me add a few words to the class template argument deduction guide.

Class Template Argument Deduction Guide

Since C++17, the compiler can deduce template parameters from template arguments. The template deduction guide is a pattern for the compiler to deduce the template arguments.

When you use `ContainerView(myVec)`, the compiler applies the following user-defined deduction guide:

User-Defined Deduction Guide for `ContainerView`

```
template<class Range>
ContainerView(Range&& range) -> ContainerView<std::ranges::views::all_t<Range>>;
```

Essentially, a call `Container(myVec)` causes the compiler to instantiate the code on the right of the arrow `->`:

Applying the deduction guide for `Container(myVec)`

```
ContainerView<std::ranges::views::all_t<std::vector<int>&>>(myVec);
```

cppreference.com[12] provides more information to the user-defined deduction guide for class templates.

5.1.7.3 A `TrimByView`

When you study the previous example `containerView.cpp`, you notice that the member functions `begin` and `end` are crucial for implementing a new view. The following view, `TrimByView`, adjusts the view's boundaries by ignoring its range's beginning and trailing `count` elements.

Creating a view from a container excluding the beginning and trailing `count` elements

```
1  // trimByView.cpp
2
3  #include <iostream>
4  #include <ranges>
5  #include <string>
6  #include <vector>
7
8  template<std::ranges::input_range Range>
9  requires std::ranges::view<Range>
10 class TrimByView : public std::ranges::view_interface<TrimByView<Range>> {
11 private:
12    Range range_{};
13    std::ranges::iterator_t<Range> begin_{ std::begin(range_) };
14    std::ranges::iterator_t<Range> end_{ std::end(range_) };
```

[12] https://en.cppreference.com/w/cpp/language/class_template_argument_deduction

```
15     std::size_t count{};
16
17  public:
18     TrimByView() = default;
19
20     constexpr TrimByView(Range r, std::size_t cnt): range_(std::move(r)) ,
21                                     begin_(std::begin(r)), end_(std::end(r)),
22                                     count(cnt) {}
23
24     constexpr auto begin() const {
25        return begin_ + count;
26     }
27     constexpr auto end() const {
28        return end_ - count;
29     }
30  };
31
32  template<typename Range>
33  TrimByView(Range&& range, std::size_t&& cnt) ->
34                                     TrimByView<std::ranges::views::all_t<Range>>;
35
36  int main() {
37
38    std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
39
40     auto myTrimByView1 = TrimByView(myVec, 1);
41     for (auto c : myTrimByView1) std::cout << c << " ";
42     std::cout << '\n';
43
44     for (auto i : std::views::reverse(TrimByView(myVec, 2))) std::cout  << i <<  ' ';
45     std::cout << '\n';
46
47     for (auto i : TrimByView(myVec, 3) | std::views::reverse) std::cout  << i <<  ' ';
48     std::cout << '\n';
49
50     std::cout << '\n';
51
52     std::string myStr = "Only for testing purpose.";
53
54     auto myTrimByView2 = TrimByView(myStr, 1);
55     for (auto c: myTrimByView2) std::cout << c << " ";
56     std::cout << '\n';
57
58     for (auto i : std::views::reverse(TrimByView(myStr, 2))) std::cout << i << ' ';
59     std::cout << '\n';
```

```
60
61      for (auto i : TrimByView(myStr, 3) | std::views::reverse) std::cout  << i <<  ' ';
62      std::cout << '\n';
63
64  }
```

The `TrimByView` is almost identical to the previous `ContainerView`. The difference is that the container of `TrimByView` needs the count (line 20), and the `begin` (line 24) and `end` member functions (line 27) are adjusted by `count`. Consequentially, `myTrimByView1` (line 40) is a view of the container excluding its first and last elements. The views in lines 44 and 47 are similar. They ignore the two beginning and trailing elements (line 44), and the three beginning and trailing elements (line 47). The according to argumentation holds for the following views (lines 54, 58, and 61) on the string `myStr`.

```
2 3 4 5 6 7 8
7 6 5 4 3
6 5 4

n l y   f o r   t e s t i n g   p u r p o s e
s o p r u p   g n i t s e t   r o f   y l
o p r u p   g n i t s e t   r o f   y
```

Creating a view from a container ignoring the `count` beginning and trailing elements

5.1.8 `std` Algorithms versus `std::ranges` Algorithms

The algorithms of the algorithm library[13] and the memory libray[14] have ranges pendants. They start with the namespace `std::ranges`. The numeric library[15] does not have a ranges pendant. Now, you may have the question: Should I use the classical `std` algorithm or the new `std::ranges` algorithm?

Let me start with a comparison of the classical `std::sort` and the new `std::ranges::sort`. First, here are the various overloads of `std::sort` and `std::range::sort`.

[13] https://en.cppreference.com/w/cpp/header/algorithm
[14] https://en.cppreference.com/w/cpp/header/memory
[15] https://en.cppreference.com/w/cpp/header/numeric

```
1  template< class RandomIt >
2  constexpr void sort( RandomIt first, RandomIt last );
3
4  template< class ExecutionPolicy, class RandomIt >
5  void sort( ExecutionPolicy&& policy,
6             RandomIt first, RandomIt last );
7
8  template< class RandomIt, class Compare >
9  constexpr void sort( RandomIt first, RandomIt last, Compare comp );
10
11 template< class ExecutionPolicy, class RandomIt, class Compare >
12 void sort( ExecutionPolicy&& policy,
13            RandomIt first, RandomIt last, Compare comp );
```

`std::sort` has four overloads in C++20. Let's see what I can deduce from the names of the function declarations. All four overloads take a range, given by a begin and end iterator. The iterators must be random access iterators. The first and third overloads (lines 1 and 8) are declared as `constexpr` and can run at compile time. The second and fourth overloads (lines 4 and 11) require an execution policy[16]. The execution policy lets you specify if the program should run sequentially, parallel, or vectorized. Additionally, the last two overloads (lines 8 and 11) let you specify the sorting strategy. `Compare` has to be a binary predicate. A binary predicate is a callable that takes two arguments and returns something convertible to a `bool`.

I assume my analysis reminded you of concepts. But there is a big difference. The names in the `std::sort` do not stand for concepts but only for documentation purposes. In `std::ranges::sort` the names are concepts.

```
1  template <std::random_access_iterator I, std::sentinel_for<I> S,
2           class Comp = ranges::less, class Proj = std::identity>
3  requires std::sortable<I, Comp, Proj>
4  constexpr I sort(I first, S last, Comp comp = {}, Proj proj = {});
5
6  template <ranges::random_access_range R, class Comp = ranges::less,
7            class Proj = std::identity>
8  requires std::sortable<ranges::iterator_t<R>, Comp, Proj>
9  constexpr ranges::borrowed_iterator_t<R> sort(R&& r, Comp comp = {}, Proj proj = {});
```

When you study the two overloads, you notice that it takes a sortable range `R` (lines 3 and 8), either given by a begin iterator and end sentinel (line 1) or by a `ranges::random_access_range` (line 6). The iterator and the range must support random access. Additionally, the overloads take a predicate `Comp`, and a projection `Proj`. The predicate `Comp` uses for default less, and the projection `Proj` the identity

[16] https://www.modernescpp.com/index.php/parallel-algorithms-of-the-stl-with-gcc

std::identity[17]. A projection is a mapping of a set into a subset. std::ranges::sort does not support execution policies[18].

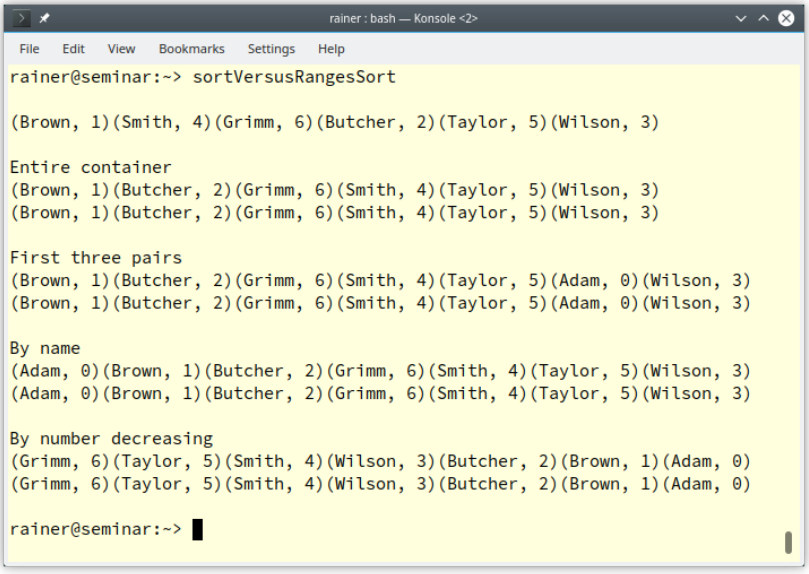
From the practical point of view, let me show the difference between std::sort and std::ranges::sort. I ignore in my comparison the execution policy.

```
// sortVersusRangesSort.cpp

#include <algorithm>
#include <functional>
#include <iostream>
#include <utility>
#include <vector>

struct PhoneBookEntry{
    std::string name;
    int number;
    auto operator<=>(const PhoneBookEntry&) const = default;
};

void printPhoneBook(const std::vector<PhoneBookEntry>& phoneBook) {
    for (const auto& entry: phoneBook) std::cout << "(" << entry.name << ", "
                                                 << entry.number << ")";
    std::cout << "\n";
}

int main() {

    std::cout << '\n';

    std::vector<PhoneBookEntry> phoneBook{ {"Brown", 1}, {"Smith", 4},
    {"Grimm", 6}, {"Butcher", 2}, {"Taylor", 5}, {"Wilson", 3} };

    printPhoneBook(phoneBook);

    std::cout << '\n';

    std::cout << "Entire container\n";
    std::sort(phoneBook.begin(), phoneBook.end());
    printPhoneBook(phoneBook);
    std::ranges::sort(phoneBook.begin(), phoneBook.end());
    printPhoneBook(phoneBook);

```

[17] https://en.cppreference.com/w/cpp/utility/functional/identity
[18] https://www.modernescpp.com/index.php/parallel-algorithms-of-the-stl-with-gcc

```
38      phoneBook.insert(phoneBook.begin() + 5, {"Adam", 0});
39
40      std::cout << "\nFirst three pairs\n";
41      std::sort(phoneBook.begin(), phoneBook.begin() + 3);
42      printPhoneBook(phoneBook);
43      std::ranges::sort(phoneBook.begin(), phoneBook.begin() + 3);
44      printPhoneBook(phoneBook);
45
46      std::cout << "\nBy name\n";
47      std::sort(phoneBook.begin(), phoneBook.end(),[](auto p, auto p2) {
48          return p.name < p2.name;
49      });
50      printPhoneBook(phoneBook);
51      std::ranges::sort(phoneBook.begin(), phoneBook.end(), {}, &PhoneBookEntry::name);
52      printPhoneBook(phoneBook);
53
54      std::cout << "\nBy number decreasing\n";
55      std::sort(phoneBook.begin(), phoneBook.end(),[](auto p, auto p2) {
56          return p.number > p2.number;
57      });
58      printPhoneBook(phoneBook);
59      std::ranges::sort(phoneBook.begin(), phoneBook.end(), std::ranges::greater(),
60                        &PhoneBookEntry::number);
61      printPhoneBook(phoneBook);
62
63      std::cout << '\n';
64
65  }
```

From the practical point of view, there is no significant difference. `std::sort` (lines 33, 41, 47, and 55) must be invoked with a begin and end iterator, and `std::ranges::sort` (lines 35, 43, 51, and 59) can be invoked with a begin and end iterator. For convenience, I overloaded the three-way comparison operator (line 12). The first sort operation happens on the entire container (lines 33 and 35), the second on the first three elements (lines 41 and 43), the third only on the names (lines 47 and 51), and the last one on the numbers in decreasing order (lines 55 and 59). The main difference is that in `std::ranges::sort` you can express what you sort and how you sort. What is provided by the projection (`&PhoneBookEntry::number`), and how by the sorting criteria (`std::ranges::greater()`) (line 59)? In contrast, the lambda function in line 55 implements the projection and the sorting criteria. `std::sort`. Both algorithms return the same results:



`std::sort` versus `std::ranges::sort`

So far, it may not convince you to prefer `std::ranges::sort` about `std::sort`. So, let me write about the differences and start with concepts.

5.1.8.1 Concepts

What happens when you invoke `std::sort` or `std::ranges::sort` with a container only supporting a bidirectional iterator?

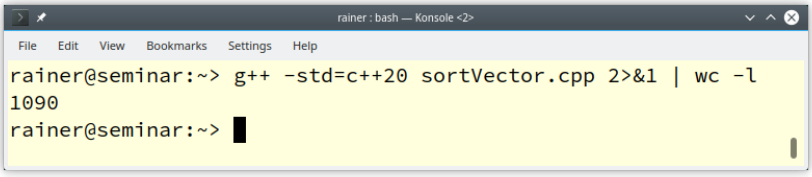
- `std::sort`

A `std::list` as a doubly-linked list provides a bidirectional iterator but no random-access iterator.

Applying `std::sort` on a `std::list`

```
// sortVector.cpp

#include <algorithm>
#include <list>

int main() {

   std::list<int> myList{1, -5, 10, 20, 0};
   std::sort(myList.begin(), myList.end());

}
```

Compiling the program `sortVector.cpp` causes an epic error message of 1090 lines.



`std::sort`: number of error lines using GCC

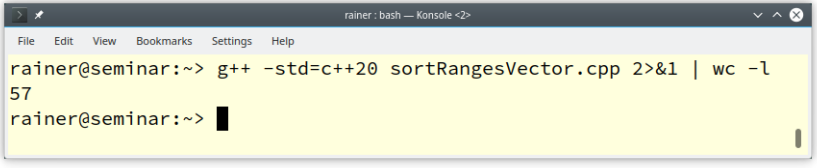
- `std::ranges::sort`

The modification to the previous program `sortVector.cpp` are minimal. Simply `std::sort` has to be replaced with `std::ranges::sort`.

Applying `std::ranges::sort` on a `std::list`

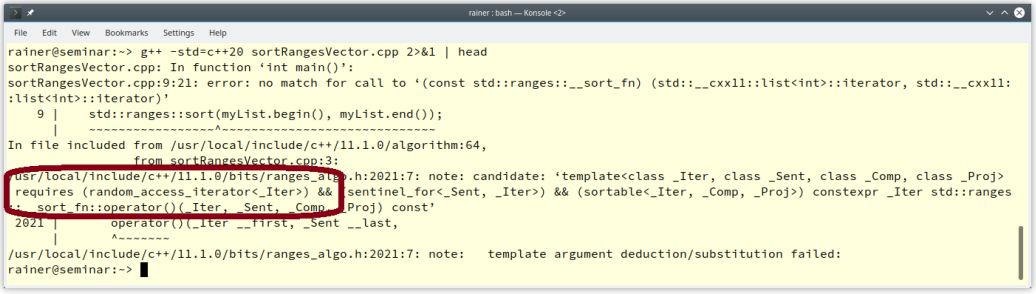
```
// sortRangesVector.cpp

#include <algorithm>
#include <list>

int main() {

   std::list<int> myList{1, -5, 10, 20, 0};
   std::ranges::sort(myList.begin(), myList.end());

}
```

Using `std::ranges::sort` instead of `std::sort` reduces the error message drastically. Now, I get 57 error lines.



`std::ranges::sort`: number of error lines using GCC

Honestly, the error message of GCC should be easier to read. Here are the first ten lines of the 57 lines. I marked the critical message in red.



The first ten error lines of GCC

The next issue is more subtle.

5.1.8.2 Unified Lookup Rules

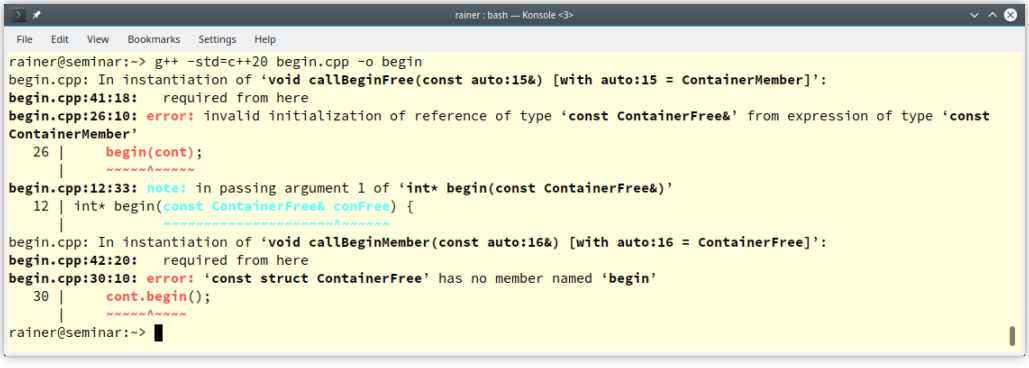
Assume you want to implement a generic function that calls `begin` on a given container. The question is if the function call `begin` on a container should assume a free `begin` function or a member function `begin`.

A free `begin` function versus a member function `begin`

```
1  // begin.cpp
2
3  #include <cstddef>
4  #include <iostream>
5  #include <ranges>
6
7  struct ContainerFree {
8      ContainerFree(std::size_t len): len_(len), data_(new int[len]){}
9      size_t len_;
10     int* data_;
11 };
12 int* begin(const ContainerFree& conFree) {
13     return conFree.data_;
14 }
15
16 struct ContainerMember {
17     ContainerMember(std::size_t len): len_(len), data_(new int[len]){}
18     int* begin() const {
19         return data_;
20     }
21     size_t len_;
22     int* data_;
23 };
24
```

```
25  void callBeginFree(const auto& cont) {
26      begin(cont);
27  }
28
29  void callBeginMember(const auto& cont) {
30      cont.begin();
31  }
32
33  int main() {
34
35      const ContainerFree contFree(2020);
36      const ContainerMember contMemb(2023);
37
38      callBeginFree(contFree);
39      callBeginMember(contMemb);
40
41      callBeginFree(contMemb);
42      callBeginMember(contFree);
43
44  }
```

`ContainerFree` (line 7) has a free function `begin` (line 12), and `ContainerMember` (line 16) has a member function `begin` (line 18). Accordingly, `contFree` can use the generic function `callBeginFree` using the free function call `begin(cont)` (line 26), and `contMemb` can use the generic function `callBeginMember` using the member function call `cont.begin` (line 30). When I invoke `callBeginFree` and `callBeginMember` with the inappropriate containers in lines 41 and 42, the compilation fails.



Compilation error if using the wrong `begin` implementation

I can solve this issue by providing two different `begin` implementations in two ways: classical and range based.

A free `begin` function versus a member function `begin`

```cpp
// beginSolved.cpp

#include <cstddef>
#include <iostream>
#include <ranges>

struct ContainerFree {
    ContainerFree(std::size_t len): len_(len), data_(new int[len]){}
    size_t len_;
    int* data_;
};
int* begin(const ContainerFree& conFree) {
    return conFree.data_;
}

struct ContainerMember {
    ContainerMember(std::size_t len): len_(len), data_(new int[len]){}
    int* begin() const {
        return data_;
    }
    size_t len_;
    int* data_;
};

void callBeginClassical(const auto& cont) {
    using std::begin;
    begin(cont);
}

void callBeginRanges(const auto& cont) {
    std::ranges::begin(cont);
}

int main() {

    const ContainerFree contFree(2020);
    const ContainerMember contMemb(2023);

    callBeginClassical(contFree);
    callBeginRanges(contMemb);

    callBeginClassical(contMemb);
    callBeginRanges(contFree);
```

```
44
45 }
```

The classical way to solve this issue is to bring `std::begin` into the scope with a so-called using declaration (line 26). Thanks to ranges, you can directly use `std::ranges::begin` (line 31). `std::ranges::begin` considers both implementations of `begin`: the free version and the member function.

Finally, let me write about safety.

5.1.8.3 Safety

The ranges library provides the expected operation to iterate through or access the range directly. They need the header `<ranges>`.

Iterators

Operation	Description
`std::ranges::begin`	Returns an iterator to the beginning of the range
`std::ranges::end`	Returns a sentinel indicating the end of the range
`std::ranges::cbegin`	Returns an iterator to the beginning of the read-only range
`std::ranges::cend`	Returns a sentinel indicating the end of the read-only range
`std::ranges::rbegin`	Returns a reverse iterator to the end of the range
`std::ranges::rend`	Returns a sentinel (reverse) indicating the beginning of the range
`std::ranges::crbegin`	Returns a reverse iterator to the end of the read-only range
`std::ranges::crend`	Returns a sentinel (reverse) indicating the beginning of the read-only range
`std::ranges::data`	Returns a pointer to the beginning of the contiguous range
`std::ranges::cdata`	Returns a pointer to the beginning of the contiguous read-only range

When you use these operations to access the underlying range, there's a big difference. The compilation fails when you use the range access on the `std::rangess`' variant if the argument is an rvalue[19]. On the contrary, using the same operation from the classical `std` namespace is undefined

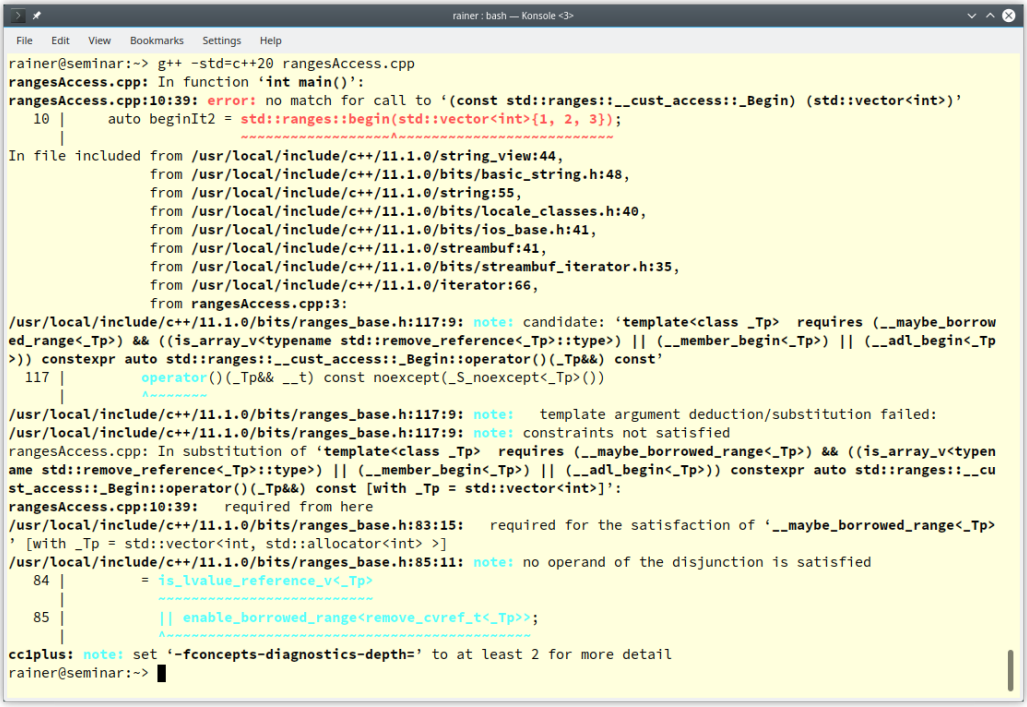
[19] https://en.cppreference.com/w/cpp/language/value_category

behavior.

```
1  // rangesAccess.cpp
2
3  #include <iterator>
4  #include <ranges>
5  #include <vector>
6
7  int main() {
8
9      auto beginIt1 = std::begin(std::vector<int>{1, 2, 3});
10     auto beginIt2 = std::ranges::begin(std::vector<int>{1, 2, 3});
11
12 }
```

`std::ranges::begin` provides only overloads for lvalues[20]. The temporary vector `std::vector<int>{1, 2, 3}` (line 10) is an rvalue[21]. Consequentially, the compilation of the program fails.

[20] https://en.cppreference.com/w/cpp/language/value_category
[21] https://en.cppreference.com/w/cpp/language/value_category



`std::ranges::begin` causes a compilation error on a temporary

You can also ask a range for its emptiness and size. They accept lvalues and rvalues.

Emptiness and size

Operation	Description
`std::ranges::empty`	Checks if the range is empty
`std::ranges::size`	Returns an integral equal to the size of the range
`std::ranges::ssize`	Returns a signed integral equal to the size of the range

Furthermore, ranges support comparison. They are defined in the header `<functional>`

Comparison

Operation	Description
`std::ranges::equal_to(fir, sec)`	Returns whether `fir` is equal to `sec`
`std::ranges::not_equal_to(fir, sec)`	Returns whether `fir` is not equal to `sec`
`std::ranges::less(fir, sec)`	Returns whether `fir` is less than `sec`
`std::ranges::greater(fir, sec)`	Returns whether `fir` is greater than `sec`
`std::ranges::less_equal(fir, sec)`	Returns whether `fir` is less than or equal to `sec`
`std::ranges::greater_equal(fir, sec)`	Returns whether `fir` is greater than or equal to `sec`

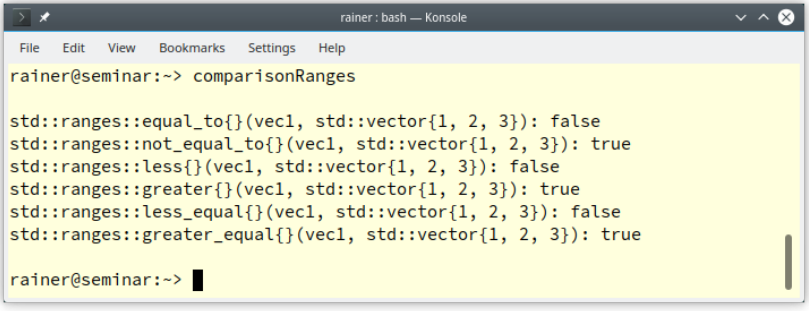
The comparators can operate on lvalues and rvalues. To use this comparators, you have to instantiate them.

Comparison of ranges

```
// comparisonRanges.cpp

#include <iostream>
#include <ranges>
#include <functional>
#include <vector>

int main() {

    std::cout << std::boolalpha << '\n';

    auto vec1 = std::vector{1, 2, 3, 4};

    std::cout << "std::ranges::equal_to{}(vec1, std::vector{1, 2, 3}): "
              << std::ranges::equal_to{}(vec1, std::vector{1, 2, 3}) << '\n';

    std::cout << "std::ranges::not_equal_to{}(vec1, std::vector{1, 2, 3}): "
              << std::ranges::not_equal_to{}(vec1, std::vector{1, 2, 3}) << '\n';

    std::cout << "std::ranges::less{}(vec1, std::vector{1, 2, 3}): "
              << std::ranges::less{}(vec1, std::vector{1, 2, 3}) << '\n';

    std::cout << "std::ranges::greater{}(vec1, std::vector{1, 2, 3}): "
              << std::ranges::greater{}(vec1, std::vector{1, 2, 3}) << '\n';
```

```
    std::cout << "std::ranges::less_equal{}(vec1, std::vector{1, 2, 3}): "
              << std::ranges::less_equal{}(vec1, std::vector{1, 2, 3}) << '\n';

    std::cout << "std::ranges::greater_equal{}(vec1, std::vector{1, 2, 3}): "
              << std::ranges::greater_equal{}(vec1, std::vector{1, 2, 3}) << '\n';

    std::cout << std::boolalpha << '\n';

}
```

The following screenshot shows the output of the program.



Comparison of ranges

The ranges library made a few unique design choices.

5.1.9 Design Choices

For efficiency reasons, the ranges library has some unique design choices. It's important to know and follow these rules.

When you study `begin` member function of `std::ranges::filter_view`, you find code equivalent to the following one:

Definition of `std::ranges::filter_view::begin`

```
1  if constexpr (!ranges::forward_range<V>)
2      return /* iterator */{*this, ranges::find_if(base_, std::ref(*pred_))};
3  else
4  {
5      if (!begin_.has_value())
6          begin_ = ranges::find_if(base_, std::ref(*pred_)); // caching
7      return /* iterator */{*this, begin_.value()};
8  }
```

Let's analyze lines 5 - 7. First, the compiler checks if `begin_.has_value()` is true. If not, it determines `begin_`. This means that this member function caches the result within the `std::ranges::filter_view` object for use on subsequent calls. This caching has serious consequences. Let me exemplify this with a code snippet.

Efficiency of `std::views::filter`

```
1  // cachingRanges.cpp
2
3  #include <numeric>
4  #include <iostream>
5  #include <ranges>
6  #include <vector>
7
8  int main() {
9
10     std::vector<int> vec(1'000'000);
11     std::iota(vec.begin(), vec.end(), 0);
12
13     for (int i: vec | std::views::filter([](auto v) { return v > 1000; })
14                     | std::views::take(5)) {
15                         std::cout << i << " ";  // 1001 1002 1003 1004 1005
16                     }
17
18 }
```

The first call of `std::views::filter([](auto v) { return v > 1000; })` determines the begin iterator and reuses it in subsequent calls. The benefit of this caching is obvious. Many subsequent iterations of the pipeline are spared. But there are also severe drawbacks: cache issues and constness issues.

5.1.9.1 Cache

Here are the two important cache rules for ranges:

- Don't use a view on modified ranges.
- Don't copy a view.

Let me play with the previous program `cachingRanges.cpp` and break both rules:

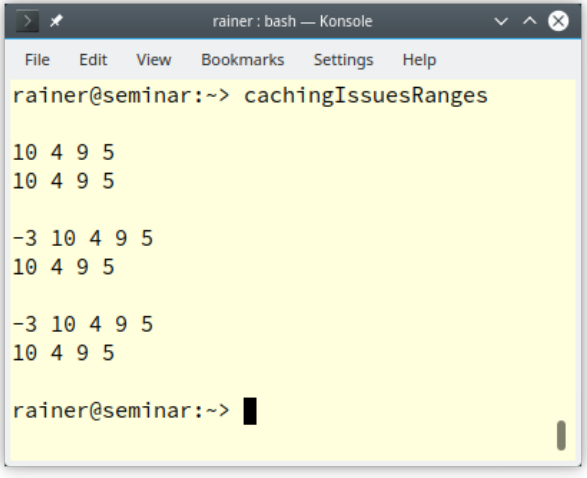
Efficiency of `std::views::filter`

```
// cachingIssuesRanges.cpp

#include <concepts>
#include <forward_list>
#include <iostream>
#include <numeric>
#include <ranges>
#include <vector>

void printElements(std::ranges::input_range auto&& rang) {
    for (int i: rang) {
        std::cout << i << " ";
    }
    std::cout << '\n';
}

int main() {

    std::cout << '\n';

    std::vector<int> vec{-3, 10, 4, -7, 9, 0, 5, -5};
    std::forward_list<int> forL{-3, 10, 4, -7, 9, 0, 5, -5};

    auto first5Vector = vec | std::views::filter([](auto v) { return v > 0; })
                            | std::views::take(5);

    auto first5ForList = forL | std::views::filter([](auto v) { return v > 0; })
                              | std::views::take(5);

    printElements(first5Vector);          // 10 4 9 5
    printElements(first5ForList);         // 10 4 9 5

    std::cout << '\n';

    vec.insert(vec.begin(), 10);
    forL.insert_after(forL.before_begin(), 10);


    printElements(first5Vector);          // -3 10 4 9 5
```

```
40      printElements(first5ForList);          // 10 4 9 5
41
42      std::cout << '\n';
43
44      auto first5VectorCopy{first5Vector};
45      auto first5ForListCopy{first5ForList};
46
47      printElements(first5VectorCopy);       // -3 10 4 9 5
48      printElements(first5ForListCopy);      // 10 10 4 9 5
49
50      std::cout << '\n';
51
52  }
```

To make it easier to follow the problem, I wrote the output directly in the source code. The program does the following steps with a `std::vector` and a `std::forward_list`. First, both containers are initialized with the initializer list `{-3, 10, 4, -7, 9, 0, 5, -5}` (lines 21 and 22). Then, I create two views (lines 24 and 27). Both views `first5Vector` and `first5ForList` consist of the first `5` elements greater than `0`. Lines 30 and 31 display the corresponding values.

Now, I break the first rule: "Don't use a view on modified ranges." I insert `10` at the beginning of both containers. Afterward, `first5Vector` displays the `-3` and `first5ForList` ignores the added `10`. After the break of the second rule, "Don't copy a view." in lines 44 and 45, the cache of `first5ForListCopy` is invalidated. `first5VectorCopy` still shows the wrong numbers. Finally, here is the output of the program.



Caching issues with views

Here is a simple rule of thumb: **Use views directly after you have defined them**.

You may have noticed that the function `printElements` takes it arguments by universal reference, aka forwarding reference.

5.1.9.2 Constness

The member function of a view may cache the position. This has two interesting consequences:

- A function taking an arbitrary view should take its arguments by universal reference.
- Reading two views concurrently may be a data race.

Let's discuss the first consequence.

5.1.9.2.1 Take Arbitrary Views by Universal Reference

The previous function `printElemets` takes its view by universal reference.

print of an arbitrary view

```
void printElements(std::ranges::input_range auto&& rang) {
    for (int i: rang) {
        std::cout << i << " ";
    }
    std::cout << '\n';
}
```

`print` takes its argument by universal reference. Taking it by lvalue reference or by value is, in general, no option.

Taking the argument by const lvalue reference may not work because the implicitly `begin` call on the view could modify it. On the contrary, a non-const lvalue reference cannot handle rvalues.

Taking the argument by value may invalidate the cache.

5.1.9.2.2 Concurrent Reading Access of Views

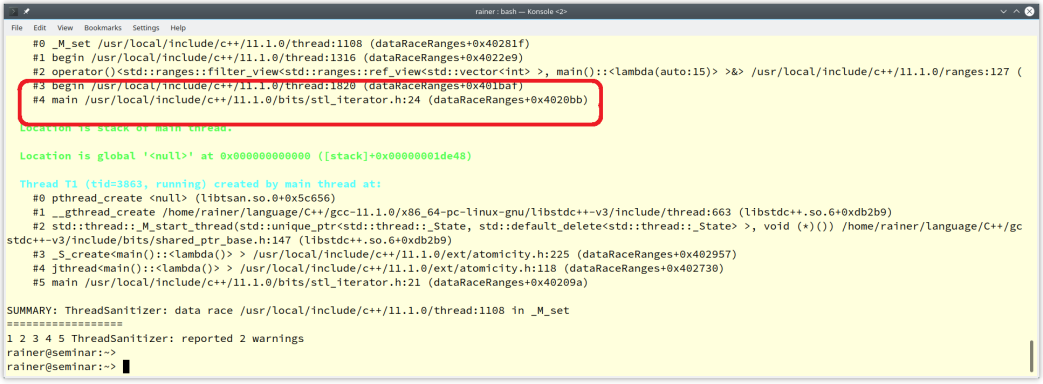
The following program exemplifies the concurrency issue with views:

A data race on views

```
1  // dataRaceRanges.cpp
2
3  #include <numeric>
4  #include <iostream>
5  #include <ranges>
6  #include <thread>
7  #include <vector>
8
9  int main() {
10
11     std::vector<int> vec(1'000);
12     std::iota(vec.begin(), vec.end(), 0);
13
14     auto first5Vector = vec | std::views::filter([](auto v) { return v > 0; })
15                             | std::views::take(5);
16
17     std::jthread thr1([&first5Vector]{
18         for (int i: first5Vector) {
19             std::cout << i << " ";
20         }
21     });
22
23
24     for (int i: first5Vector) {
25         std::cout << i << " ";
26     }
27
28     std::cout << "\n\n";
29
30 }
```

In the program `dataRaceRanges.cpp`, I iterate concurrency two times through a view in a non-modifying way. First, I iterate in the `std::jthread thr1` (line 17) and second in the main function (line 24). This is a data race because both iterations implicitly use the member function `begin`, which may cache the position. ThreadSanitizer[22] visualizes this data race and complains that there is a previous write on line 24.

[22] https://clang.llvm.org/docs/ThreadSanitizer.html



A data race on views

On the contrary, iterating through a classical range such as `std::vector` is thread-safe. There is an additional difference between classical ranges and views.

5.1.9.2.3 Propagation of Const

Classical ranges model deep constness. They propagate their constness to their elements. This means that modifying elements of a constant container is impossible.

Const propagation of a `std::vector`

```
1  // constPropagationContainer.cpp
2
3  #include <iostream>
4  #include <vector>
5
6  template <typename T>
7  void modifyConstRange(const T& cont) {
8      cont[0] = 5;
9  }
10
11 int main() {
12
13     std::vector myVec{1, 2, 3, 4, 5};
14     modifyConstRange(myVec);          // ERROR
15
16 }
```

The call `modifyConstRange(myVec)` causes a compile-time error.

On the contrary, views model shallow constness. They do not propagate the constness to their elements. They can still be modified.

Const propagation of a `std::vector`

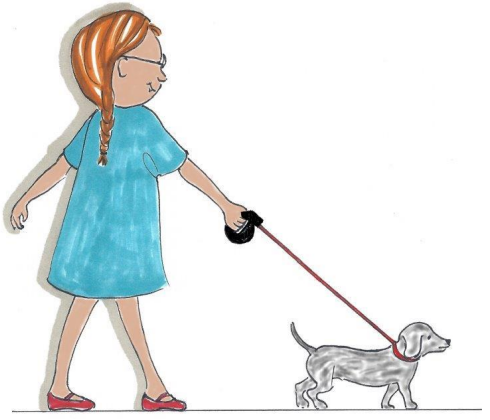
```
// constPropagationViews.cpp

#include <iostream>
#include <ranges>
#include <vector>

template <typename T>
void modifyConstRange(const T& cont) {
    cont[0] = 5;
}

int main() {

    std::vector myVec{1, 2, 3, 4, 5};

    modifyConstRange(std::views::all(myVec));   // OK

}
```

The call `modifyConstRange(std::views::all(myVec))` is fine.

Distilled Information

- The ranges library provides us with an additional version of the STL algorithms that operate on ranges. A range is a group of items you can iterate over. The range is typically given by two iterators, an iterator and a size, or C-array.
- The algorithms of the ranges library
  - are lazy and can, therefore, be invoked on infinite data streams.
  - can operate directly on the container.
  - can be composed using the pipe (`|`) symbol.
  - support projections to address only a subset of processed items.
- Views implement some unique design choices. They may cache their begin iterator and can, in general, not be declared as `const`.

5.2 std::span



Cippi walks the dog

A std::span represents an object that refers to a contiguous sequence of objects. A std::span, sometimes also called a view, is never an owner. This contiguous sequence of objects can be a plain C-array, a pointer with a size, a std::array, a std::vector, or a std::string.

A std::span can have a *static extent* or a *dynamic extent*. By default, std::span has a *dynamic extent*:

Definition of std::span

```
template <typename T, std::size_t Extent = std::dynamic_extent>
class span;
```

5.2.1 Static versus Dynamic Extent

When a std::span has a ***static extent***, its size is known at compile time and part of the type: std::span<T, size>. Consequently, its implementation needs only a pointer to the first element of the contiguous sequence of objects.

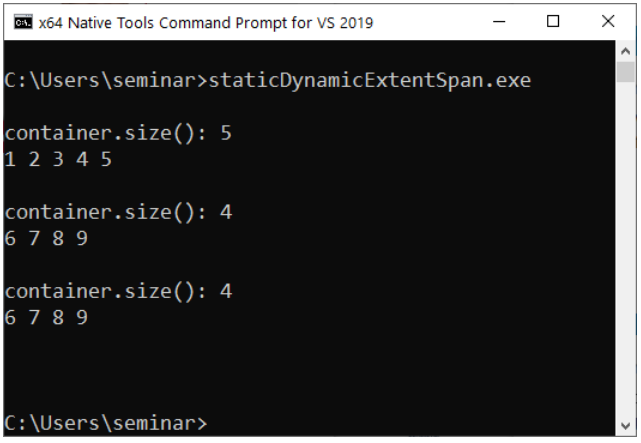
Implementing a std::span with a ***dynamic extent*** consists of a pointer to the first element and the size of the contiguous sequence of objects. The size is not part of the std::span<T> type.

The next example, staticDynamicExtentSpan.cpp, emphasizes the differences between the two kinds of ranges.

std::spans with static and dynamic extent

```
1  // staticDynamicExtentSpan.cpp
2
3  #include <iostream>
4  #include <span>
5  #include <vector>
6
7  void printMe(std::span<int> container) {
8
9      std::cout << "container.size(): " << container.size() << '\n';
10     for (auto e : container) std::cout << e << ' ';
11     std::cout << "\n\n";
12 }
13
14 int main() {
15
16     std::cout << '\n';
17
18     std::vector myVec1{1, 2, 3, 4, 5};
19     std::vector myVec2{6, 7, 8, 9};
20
21     std::span<int> dynamicSpan(myVec1);
22     std::span<int, 4> staticSpan(myVec2);
23
24     printMe(dynamicSpan);
25     printMe(staticSpan);  // implicitly converted into a dynamic span
26
27     // staticSpan = dynamicSpan; ERROR
28     dynamicSpan = staticSpan;
29
30     printMe(staticSpan);
31
32     std::cout << '\n';
33
34 }
```

`dynamicSpan` (line 21) has a dynamic extent, while `staticSpan` (line 22) has a static extent. Both `std::spans` return their size in the `printMe` function (line 9). A `std::span` with static extent can be assigned to a `std::span` with dynamic extent, but not vice versa. Line 27 would cause an error, but lines 7, 25, and 28 are valid.



std::spans with static and dynamic extent

Distinguish between std::span, std::ranges::range, std::ranges::view, and std::string_view

You may remember that a std::span is sometimes called a view.

std::ranges::range and std::ranges::view

A std::span models the concept of a range and can, therefore, be used in the algorithms of the ranges library.

A std::span is a range

```
std::vector<int> myVec{-5, 7, 10, 0, 8};
std::ranges::sort(std::span{myVec});
```

Additionally, a std::span with a dynamic extent has a default constructor and models the concept of a view.

std::string_view

A std::span and a std::string_view[23] are non-owning views and can deal with strings. The main difference between a std::span and a std::string_view is that a std::span can modify its referenced objects. A string_view also models the concept of a view.

5.2.2 Creation

There are various ways to create a std::span.

[23] https://en.cppreference.com/w/cpp/string/basic_string_view

5.2.2.1 Default constructor

You can only default construct a `std::span` with dynamic extent (`std::span<int> sp`). Creation of a `std::span` with static extent gives a compile-time error (`std::span<int, 5 sp`) if the size is not `0`: `std::span<int, 0> sp`.

5.2.2.2 Constructing and Initializing

In general, a `std::span` can be initialized using an contiguous range, two iterators defining a contiguous range, an iterator and a length, or array. The array can be a C-array or a C++-array (`std::array`). Let me show you all variations.

Constructing and initalizing `std::span`'s with static and dynamic extent

```
// constructingInitalizingSpan.cpp

#include <array>
#include <list>
#include <vector>
#include <span>

int main() {

    std::vector<int> myVec{1, 2, 3, 4, 5};
    std::list<int> myList{1, 2, 3, 4, 5};

    // Direct from a container

    std::span<int> mySpan1{myVec};
    std::span<int, 5> mySpan2{myVec};
    std::span<int, 3> mySpan3{myVec};  // undefined behavior
    // std::span<int> mySpan4{list};   // compile-time error

    // Two iterators defining a contiguous range

    std::span<int> mySpan11{std::begin(myVec), std::end(myVec)};
    std::span<int, 5> mySpan12{std::begin(myVec), std::end(myVec)};
    std::span<int, 3> mySpan13{std::begin(myVec), std::end(myVec)};   // undefined
    // std::span<int> mySpan14{std::begin(myList), std::end(myList)}; // error

    std::span<int> mySpan15{std::begin(myVec) + 2, std::end(myVec)};
    std::span<int, 3> mySpan16{std::begin(myVec), std::end(myVec) - 2};

    // An iterator and a size

    std::span<int> mySpan31{std::begin(myVec), 5};
```

```
33      std::span<int, 5> mySpan32{std::begin(myVec), 5};
34      std::span<int, 3> mySpan33{std::begin(myVec), 5};   // undefined behavior
35      // std::span<int> mySpan34{std::begin(myList), 5};  // compile-time error
36
37      std::span<int> mySpan35{myVec.data(), 5};
38      std::span<int, 5> mySpan36{myVec.data(), 5};
39
40      // A C-array and a C++-array
41
42      int cArray[5]{1, 2, 3, 4, 5};
43      std::array<int, 5> cppArray{1, 2, 3, 4, 5};
44
45      std::span<int> mySpan41{cArray};          // creates std::span<int, 5>
46      std::span<int> mySpan42{cppArray};        // creates std::span<int, 5>
47      // std::span<int, 3> mySpan43{cArray};    // compile-time error
48      // std::span<int, 3> mySpan43{cppArray};  // compile-time error
49
50  }
```

I use a `std::vector` and a `std::list` to initialize a `std::span` with a dynamic and static extent. Lines 15 - 18 use the container directly, lines 22 - 28 use a contiguous range defined by two iterators, lines 32 - 38 apply an iterator and a size, and, finally, lines 42 - 48 use a C-array and a C++-array. When you initialize a `std::span` with dynamic extent with a C-array (line 45) or a `std::array` (line 45), you get a `std::span` with static extent (`std::span<int, 5>`). In all other cases, `std::span` with dynamic extent stays a `std::span` with dynamic extent when initialized. All other use cases in the example result in undefined behavior or a compile-time error.

Initializing a `std::span` with a static extent using a container (line 17), two iterators (line 23), or an iterator with a size is undefined behavior if the size is wrong.

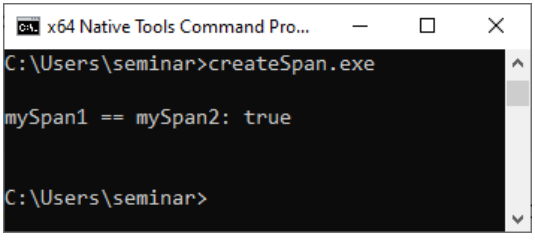
The iterator must be a `std::contiguous_iterator`. Attempting to use a `std::list` directly (line 18), or via iterators (lines 25 and 35) gives a compile-time error. A `std::list` models a `std::bidirectional_iterator`. Trying to initalize a `std::span` with static extent using an incorrectly sized container, gives a compile-time error (lines 47 and 48).

To complete this section about the creation and initialization of a `std::span`, I use in the following program a container, an iterator, and a size to create a `std:span` with dynamic extent.

Create a std::span

```
1  // createSpan.cpp
2
3  #include <algorithm>
4  #include <iostream>
5  #include <span>
6  #include <vector>
7
8  int main() {
9
10     std::cout << '\n';
11     std::cout << std::boolalpha;
12
13     std::vector myVec{1, 2, 3, 4, 5};
14
15     std::span mySpan1{myVec};
16     std::span mySpan2{myVec.data(), myVec.size()};
17
18     bool spansEqual = std::equal(mySpan1.begin(), mySpan1.end(),
19                                  mySpan2.begin(), mySpan2.end());
20
21     std::cout << "mySpan1 == mySpan2: " << spansEqual << '\n';
22
23     std::cout << '\n';
24
25 }
```

As you may expect, `mySpan1`, created from the `std::vector` (line 15), and `mySpan2`, created from a pointer and a size (line 16), are equal (line 21).



Create a `std::span` from a pointer and a size

`std::span` supports conversion in restricted ways.

5.2.2.3 Conversion

`std::span` does not support implicit conversions applied to the underlying elements. It only supports conversions by adding an additional qualifier to them. When you use a `std::span` with static or dynamic extent to initialize a `std::span` with static extent, the size must fit.

Conversion of `std::span`'s with static and dynamic extent

```
// conversionSpans.cpp

#include <array>
#include <list>
#include <vector>
#include <span>

int main() {

    std::vector<int> myVec{1, 2, 3, 4, 5};

    // conversion of underlying elements

    std::span<int> sp1{myVec};
    std::span<const int> sp2{myVec};
    std::span<int, 5> sp3{myVec};
    std::span<const int, 5> sp4{sp3};
    // std::span<int> sp5{sp2};    // compile-time error
    // std::span<long> sp6{sp1};   // compile-time error

    // static extent => dynamic extent

    std::span<int, 5> sp11{myVec};
    std::span<int> sp12{myVec};

    // dynamic extent => static extent

    std::span<int, 5> sp21{sp12};
    std::span<int, 6> sp22{sp12};  // undefined behavior

    // static extent => static extent

    // std::span<int, 6> sp31{sp21};   // compile-time error
    // std::span<int, 4> sp32{sp22};   // compile-time error

}
```

You can initialize a `std::span` with `const` underlying elements with a `std::span` with non-`const`

underlying. That holds for a `std::span` with a dynamic extent (line 15) and a `std::span` with a static extent (line 17). Doing it the other way around (line 18) or trying to initialize a `std::span<long>` with a `std::span<int>` gives a compile-time error.

You can use a `std::span` with static extent to initialize a `std::span` with dynamic extent (line 28). Initializing a `std::span` with static extent with a `std::span` with dynamic extent is undefined behavior if the sizes of the `std::span`'s differ (line 29). Furthermore, it causes a compile-time error when you use `std::span`'s with static extent to initialize a `std::span` with static extent, both having a different size.

One important reason for having a `std::span<T>` is that a plain C-array decays[24] to a pointer if passed to a function; therefore, the size is lost. This decay is a typical reason for errors in C/C++.

5.2.3 Automatically Deduces the Size of a Contiguous Sequence of Objects

In contrast to a C-array, `std::span<T>` automatically deduces the size of contiguous sequences of objects.

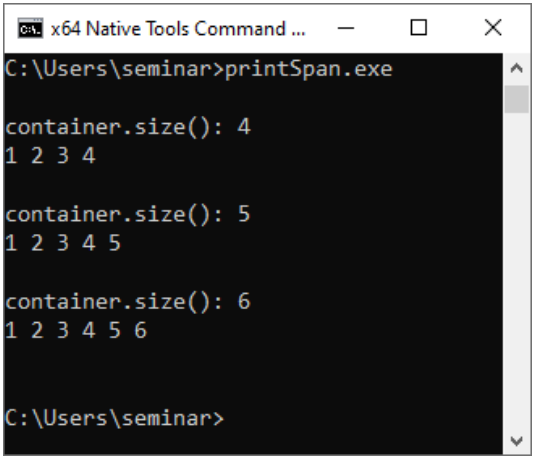
A `std::span` automatically deduces the size of its referenced sequence of objects

```
1  // printSpan.cpp
2
3  #include <iostream>
4  #include <vector>
5  #include <array>
6  #include <span>
7
8  void printMe(std::span<int> container) {
9
10     std::cout << "container.size(): " << container.size() << '\n';
11     for (auto e : container) std::cout << e << ' ';
12     std::cout << "\n\n";
13 }
14
15 int main() {
16
17     std::cout << '\n';
18
19     int arr[]{1, 2, 3, 4};
20     printMe(arr);
21
22     std::vector vec{1, 2, 3, 4, 5};
23     printMe(vec);
```

[24] https://en.cppreference.com/w/cpp/types/decay

```
24
25      std::array arr2{1, 2, 3, 4, 5, 6};
26      printMe(arr2);
27
28  }
```

The C-array (line 19), `std::vector` (line 22), and the `std::array` (line 25) contain `int` values. Consequently, `std::span` also holds `int` values. There is something more interesting in this simple example. For each container, `std::span` can deduce its size (line 10).



Automatic size deduction of a std::span

5.2.4 Modifying the Referenced Objects

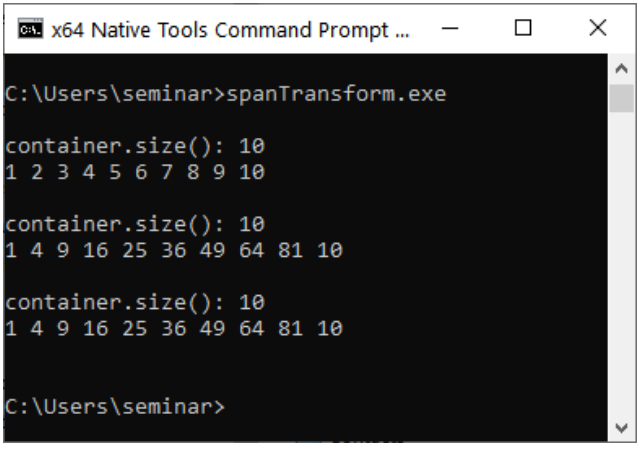
You can modify an entire span or only a subspan. When you modify a span, you modify the referenced objects.

The following program shows how a subspan can be used to modify the referenced objects from a `std::vector`.

Modify the objects referenced by a `std::span`

```
1  // spanTransform.cpp
2
3  #include <algorithm>
4  #include <iostream>
5  #include <vector>
6  #include <span>
7
8  void printMe(std::span<int> container) {
9
10     std::cout << "container.size(): " << container.size() << '\n';
11     for (auto e : container) std::cout << e << ' ';
12     std::cout << "\n\n";
13 }
14
15 int main() {
16
17     std::cout << '\n';
18
19     std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
20     printMe(vec);
21
22     std::span span1(vec);
23     std::span span2{span1.subspan(1, span1.size() - 2)};
24
25
26     std::transform(span2.begin(), span2.end(),
27                    span2.begin(),
28                    [](int i){ return i * i; });
29
30
31     printMe(vec);
32     printMe(span1);
33
34 }
```

`span1` references the `std::vector vec` (line 22). In contrast, `span2` references only the underlying `vec` elements, excluding the first and the last element (line 23). Consequently, the mapping of each element to its square addresses only those elements (line 26).



Modify the objects referend by a `std::span`

There are various convenience functions to address the elements of the `std::span`.

5.2.5 `std::span`'s Operations

The following table presents the operations you can apply on a `std::span`.

Interface of a `std::span sp`

Operation	Description
`sp.front()`	Access the first element of the sequence.
`sp.back()`	Access the last element of the sequence.
`sp[i]`	Access the `i`-th element of the sequence.
`sp.data()`	Returns a pointer to the beginning of the sequence.
`sp.size()`	Returns the number of elements of the sequence.
`sp.size_bytes()`	Returns the sequence size in bytes.
`sp.empty()`	Returns `true` if the sequence is empty.
`sp.first<count>()` `sp.first(count)`	Returns a subspan with static extent of the first `count` sequence elements. Returns a subspan with dynamic extent of the first `count` sequence elements.
`sp.last<count>()` `sp.last(count)`	Returns a subspan with static extent of the last `count` sequence elements. Returns a subspan with dynamic extent of the last `count` sequence elements.

Interface of a `std::span sp`

Operation	Description
`sp.subspan<first>()`	Returns a subspan with the **same** extent of the elements starting at `first`.
`sp.subspan(first)`	Returns a subspan with dynamic extent of the elements starting at `first`.
`sp.subspan<first, count>()`	Returns a subspan with static extent of `count` elements starting at `first`.
`sp.subspan(first, count)`	Returns a subspan with static extent of `count` elements starting at `first`.
`as_bytes`	Returns the sequence as a span of read-only `std::bytes`.
`as_writable_bytes`	Returns the sequence as a span of writable `std::bytes`.

The program `subspan.cpp` shows the member function `subspan` usage.

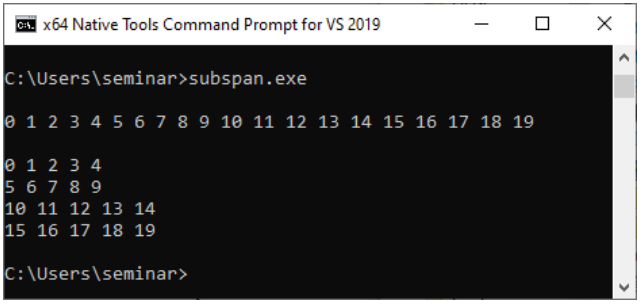
Use of the member function `subspan`

```
1  // subspan.cpp
2
3  #include <iostream>
4  #include <numeric>
5  #include <span>
6  #include <vector>
7
8  int main() {
9
10     std::cout << '\n';
11
12     std::vector<int> myVec(20);
13     std::iota(myVec.begin(), myVec.end(), 0);
14     for (auto v: myVec) std::cout << v << " ";
15
16     std::cout << "\n\n";
17
18     std::span<int> mySpan(myVec);
19     auto length = mySpan.size();
20
21     std::size_t count = 5;
22     for (std::size_t first = 0; first <= (length - count); first += count ) {
23         for (auto ele: mySpan.subspan(first, count)) std::cout << ele << " ";
24         std::cout << '\n';
25     }
26
27 }
```

Line 13 fills the vector with all numbers from 0 to 19 (line 13) using the algorithm `std::iota`[25].

[25] https://en.cppreference.com/w/cpp/algorithm/iota

Additionally, this vector initializes a `std::span` (line 18). Finally, the for loop (line 22) uses the function `subspan` to create all subspans starting at `first` and having `count` elements until `mySpan` is consumed.



Use of the member function `subspan`

Kilian Henneberger reminded me of a particular use case of `std::span`. A `std::span` can be a constant range of modifiable elements.

5.2.6 A Constant Range of Modifiable Elements

For simplicity, I name a `std::vector` and a `std::span` range. A `std::vector`, like a `std::string` models a modifiable range of modifiable elements: `std::vector<T>`. When you declare this `std::vector` as `const`, range models a constant range of constant objects: `const std::vector<T>`. You cannot model a constant range of modifiable elements. This is where `std::span` comes into play. A `std::span` models a constant range of modifiable objects: `std::span<T>`. The following table emphasizes the variations of (constant/modifiable) ranges and (constant/modifiable) elements.

(Constant/modifiable) ranges of (constant/modifiable) elements

	Modifiable Elements	Constant Elements
Modifiable Range	`std::vector<T>`	
Constant Range	`std::span<T>`	`const std::vector<T>` `std::span<const T>`

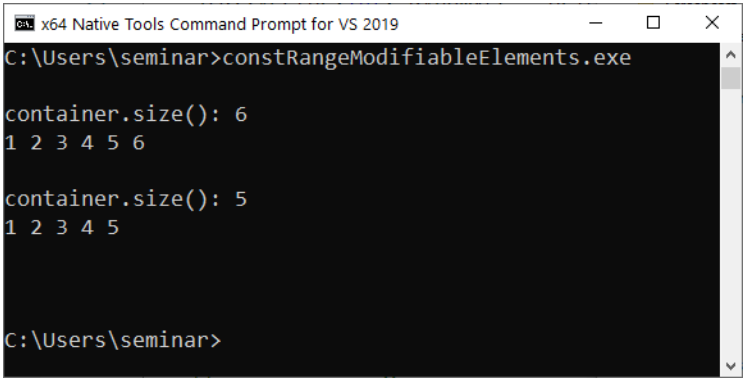
The program `constRangeModifiableElements.cpp` exemplifies each combination.

(Constant/modifiable) ranges of (constant/modifiable) elements

```
1  // constRangeModifiableElements.cpp
2
3  #include <iostream>
4  #include <span>
5  #include <vector>
6
7  void printMe(std::span<int> container) {
8
9      std::cout << "container.size(): " << container.size() << '\n';
10     for (auto e : container) std::cout << e << ' ';
11     std::cout << "\n\n";
12 }
13
14 int main() {
15
16     std::cout << '\n';
17
18     std::vector<int> origVec{1, 2, 2, 4, 5};
19
20     // Modifiable range of modifiable elements
21     std::vector<int> dynamVec = origVec;
22     dynamVec[2] = 3;
23     dynamVec.push_back(6);
24     printMe(dynamVec);
25
26     // Constant range of constant elements
27     const std::vector<int> constVec = origVec;
28     // constVec[2] = 3;         ERROR
29     // constVec.push_back(6);   ERROR
30     std::span<const int> constSpan(origVec);
31     // constSpan[2] = 3;        ERROR
32
33     // Constant range of modifiable elements
34     std::span<int> dynamSpan{origVec};
35     dynamSpan[2] = 3;
36     printMe(dynamSpan);
37
38     std::cout << '\n';
39
40 }
```

The vector `dynamVec` (line 21) is a modifiable range of modifiable elements. This observation does not hold for the vector `constVec` (line 27). Neither can `constVec` change its elements nor its size. `constSpan`

(line 30) behaves accordingly. `dynamSpan` models the unique use case of a constant range of modifiable elements.



(Constant/modifiable) ranges of (constant/modifiable) elements

Finally, I want to mention two dangers you should know when using `std::span`.

5.2.7 Dangers of `std::span`

The typical issues of `std::span` are twofold. First, a `std::span` should not act on a temporary and second, the size of the underlying contiguous range of a `std::span` should not be modified.

5.2.7.1 A `std::span` on a Temporary

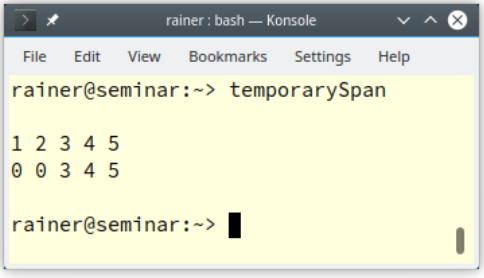
A `std::span` is never an owner. Therefore, a `std::span` does not extend the lifetime of its data. Consequently, a `std::span` should only operate on an lvalue. Using a `std::span` on a temporary is undefined behavior.

A `std::span` on temporary data

```
1  // temporarySpan.cpp
2
3  #include <iostream>
4  #include <span>
5  #include <vector>
6
7  std::vector<int> getVector() {
8      return {1, 2, 3, 4, 5};
9  }
10
11 int main() {
12
13      std::cout << '\n';
```

```
14
15      std::vector<int> myVec{1, 2, 3, 4, 5};
16      std::span<int, 5> mySpan1{myVec};
17      std::span<int, 5> mySpan2{getVector().begin(), 5};
18
19      for (auto v: std::span{myVec}) std::cout << v << " ";
20      std::cout << '\n';
21      for (auto v: std::span{getVector().begin(), 5}) std::cout << v << " ";
22
23       std::cout << "\n\n";
24
25  }
```

Using a `std::span` with a static extent (line 16) or a `std::span` with a dynamic extent (line 19) on the lvalue is fine. When I switch from the lvalue `std::vector<int>` in line 15 to a temporary `std::vector<int>`, given by the function `getVector` (lines 7 - 9), the program has undefined behavior. Both lines 17 and 21 are not valid. Consequently, executing the program exposes the undefined behavior. The output of line 21 does not match with the `std::vector<int>`, generated by the function `getVector()`.



A `std::span` on a temporary

5.2.7.2 Changing the Size of the Underlying Contiguous Range

When you change the size of the underlying contiguous range, the contiguous range may be reallocated, and the `std::span` refers to stale data. Only a `std::span` with dynamic extent can have a resizable underlying contiguous range and can, therefore, be a victim of this subtle issue.

Possible resizing of the underlying contiguous range

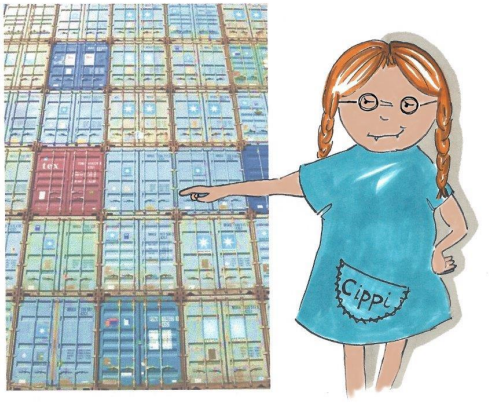
```
std::vector<int> myVec{1, 2, 3, 4, 5};

std::span<int> sp1{myVec};

myVec.push_back(6);  // undefined behavior
```

The statement `myVec.push_back(6)` can trigger a reallocation of the container. Consequently, `myVec.push_back()` causes undefined behavior.

Distilled Information

- A `std::span` is an object that refers to a contiguous sequence of objects. A `std::span`, also known as view, is never an owner and, therefore, does not allocate memory. The contiguous sequence of objects can be a plain C-array, a pointer with a size, a `std::array`, a `std::vector`, or a `std::string`.
- A `std::span` can have a static extend or a dynamic extent. The size of a `std::span` with static extent is known at compile time and cannot be changed.
- In contrast to a C-array, a `std::span` automatically deduces the size of its referenced sequence of objects.
- When a `std::span` modifies its elements, the reference objects are also modified.
- Using a `std::span` on a temporary or changing the size of the underlying range is undefined behavior.

5.3 Container and Algorithm Improvements



Cippi inspects the container

C++20 has many improvements regarding containers of the Standard Template Library. First, `std::vector` and `std::string` have **constexpr constructors** and can be used at compile time. All containers support **consistent container erasure** and the associative containers a member function **`contains`**. Thanks to the new algorithms `std::shift_left` and `std::shift_right`, you can shift the content of a container. Additionally, `std::string` allows you to **check for a prefix or suffix**. The execution policy `std::execution::unseq` permits the vectorized execution of an algorithm.

5.3.1 `constexpr` Containers and Algorithms

C++20 supports the `constexpr` containers `std::vector` and `std::string`, where `constexpr` means that the member functions of both containers can be applied at compile time. Additionally, the more than 100 algorithms[26] of the Standard Template Library are declared as `constexpr`.

Consequently, you can sort a `std::vector` of `ints` at compile time.

[26] https://en.cppreference.com/w/cpp/algorithm

Sort a `std::vector` at compile time

```
1  // constexprVector.cpp
2
3  #include <algorithm>
4  #include <iostream>
5  #include <vector>
6
7  constexpr int maxElement() {
8      std::vector myVec = {1, 2, 4, 3};
9      std::sort(myVec.begin(), myVec.end());
10     return myVec.back();
11 }
12 int main() {
13
14     std::cout <<  '\n';
15
16     constexpr int maxValue = maxElement();
17     std::cout << "maxValue: " << maxValue << '\n';
18
19     constexpr int maxValue2 = [] {
20         std::vector myVec = {1, 2, 4, 3};
21         std::sort(myVec.begin(), myVec.end()) ;
22         return myVec.back();
23     }();
24
25     std::cout << "maxValue2: " << maxValue2 << '\n';
26
27     std::cout << '\n';
28
29 }
```

The two containers `std::vector` (line 8 and 20) are sorted at compile time using `constexpr`-declared functions. In the first case, the function `maxElement` returns the last element of the vector `myVec`, which is its maximum value. In the second case, I use an immediately-invoked lambda that is declared `constexpr`.

```
maxValue: 4
maxValue2: 4
```

Sort a `std::vector` at compile time

The crucial idea for `constexpr` containers is transient allocation.

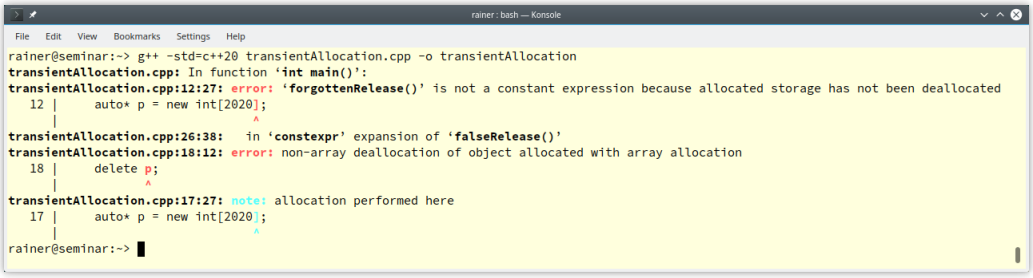
5.3.1.1 Transient Allocation

Transient allocation means that memory allocated at compile time must also be released at compile time. Consequently, the compiler can detect a mismatch of allocation and deallocation in a `constexpr` function.

Mismatch of allocation and deallocation in `constexpr` functions

```
1  // transientAllocation.cpp
2
3  #include <memory>
4
5  constexpr auto correctRelease() {
6       auto* p = new int[2020];
7       delete [] p;
8       return 2020;
9  }
10
11 constexpr auto forgottenRelease() {
12     auto* p = new int[2020];
13     return 2020;
14 }
15
16 constexpr auto falseRelease() {
17     auto* p = new int[2020];
18     delete p;
19     return 2020;
20 }
21
22 int main() {
23
24     constexpr int res1 = correctRelease();
25     constexpr int res2 = forgottenRelease();
26     constexpr int res3 = falseRelease();
27
28 }
```

The small program has two serious issues. First, the memory in the `constexpr` function `forgottenRelease` (line 11) is not released. Second, the non-array deallocation (line 18) in the `constexpr` function `falseRelease` (line 16) does not match with the array allocation.



Mismatch of allocation and deallocation in `constexpr` functions

A consequence of transient allocation is that you cannot create a `std::vector` at compile time and use it at run time.

Transient allocation of a `std::vector` fails

```
1  // transientAllocationFailed.cpp
2
3  #include <vector>
4
5  constexpr std::vector<int> getVector() {
6      std::vector vec{1, 2, 3};
7      return vec;
8  }
9
10 int main() {
11
12     constexpr std::vector vec1{1, 2 ,3};        // ERROR
13     constexpr std::vector vec2 = getVector();   // ERROR
14
15 }
```

Neither can you create a `constexpr` vector in a run-time function (line 12) function nor can you return a `std::vector` from a `constexpr` function (line 13).

5.3.2 `std::array`

C++20 offers two convenient ways to create arrays. `std::to_array` creates a `std::array` and `std::make_shared` allows it to create a `std::shared_ptr` of arrays.

5.3.2.1 `std::to_array`

`std::to_array` creates a `std::array` from an existing one-dimensional array. The elements of the created `std::array` are copy-initialized from the existing one-dimensional array.

The one-dimensional existing array can be a C-string, a `std::initializer_list`, or a one-dimensional array of `std::pair`. The following example is from cppreference.com/to_array[27].

Create a `std::array` from various one-dimensional arrays

```
1  // toArray.cpp
2
3  #include <iostream>
4  #include <utility>
5  #include <array>
6  #include <memory>
7
8  int main() {
9
10     std::cout << '\n';
11
12     auto arr1 = std::to_array("A simple test");
13     for (auto a: arr1) std::cout << a;
14     std::cout << "\n\n";
15
16     auto arr2 = std::to_array({1, 2, 3, 4, 5});
17     for (auto a: arr2) std::cout << a;
18     std::cout << "\n\n";
19
20     auto arr3 = std::to_array<double>({0, 1, 3});
21     for (auto a: arr3) std::cout << a;
22     std::cout << '\n';
23     std::cout << "typeid(arr3[0]).name(): " << typeid(arr3[0]).name() << '\n';
24     std::cout << '\n';
25
26     auto arr4 = std::to_array<std::pair<int, double>>({ {1, 0.0}, {2, 5.1},
27                                                         {3, 5.1} });
28     for (auto p: arr4) {
29         std::cout << "(" << p.first << ", " << p.second << ")" << '\n';
30     }
31
32     std::cout << "\n\n";
33
34 }
```

I created a `std::array` from a C-string (line 12), from a `std::initializer_list` (lines 16 and 20), and from a `std::initializer_list` of `std::pair`'s (line 26). In general, the compiler can deduce the type of the `std::array`. Optionally, you can specify the type (lines 20 and 26).

[27] https://en.cppreference.com/w/cpp/container/array/to_array

```
A simple test

12345

013
typeid(arr3[0]).name(): d

(1, 0)
(2, 5.1)
(3, 5.1)
```

Create various `std::array` from existing one-dimensional arrays

5.3.2.2 `std::make_shared`

Since C++11, C++ supports the creation of the `std::shared_ptr` via the factory function `std::make_shared`[28]. With C++20, this factory function supports the creation of arrays of `std::shared_ptr`.

- `std::shared_ptr<double[]> shar = std::make_shared<double[]>(1024)`: creates a `shared_ptr` with 1024 default-initialized doubles
- `std::shared_ptr<double[]> shar = std::make_shared<double[]>(1024, 1.0)`: creates a `shared_ptr` with 1024 doubles initialized to `1.0`

5.3.3 Consistent Container Erasure

Before C++20, removing elements from a container was too complicated. Let me show why.

5.3.3.1 The erase-remove Idiom

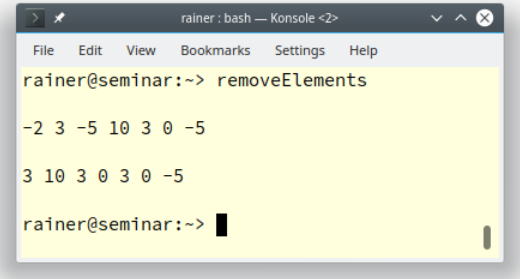
Removing an element from a container seems to be quite easy. In the case of a `std::vector`, you can use the function `std::remove_if`.

[28] https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared

Using `std::remove_if` to remove elements from a container

```
1  // removeElements.cpp
2
3  #include <algorithm>
4  #include <iostream>
5  #include <vector>
6
7  int main() {
8
9      std::cout << '\n';
10
11     std::vector myVec{-2, 3, -5, 10, 3, 0, -5 };
12
13     for (auto ele: myVec) std::cout << ele << " ";
14     std::cout << "\n\n";
15
16     std::remove_if(myVec.begin(), myVec.end(), [](int ele){ return ele < 0; });
17     for (auto ele: myVec) std::cout << ele << " ";
18
19     std::cout << "\n\n";
20
21 }
```

The program `removeElements.cpp` removes all elements from the `std::vector` that are less than zero. Easy, right? Maybe not; now, you fall into the trap that is well-known to many seasoned C++ programmer.



Using `std::remove_if` to remove elements from a container

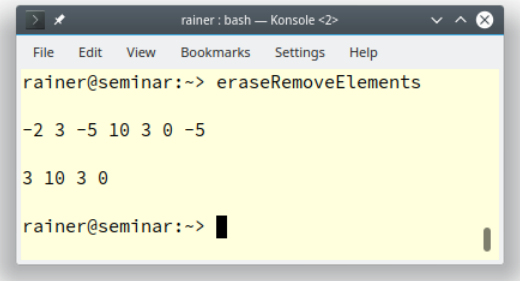
`std::remove_if` (lines 16) does not remove anything. The `std::vector` still has the same number of arguments. Both algorithms return the new logical end of the modified container.

To modify a container, you have to apply the new logical end to the container.

Applying the erase-remove idiom to a container

```
1  // eraseRemoveElements.cpp
2
3  #include <algorithm>
4  #include <iostream>
5  #include <vector>
6
7  int main() {
8
9      std::cout << '\n';
10
11     std::vector myVec{-2, 3, -5, 10, 3, 0, -5 };
12
13     for (auto ele: myVec) std::cout << ele << " ";
14     std::cout << "\n\n";
15
16     auto newEnd = std::remove_if(myVec.begin(), myVec.end(),
17                                  [](int ele){ return ele < 0; });
18     myVec.erase(newEnd, myVec.end());
19     // myVec.erase(std::remove_if(myVec.begin(), myVec.end(),
20     //             [](int ele){ return ele < 0; }), myVec.end());
21     for (auto ele: myVec) std::cout << ele << " ";
22
23     std::cout << "\n\n";
24
25 }
```

Line (16) returns the new logical end `newEnd` of the container `myVec`. This new logical end is applied in line 18 to remove all elements from `myVec` starting at `newEnd`. When you apply the functions `remove` and `erase` in one expression such as in line 19, you see exactly why this construct is called erase-remove idiom.



Using the erase-remove idiom

Thanks to the new functions `erase` and `erase_if` in C++20, erasing elements from containers is far more convenient.

5.3.3.2 `erase` and `erase_if` in C++20

With `erase` and `erase_if`, you can directly operate on the container. In contrast, the previously presented erase-remove idiom is quite verbose: it requires two iterations.

Let's see what the new functions `erase` and `erase_if` mean in practice. The following program erases elements from a few containers.

Erase elements from a container

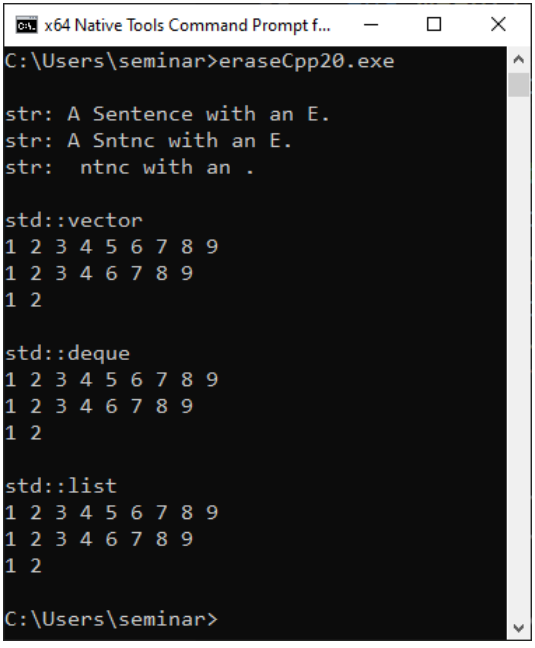
```
1  // eraseCpp20.cpp
2
3  #include <iostream>
4  #include <numeric>
5  #include <deque>
6  #include <list>
7  #include <string>
8  #include <vector>
9
10 template <typename Cont>
11 void eraseVal(Cont& cont, int val) {
12     std::erase(cont, val);
13 }
14
15 template <typename Cont, typename Pred>
16 void erasePredicate(Cont& cont, Pred pred) {
17     std::erase_if(cont, pred);
18 }
19
```

```
20  template <typename Cont>
21  void printContainer(Cont& cont) {
22      for (auto c: cont) std::cout << c << " ";
23      std::cout << '\n';
24  }
25
26  template <typename Cont>
27  void doAll(Cont& cont) {
28      printContainer(cont);
29      eraseVal(cont, 5);
30      printContainer(cont);
31      erasePredicate(cont, [](auto i) { return i >= 3; } );
32      printContainer(cont);
33  }
34
35  int main() {
36
37      std::cout << '\n';
38
39      std::string str{"A Sentence with an E."};
40      std::cout << "str: " << str << '\n';
41      std::erase(str, 'e');
42      std::cout << "str: " << str << '\n';
43      std::erase_if( str, [](char c){ return std::isupper(c); });
44      std::cout << "str: " << str << '\n';
45
46      std::cout << "\nstd::vector " << '\n';
47      std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
48      doAll(vec);
49
50      std::cout << "\nstd::deque " << '\n';
51      std::deque deq{1, 2, 3, 4, 5, 6, 7, 8, 9};
52      doAll(deq);
53
54      std::cout << "\nstd::list" << '\n';
55      std::list lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
56      doAll(lst);
57
58  }
```

Line 41 erases all the `'e'` characters from the given string `str`. Line 43 applies the lambda expression to the same string, eraseing all uppercase letters.

In the rest of the program, elements of the sequence containers `std::vector` (line 47), `std::deque` (line 51), and `std::list` (line 55) are erased. On each container, the function template `doAll` (line 26) is

applied. `doAll` erases the element 5 and all elements greater than or equal to 3. The function template `eraseVal` (line 10) uses the new function `erase` and the function template `erasePredicate` (line 15) uses the new function `erase_if`.



Application of the new functions erase and erase_if

The new functions `erase` and `erase_if` can be applied to all containers of the Standard Template Library. This does not hold for the next convenience function `contains`, which requires an associative container.

5.3.4 `contains` for Associative Containers

Thanks to the function `contains`, you can easily check if an element exists in an associative container. Stop, you may say, we can already do this with `find` or `count`.

No, both functions are not beginner-friendly and have their downsides.

Erase elements from a container

```
1  // checkExistence.cpp
2
3  #include <set>
4  #include <iostream>
5
6  int main() {
7
8      std::cout << '\n';
9
10     std::set mySet{3, 2, 1};
11     if (mySet.find(2) != mySet.end()) {
12         std::cout << "2 inside" << '\n';
13     }
14
15     std::multiset myMultiSet{3, 2, 1, 2};
16     if (myMultiSet.count(2)) {
17         std::cout << "2 inside" << '\n';
18     }
19
20     std::cout << '\n';
21
22 }
```

The functions produce the expected result.

```
2 inside
2 inside
```

Use of `find` and `count` to check if a container has a given element

There are issues with both calls. The `find` call (line 11) is too verbose. The same argument holds for the `count` call (line 16). The `count` call also has a performance issue. When you want to know if an element is in a container, you should stop when you found it and not count until the end. In the concrete case, `myMultiSet.count(2)` returned 2.

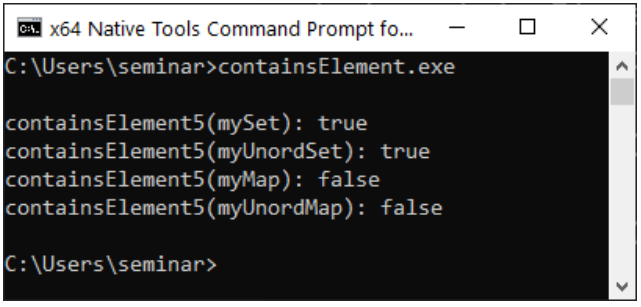
Unlike `find` and `count`, the `contains` member function in C++20 is quite convenient to use.

`contains` in C++20

```
// containsElement.cpp

#include <iostream>
#include <set>
#include <map>
#include <unordered_set>
#include <unordered_map>

template <typename AssocCont>
bool containsElement5(const AssocCont& assocCont) {
    return assocCont.contains(5);
}

int main() {

    std::cout << std::boolalpha;

    std::cout << '\n';

    std::set<int> mySet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    std::cout << "containsElement5(mySet): " << containsElement5(mySet);

    std::cout << '\n';

    std::unordered_set<int> myUnordSet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    std::cout << "containsElement5(myUnordSet): " << containsElement5(myUnordSet);

    std::cout << '\n';

    std::map<int, std::string> myMap{ {1, "red"}, {2, "blue"}, {3, "green"} };
    std::cout << "containsElement5(myMap): " << containsElement5(myMap);

    std::cout << '\n';

    std::unordered_map<int, std::string> myUnordMap{ {1, "red"},
                                                     {2, "blue"}, {3, "green"} };
    std::cout << "containsElement5(myUnordMap): " << containsElement5(myUnordMap);

    std::cout << '\n';

}
```

There is not much to add to this example. The function template `containsElement5` returns `true` if

the associative container contains the key 5. In my example, I used only the associative containers std::set, std::unordered_set, std::map, and std::unordered_set, none of which can hold a given key more than once.



Use of the new function contains

5.3.5 Shift the Content of a Container

Thanks to the new algorithms std::shift_left and std::shift_right, you can shift the content of a container left or right by n positions. The following rules apply to the range [begin, end).

- std::shift_left(begin, end, n): Shifts the elements left by n positions.
- std::right_left(begin, end, n): Shifts the elements right by n positions.

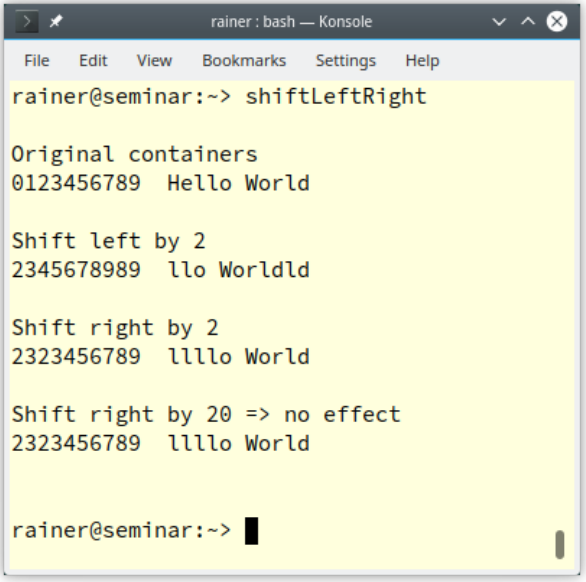
The calls have no effect if n is zero (n == 0) or n is equal to or bigger than the size of the container (n >= end - begin). Elements of the range in the original range but not the new range are afterward in a valid, but unspecified state. This essentially means that the elements are valid, but you don't know their values. The following program applies shift operations onto a std::vector and a std::string.

std::shift and std::shift_right applied onto two containers

```
1  // shiftLeftRigth.cpp
2
3  #include <algorithm>
4  #include <iostream>
5  #include <string>
6  #include <vector>
7
8  void printBoth(const std::vector<int>& myVec, const std::string& myStr,
9                 const std::string& mess) {
10
11     std::cout << mess << '\n';
12     for (auto v: myVec) std::cout << v;
13     std::cout << "  ";
14     for (auto s: myStr) std::cout << s;
15     std::cout << "\n\n";
```

```
16
17  }
18
19  int main() {
20
21      std::cout << '\n';
22
23      std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
24      std::string myStr("Hello World");
25
26      printBoth(myVec, myStr, "Original containers");
27
28      std::shift_left(std::begin(myVec), std::end(myVec), 2);
29      std::shift_left(std::begin(myStr), std::end(myStr), 2);
30
31      printBoth(myVec, myStr, "Shift left by 2");
32
33      std::shift_right(std::begin(myVec), std::end(myVec), 2);
34      std::shift_right(std::begin(myStr), std::end(myStr), 2);
35
36      printBoth(myVec, myStr, "Shift right by 2");
37
38      std::shift_right(std::begin(myVec), std::end(myVec), 20);
39      std::shift_right(std::begin(myStr), std::end(myStr), 20);
40
41      printBoth(myVec, myStr, "Shift right by 20 => no effect");
42
43      std::cout << '\n';
44
45  }
```

In the program `shiftLeftRight.cpp` the `std::vector` and `std::string` are left-shifted by 2 (lines 28 and 29) and right-shifted by 2 (lines 33 and 34). The left shift operation by 2 puts the two last elements in a valid but unspecified state. Accordingly, the same holds for the right shift operation for the first two elements. The left shift operations in lines 38 and 39 have no effect because 20 is bigger than the size of the container.



`std::shift` and `std::shift_right` applied onto two containers

5.3.6 String prefix and suffix checking

`std::string` gets new member functions `starts_with` and `ends_with`. They allow you to check if a `std::string` starts or ends with a specified substring.

Check if a string starts with or ends with a given string

```
1  // stringStartsWithEndsWith.cpp
2
3  #include <iostream>
4  #include <string_view>
5  #include <string>
6
7  template <typename PrefixType>
8  void startsWith(const std::string& str, PrefixType prefix) {
9      std::cout << "           starts with " << prefix << ": "
10               << str.starts_with(prefix) << '\n';
11 }
12
13 template <typename SuffixType>
14 void endsWith(const std::string& str, SuffixType suffix) {
15     std::cout << "           ends with " << suffix << ": "
16               << str.ends_with(suffix) << '\n';
```

```
17  }
18
19  int main() {
20
21      std::cout << '\n';
22
23      std::cout << std::boolalpha;
24
25      std::string helloWorld("Hello World");
26
27      std::cout << helloWorld << '\n';
28
29      startsWith(helloWorld, helloWorld);
30
31      startsWith(helloWorld, std::string_view("Hello"));
32
33      startsWith(helloWorld, 'H');
34
35      std::cout << "\n\n";
36
37      std::cout << helloWorld << '\n';
38
39      endsWith(helloWorld, helloWorld);
40
41      endsWith(helloWorld, std::string_view("World"));
42
43      endsWith(helloWorld, 'd');
44
45  }
```

Both member functions `starts_with` and `ends_with` are predicates and, hence, return a boolean. You can invoke the new member functions `starts_with` and `ends_with` with a `std::string` (lines 29 and 39), a `std::string_view` (lines 31 and 41), and a `char` (lines 33 and 43).

```
Hello World
            starts with Hello World: true
            starts with Hello: true
            starts with H: true


Hello World
            ends with Hello World: true
            ends with World: true
            ends with d: true
```

Check if a string starts with or ends with a given string

5.3.7 Vectorized Execution Policy: `std::execution::unseq`

Using an execution policy in C++17, you can specify whether the algorithm should run sequentially, in parallel, or parallel with vectorization.

The policy tag specifies whether an algorithm should run sequentially, in parallel, or in parallel with vectorization.

- **`std::execution::seq`**: runs the algorithm sequentially
- **`std::execution::par`**: runs the algorithm in parallel on multiple threads
- **`std::execution::par_unseq`**: runs the algorithm in parallel on multiple threads and allows the interleaving of individual loops; permits a vectorized version with SIMD[29] (**S**ingle **I**nstruction **M**ultiple **D**ata) extensions.

C++20 supports a new execution policy: `std::execution::unseq`:

- **`std::execution::unseq`**: runs the algorithm on **one thread**; permits a vectorized version with SIMD[30] (**S**ingle **I**nstruction **M**ultiple **D**ata) extensions.

The following code snippet shows the application of the execution policies.

[29] https://en.wikipedia.org/wiki/SIMD
[30] https://en.wikipedia.org/wiki/SIMD

The execution policies

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};

// sequential execution (C++98)
std::sort(v.begin(), v.end());

// sequential execution (C++17)
std::sort(std::execution::seq, v.begin(), v.end());

// permitting parallel execution (C++17)
std::sort(std::execution::par, v.begin(), v.end());

// permitting parallel and vectorized execution (C++17)
std::sort(std::execution::par_unseq, v.begin(), v.end());

// permitting vectorized execution (C++20)
std::sort(std::execution::unseq, v.begin(), v.end());
```

Applying the vectorized execution policies (`std::execution::par_unseq` or `std::execution::unseq`) does not guarantee vectorized execution. You permit your architecture to execute it vectorized. Additionally, you should not use a blocking mechanism such as a mutex. This may end in a deadlock.

Distilled Information

- `std::vector` and `std::string` have `constexpr` constructors and can, therefore, be instantiated at compile time. Thanks to the `constexpr` algorithms of the Standard Template Library (STL), you can manipulate them at compile time.
- C++20 offers two convenient ways to create arrays. `std::to_array` creates a `std::array` and `std::make_shared` allows the creation of a `std::shared_ptr` wrapping a C-array.
- The new algorithm `std::erase` and `std::erase_if` are used to erase specific elements (`erase`) or elements satisfying a predicate (`erase_if`) from an arbitrary container of the STL.
- Thanks to the member function `contains`, you can check for an associative container if it has the requested key.
- The new algorithms `std::shift_left` and `std::shift_right` enable it to shift the content of a container by n positions.
- `std::string` supports the new member function `start_with` and `end_with` to check if the container has a specific prefix or suffix.
- The new execution policy `std::execution::unseq` permits the vectorized execution of an algorithm.

5.4 Arithmetic Utilities

Cippi studies arithmetic

Compairing signed and unsigned integers is a subtle cause for unexpected behavior and, therefore, bugs. Thanks to the new safe comparison functions for integers, `std::cmp_*`, a source of subtle bugs is gone. Additionally, C++20 includes **mathematical constants** such as e, π, or ϕ, and with the functions `std::midpoint` and `std::lerp`, you can calculate the midpoint of two numbers or their linear interpolation. The new bit manipulation allows you to access and modify individual bits or bit sequences.

5.4.1 Safe Comparison of Integers

When you compare signed and unsigned integers, you may not get the result you expect. Thanks to the six `std::cmp_*` functions, there is a cure in C++20. Motivating safe comparison of integers, I want to start with the unsafe variant.

Integral versus Integer

The terms integral and integer are synonyms in C++. This is the wording from the standard for fundamental types: *"Types `bool`, `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`, and the `signed` and `unsigned` integer types are collectively called integral types. A synonym for [an] integral type is integer type"*. I prefer the term integer in this book.

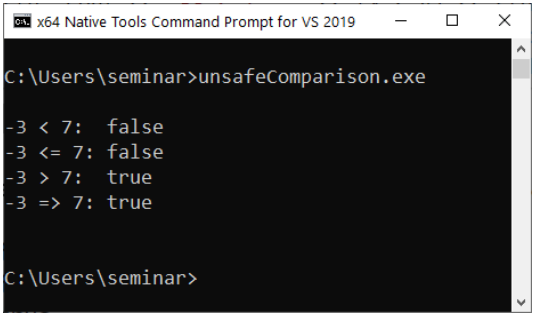
5.4.1.1 Unsafe Comparison

Of course, there is a reason for the name `unsafeComparison.cpp` of the following program.

Unsafe comparison of integers

```
1  // unsafeComparison.cpp
2
3  #include <iostream>
4
5  int main() {
6
7      std::cout << '\n';
8
9      std::cout << std::boolalpha;
10
11     int x = -3;
12     unsigned int y = 7;
13
14     std::cout << "-3 < 7:  " << (x < y) << '\n';
15     std::cout << "-3 <= 7: " << (x <= y) << '\n';
16     std::cout << "-3 > 7:  " << (x > y) << '\n';
17     std::cout << "-3 => 7: " << (x >= y) << '\n';
18
19     std::cout << '\n';
20
21 }
```

When I execute the program, the output may not meet your expectations.



Surprises with unsafe comparisons of integers

If you read the program's output, you will see that `-3` is greater than `7`. You presumably know the reason. I compared a `signed x` (line 11) with an `unsigned y` (line 12). What is happening under the hood? The following program provides the answer.

Unsafe comparison of integers resolved

```
1  // unsafeComparison2.cpp
2
3  int main() {
4      int x = -3;
5      unsigned int y = 7;
6
7      bool val = x < y;
8      static_assert(static_cast<unsigned int>(-3) == 4'294'967'293);
9  }
```

In the example, I'm focusing on the less-than operator. C++ Insights[31] gives me the following output:

```
int main()
{
  int x = -3;
  unsigned int y = 7;
  bool val = static_cast<unsigned int>(x) < y;
  /* PASSED: static_assert(static_cast<long>(static_cast<unsigned int>(-3)) == 4294967293L); */
}
```

Unsafe comparison analyzed

Here is what's happening:

- The compiler transforms the expression `x < y` (line 7) into `static_cast<unsigned int>(x) < y`. In particular, the `signed x` is converted to an `unsigned int`.
- Due to the conversion, `-3` becomes `4'294'967'293`.
- `4'294'967'293` is equal to $-3 \mod 2^{32}$
- `32` is the number of bits of an `unsigned int` on C++ Insights.

Thanks to C++20, we have a safe comparison of integers.

5.4.1.2 Safe Comparison of Integers

C++20 supports six comparison functions for integers:

[31] https://cppinsights.io/s/62732a01

Six safe comparison functions

Compare Function	Meaning
`std::cmp_equal`	`==`
`std::cmp_not_equal`	`!=`
`std::cmp_less`	`<`
`std::cmp_less_equal`	`<=`
`std::cmp_greater`	`>`
`std::cmp_greater_equal`	`>=`

Thanks to the six comparison functions, I can easily transform the previous program `unsafeComparison.cpp` into the program `safeComparison.cpp`. The new comparison functions require the header `<utility>`.

Safe comparison of integers

```
// safeComparison.cpp

#include <iostream>
#include <utility>

int main() {

    std::cout << '\n';

    std::cout << std::boolalpha;

    int x = -3;
    unsigned int y = 7;

    std::cout << "-3 == 7:  " << std::cmp_equal(x, y) << '\n';
    std::cout << "-3 != 7:  " << std::cmp_not_equal(x, y) << '\n';
    std::cout << "-3 < 7:   " << std::cmp_less(x, y) << '\n';
    std::cout << "-3 <= 7:  " << std::cmp_less_equal(x, y) << '\n';
    std::cout << "-3 > 7:   " << std::cmp_greater(x, y) << '\n';
    std::cout << "-3 => 7:  " << std::cmp_greater_equal(x, y) << '\n';

    std::cout << '\n';

}
```

Additionally, I applied the equal and not equal operators.

```
-3 == 7:  false
-3 != 7:  true
-3 < 7:   true
-3 <= 7:  true
-3 > 7:   false
-3 => 7:  false
```

Safe comparison

Invoking a safe-comparison function with a non-integer, such as a `double`, causes a compile-time error.

Safe comparison of an `unsigned int` and a `double`

```
// safeComparison2.cpp

#include <iostream>
#include <utility>

int main() {

    double x = -3.5;
    unsigned int y = 7;

    std::cout << "-3.5 < 7:  " << std::cmp_less(x, y);  // ERROR

}
```

On the other hand, you can compare a `double` and an `unsigned int` the classical way. The program `classicalComparison.cpp` applies classical comparison of a `double` and an `unsigned int`.

Classical comparison of an `unsigned int` and a `double`

```
// classicalComparison.cpp

int main() {

    double x = -3.5;
    unsigned int y = 7;

    auto res = x < y;    // true

}
```

It works. The `unsigned int` is `floating-point` promoted[32] to `double`. C++ Insights[33] shows the truth:

[32] https://en.cppreference.com/w/cpp/language/implicit_conversion
[33] https://cppinsights.io/s/44216566

```
int main()
{
  double x = -3.5;
  unsigned int y = 7;
  bool res = x < static_cast<double>(y);
}
```

Floating point promotion to `double`

Additionally, the function `std::in_range<R>(t)` returns `true` if `t` can be represented in the type `R`. `std::in_range` determines if `t` is greater than or equal to the minimum value and less than or equal to the maximum value of type `R`.

The following code-snippet shows a possible implementation from cppreference.com/std::in_range[34].

Possible implementation of `std::in_range`

```
template<class R, class T>
constexpr bool in_range(T t) noexcept {
    return std::cmp_greater_equal(t, std::numeric_limits<R>::min()) &&
           std::cmp_less_equal(t, std::numeric_limits<R>::max());
}
```

5.4.2 Mathematical Constants

First of all, the constants need the header `<numbers>` and the namespace `std::numbers`. The following table gives you an overview.

The mathematical constants

Mathematical Constant	Description
`std::numbers::e`	e
`std::numbers::log2e`	$\log_2 e$
`std::numbers::log10e`	$\log_{10} e$
`std::numbers::pi`	π
`std::numbers::inv_pi`	$\frac{1}{\pi}$
`std::numbers::inv_sqrtpi`	$\frac{1}{\sqrt{\pi}}$
`std::numbers::ln2`	$\ln 2$

[34] https://en.cppreference.com/w/cpp/utility/in_range

The mathematical constants

Mathematical Constant	Description
`std::numbers::ln10`	$\ln 10$
`std::numbers::sqrt2`	$\sqrt{2}$
`std::numbers::sqrt3`	$\sqrt{3}$
`std::numbers::inv_sqrt3`	$\frac{1}{\sqrt{3}}$
`std::numbers::egamma`	Euler-Mascheroni constant[35]
`std::numbers::phi`	ϕ

The program `mathematicConstants.cpp` applies the mathematical constants.

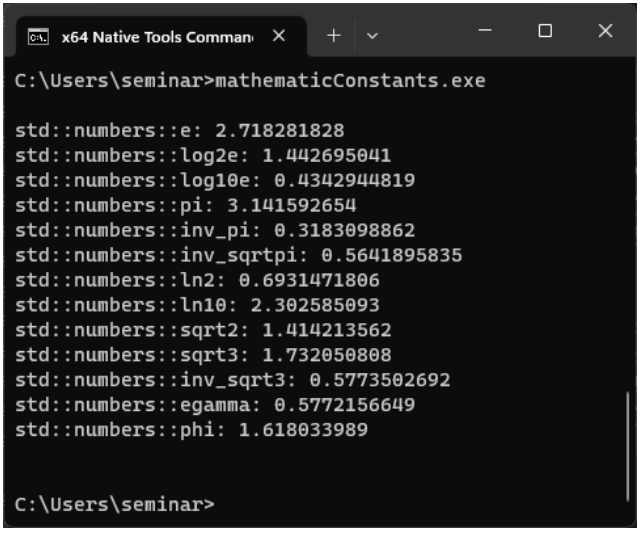
The mathematical constants

```
// mathematicConstants.cpp

#include <iomanip>
#include <iostream>
#include <numbers>

int main() {

    std::cout << '\n';

    std::cout<< std::setprecision(10);

    std::cout << "std::numbers::e: " <<  std::numbers::e << '\n';
    std::cout << "std::numbers::log2e: " <<  std::numbers::log2e << '\n';
    std::cout << "std::numbers::log10e: " <<  std::numbers::log10e << '\n';
    std::cout << "std::numbers::pi: " <<  std::numbers::pi << '\n';
    std::cout << "std::numbers::inv_pi: " <<  std::numbers::inv_pi << '\n';
    std::cout << "std::numbers::inv_sqrtpi: " <<  std::numbers::inv_sqrtpi << '\n';
    std::cout << "std::numbers::ln2: " <<  std::numbers::ln2 << '\n';
    std::cout << "std::numbers::ln10: " <<  std::numbers::ln10 << '\n';
    std::cout << "std::numbers::sqrt2: " <<  std::numbers::sqrt2 << '\n';
    std::cout << "std::numbers::sqrt3: " <<  std::numbers::sqrt3 << '\n';
    std::cout << "std::numbers::inv_sqrt3: " <<  std::numbers::inv_sqrt3 << '\n';
    std::cout << "std::numbers::egamma: " <<  std::numbers::egamma << '\n';
    std::cout << "std::numbers::phi: " <<  std::numbers::phi << '\n';
```

[35] https://en.wikipedia.org/wiki/Euler%E2%80%93Mascheroni_constant

```
    std::cout << '\n';

}
```

Here is the output of the program with the MSVC compiler.



Use of all mathematical constants

The mathematical constants are available for `float`, `double`, and `long double`. By default, `double` is used but, you can also specify `float` (`std::numbers::pi_v<float>`) or `long double` (`std::numbers::pi_v<long double>`).

5.4.3 Midpoint and Linear Interpolation

- **std::midpoint(a, b)**: calculates the midpoint `(a + (b - a) / 2)` of integers, floating points, or pointers. If `a` and `b` are pointers, they have to point to the same array object. The function needs the header `<numeric>`.
- **std::lerp(a, b, t)**: calculates the linear interpolation `(a + t(b - a))`. When `t` is outside the range `[0, 1]`, it calculates the linear extrapolation. The function needs the header `<cmath>`.

The program `midpointLerp.cpp` applies both functions.

Calculating the midpoint and the linear interpolation of numbers

```
// midpointLerp.cpp

#include <cmath>
#include <numeric>
#include <iostream>

int main() {

    std::cout << '\n';

    std::cout << "std::midpoint(10, 20): " << std::midpoint(10, 20) << '\n';

    std::cout << '\n';

    for (auto v: {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}) {
        std::cout << "std::lerp(10, 20, " << v << "): " << std::lerp(10, 20, v)
                  << '\n';
    }

    std::cout << '\n';

}
```

The program should, together with its output, be self-explanatory.

```
std::midpoint(10, 20): 15

std::lerp(10, 20, 0): 10
std::lerp(10, 20, 0.1): 11
std::lerp(10, 20, 0.2): 12
std::lerp(10, 20, 0.3): 13
std::lerp(10, 20, 0.4): 14
std::lerp(10, 20, 0.5): 15
std::lerp(10, 20, 0.6): 16
std::lerp(10, 20, 0.7): 17
std::lerp(10, 20, 0.8): 18
std::lerp(10, 20, 0.9): 19
std::lerp(10, 20, 1): 20
```

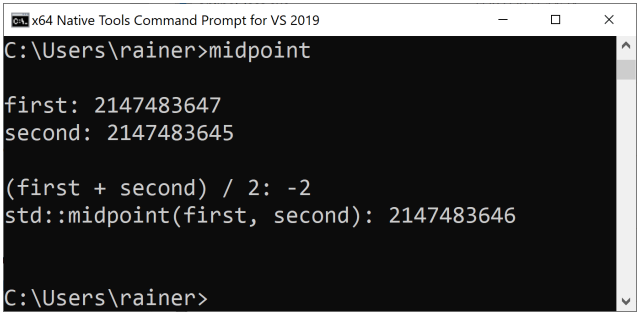
Calculating the midpoint and the linear interpolation of numbers

In contrast to a naive calculation of the midpoint of two numbers `first` and `second` with the expression `(first + second) / 2`, `std::midpoint(first, second)` automatically deals with overflow errors.

Calculating the midpoint of two big numbers without overflow

```
1  // midpoint.cpp
2
3  #include <limits>
4  #include <numeric>
5  #include <iostream>
6
7  int main() {
8
9      std::cout << '\n';
10
11     int first = std::numeric_limits<int>::max();
12     int second = std::numeric_limits<int>::max() - 2;
13
14     std::cout << "first: " << first << '\n';
15     std::cout << "second: " << second << '\n';
16
17     std::cout << '\n';
18
19     std::cout << "(first + second) / 2: " << (first + second) / 2 << '\n';
20     std::cout << "std::midpoint(first, second): " << std::midpoint(first, second) << '\n';
21
22     std::cout << '\n';
23
24 }
```

The calculation `(first + second) / 2` (line 19) causes an overflow because the variable `first` (line 19) is the largest possible value for type `int` and the variable `second` (line 20) is quite close to it. On the contrary, `std::midpoint(first, second)` (line 20) returns the correct value.



Calculating the midpoint of two big numbers without overflow

5.4.4 Bit Manipulation

The header <bit> supports various constexpr functions to access and manipulate individual bits or bit sequences.

5.4.4.1 std::endian

Thanks to the new type std::endian, you get the endianness of a scalar type. Endianness can be big-endian or little-endian. Big-endian means that the most significant byte is furthest left, little-endian means that the least significant byte is furthest left. A scalar type is either an arithmetic type, an enum, a pointer, a member pointer, or a std::nullptr_t.

The class endian provides the endianness of all scalar types:

enum class endian

```
enum class endian
{
    little = /*implementation-defined*/,
    big    = /*implementation-defined*/,
    native = /*implementation-defined*/
};
```

- If all scalar types are little-endian, std::endian::native is equal to std::endian::little.
- If all scalar types are big-endian, std::endian::native is equal to std::endian::big.

Even corner cases are supported:

- If all scalar types have sizeof 1 and therefore endianness does not matter, the values of the enumerators std::endian::little, std::endian::big, and std::endian::native are identical.
- If the platform uses mixed endianness, std::endian::native is neither equal to std::endian::big nor std::endian::little.

When I perform the following program getEndianness.cpp on a x86 architecture, I get the answer little-endian.

enum class endian

```
// getEndianness.cpp

#include <bit>
#include <iostream>

int main() {

    if constexpr (std::endian::native == std::endian::big) {
        std::cout << "big-endian" << '\n';
```

```
    }
    else if constexpr (std::endian::native == std::endian::little) {
        std::cout << "little-endian"  << '\n';      // little-endian
    }

}
```

`constexpr if` enables the compiler to compile source code conditionally. Thhat is, compilation depends on the endianness of your architecture.

5.4.4.2 Accessing or Manipulating Bits or Bit Sequences

The following table gives you an overview of all functions. You can find the functions in the header `<bit>`.

Bit manipulation

Function	Description
`std::bit_cast`	Reinterprets the object representation
`std::has_single_bit`	Checks if a number is a power of two
`std::bit_ceil`	Finds the smallest integer power of two that is not smaller than the given value
`std::bit_floor`	Finds the largest integer power of two that is not greater than the given value
`std::bit_width`	Finds the smallest number of bits to represent the given value
`std::rotl`	Computes the bitwise left-rotation
`std::rotr`	Computes the bitwise right-rotation
`std::countl_zero`	Counts the number of consecutive `0`s, starting with the most significant bit
`std::countl_one`	Counts the number of consecutive `1`s, starting with the most significant bit
`std::countr_zero`	Counts the number of consecutive `0`s, starting with the least significant bit
`std::countr_one`	Counts the number of consecutive `1`s, starting with the least significant bit
`std::popcount`	Counts the number of `1`s in an unsigned integer

All of the functions except `std::bit_cast` require an unsigned integral type (`unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`). `std::bit_cast` guarantees that the

number of bits fits. When you invoke the rotate functions `std::rotl`, or `std::rotr` with a negative number, the rotation changes direction.

The program `bit.cpp` shows the application of the functions.

Bit manipulation

```
// bit.cpp

#include <bit>
#include <bitset>
#include <cstdint>
#include <iostream>

int main() {

    std::uint8_t num= 0b00110010;

    std::cout << std::boolalpha;

    std::cout << "std::has_single_bit(0b00110010): " << std::has_single_bit(num)
              << '\n';

    std::cout << "std::bit_ceil(0b00110010): " << std::bitset<8>(std::bit_ceil(num))
              << '\n';
    std::cout << "std::bit_floor(0b00110010): "
              << std::bitset<8>(std::bit_floor(num)) << '\n';

    std::cout << "std::bit_width(5u): " << std::bit_width(5u) << '\n';

    std::cout << "std::rotl(0b00110010, 2): " << std::bitset<8>(std::rotl(num, 2))
              << '\n';
    std::cout << "std::rotr(0b00110010, 2): " << std::bitset<8>(std::rotr(num, 2))
              << '\n';

    std::cout << "std::countl_zero(0b00110010): " << std::countl_zero(num) << '\n';
    std::cout << "std::countl_one(0b00110010): " << std::countl_one(num) << '\n';
    std::cout << "std::countr_zero(0b00110010): " << std::countr_zero(num) << '\n';
    std::cout << "std::countr_one(0b00110010): " << std::countr_one(num) << '\n';
    std::cout << "std::popcount(0b00110010): " << std::popcount(num) << '\n';

}
```

Here is the output of the program.

```
std::has_single_bit(0b00110010): false
std::bit_ceil(0b00110010): 01000000
std::bit_floor(0b00110010): 00100000
std::bit_width(5u): 3
std::rotl(0b00110010, 2): 11001000
std::rotr(0b00110010, 2): 10001100
std::countl_zero(0b00110010): 2
std::countl_one(0b00110010): 0
std::countr_zero(0b00110010): 1
std::countr_one(0b00110010): 0
std::popcount(0b00110010): 3
```

Bit manipulation

The following program shows the `std::bit_floor`, `std::bit_ceil`, `std::bit_width`, and `std::popcount` for the numbers 2 to 7.

Displaying `std::bit_floor`, `std::bit_ceil`, `std::bit_width`, and `std::popcount` for a few numbers

```
// bitFloorCeil.cpp

#include <bit>
#include <bitset>
#include <iostream>

int main() {

    std::cout << '\n';

    std::cout << std::boolalpha;

    for (auto i = 2u; i < 8u; ++i) {
         std::cout << "bit_floor(" << std::bitset<8>(i) << ") = "
                   << std::bit_floor(i) << '\n';

        std::cout << "bit_ceil(" << std::bitset<8>(i) << ") = "
                  << std::bit_ceil(i) << '\n';

        std::cout << "bit_width(" << std::bitset<8>(i) << ") = "
                  << std::bit_width(i) << '\n';

        std::cout << "popcount(" << std::bitset<8>(i) << ") = "
                  << std::popcount(i) << '\n';

        std::cout << '\n';
    }
```

```
    std::cout << '\n';

}
```

```
bit_floor(00000010) = 2
bit_ceil(00000010) = 2
bit_width(00000010) = 2
popcount(00000010) = 1

bit_floor(00000011) = 2
bit_ceil(00000011) = 4
bit_width(00000011) = 2
popcount(00000011) = 2

bit_floor(00000100) = 4
bit_ceil(00000100) = 4
bit_width(00000100) = 3
popcount(00000100) = 1

bit_floor(00000101) = 4
bit_ceil(00000101) = 8
bit_width(00000101) = 3
popcount(00000101) = 2

bit_floor(00000110) = 4
bit_ceil(00000110) = 8
bit_width(00000110) = 3
popcount(00000110) = 2

bit_floor(00000111) = 4
bit_ceil(00000111) = 8
bit_width(00000111) = 3
popcount(00000111) = 3
```

Displaying `std::bit_floor`, `std::bit_ceil`, `std::bit_width`, and `std::popcount` for a few numbers

Distilled Information

- The `cmp_*` functions in C++20 support the safe comparison of integrals because they detect the comparison of a signed and an unsigned integral. In the case of an unsafe comparison, the compilation fails.
- Many mathematical constants such as e, $\log_2 e$, or π are now defined.
- C++20 provides utility functions for calculating the midpoint or linear interpolation of two values.
- New functions to access and manipulate individual bits or bit sequences are available.

5.5 Formatting Library

Cippi forms a cup

5.5.1 Formatting Functions

C++20 supports the following formatting functions:

Formatting Functions

Function	Description
`std::format`	Returns the formatted string
`std::format_to`	Writes the result to the output iterator
`std::format_to_n`	Writes at most `n` characters to the output iterator
`std::vformat`	Returns the formatted string
`std::vformat_to`	Writes the result to the output iterator

The functions `std::format` and `std::format_to` are functionally equivalent to their pendants `std::vformat` and `std::vformat_to`, but they differ in a few points:

- `std::format`, `std::_format_to`, and `std::format_to_n`: They require a compile-time value as

format string. This format string can be a constexpr string or a string literal. std::formatted_size returns the number of characters of a compile-time format string.

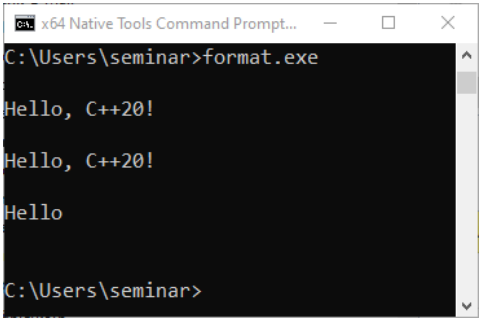
- std::vformat, and std::vformat_t: It's format string can be a lvalue. The arguments must be passed to the variadic function std::make_format_args. e.g.: std::vformat(formatString, std::make_format_args(args)).

The formatting functions accept an arbitrary number of arguments. The following program format.cpp gives a first impression of the functions std::format, std::format_to, and std::format_to_n.

A first impression of std::format, std::format_to, and std::format_to_n

```
1  // format.cpp
2
3  #include <format>
4  #include <iostream>
5  #include <iterator>
6  #include <string>
7
8  int main() {
9
10     std::cout << '\n';
11
12     std::cout << std::format("Hello, C++{}!\n", "20") << '\n';
13
14     std::string buffer;
15
16     std::format_to(
17         std::back_inserter(buffer),
18         "Hello, C++{}!\n",
19         "20");
20
21     std::cout << buffer << '\n';
22
23     buffer.clear();
24
25     std::format_to_n(
26         std::back_inserter(buffer), 5,
27         "Hello, C++{}!\n",
28         "20");
29
30     std::cout << buffer << '\n';
31
32
33     std::cout << '\n';
34
35 }
```

The program directly displays on line 12 the formatted string. However, the calls on lines 16 and 25 use a string as a buffer. Additionally, `std::format_to_n` pushes only five characters onto the buffer.



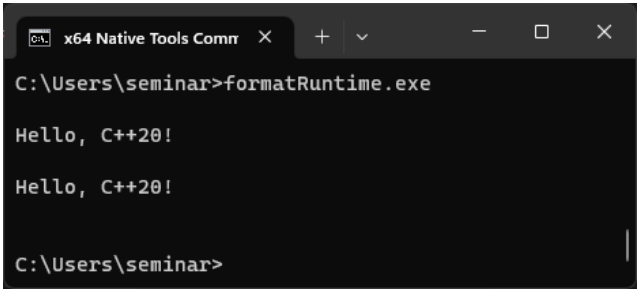
Formatted output

Accordingly, here is the corresponding program using `std::vformat` and `std::vformat_n`.

A first impression of `std::vformat`, and `std::vformat_to`

```
1  // formatRuntime.cpp
2
3  #include <format>
4  #include <iostream>
5  #include <iterator>
6  #include <string>
7
8  int main() {
9
10     std::cout << '\n';
11
12     std::string formatString = "Hello, C++{}!\n";
13
14     std::cout << std::vformat(formatString, std::make_format_args("20")) << '\n';
15
16     std::string buffer;
17
18     std::vformat_to(
19         std::back_inserter(buffer),
20         formatString,
21         std::make_format_args("20"));
22
23     std::cout << buffer << '\n';
24
25 }
```

The `formatString` in lines 12 and 18 is a lvalue.



Formatted output

Presumably, the most exciting part of the formatting functions is the format string (`"Hello, C++{}!\n"`).

5.5.2 Format String

The formatting string syntax is identical for the formatting functions `std::format`, `std::format_to`, `std::format_to_n`, `std::vformat`, and `std::vformat_to`. I use `std::format` in my examples.

- Syntax: **std::format(FormatString, Args)**

The format string **FormatString** consists of

- Ordinary characters (except `{` and `}`)
- Escape sequences `{{` and `}}` that are replaced by `{` and `}`
- Replacement fields

A replacement field has the format `{ }`

- You can use an argument id and a colon inside the replacement field followed by a format specification. Both components are optional.

The argument id allows you to specify the index of the arguments in **Args**. The ids start with `0`. When you don't provide the argument id, the fields are filled in the same order as the arguments are given. Either all replacement fields have to use an argument id or none; i.e., `std::format("{}, {}", "Hello", "World")` and `std::format("{1}, {0}", "World", "Hello")` will both compile, but `std::format("{1}, {}", "World", "Hello")` won't.

`std::formatter` and its specializations define the **format specification** for the argument types.

- Basic types and `std::string`: standard format specification[36] based on Python's format specification[37]
- Chrono types: Chrono format specification[38]. I present them in the Chrono I/O section of the Calendar and Time Zones chapter.
- Other formattable types: User-defined `std::formatter` specialization

The fact that the format strings is a compile time variable, has two interesting consequences: performance and safety.

[36] https://en.cppreference.com/w/cpp/utility/format/formatter#Standard_format_specification
[37] https://docs.python.org/3/library/stdtypes.html#str.format
[38] https://en.cppreference.com/w/cpp/chrono/system_clock/formatter#Format_specification

5.5.2.1 Compile Time

- Performance: If the format string is checked at compile time, there is nothing to do at run time. Consequentially, the three functions `std::format`, `std::format_to`, and `std::format_to_n` promise excellent performance. The prototype library fmt[39] has a few exciting benchmarks.
- Safety: Using a wrong format string at compile time causes a compilation error. On the contrary, using a format string at run time with `std::vformat` or `std::vformat_to` causes a `std::format_error` exception.

I will use the next sections to fill in the theory with practice. Let me start with the argument id and continue with the format specification.

5.5.2.2 Argument ID

Thanks to the argument id, you can reorder or address particular arguments.

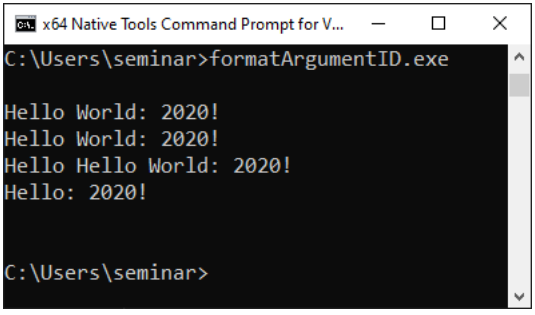
Using the argument id

```
1  // formatArgumentID.cpp
2
3  #include <format>
4  #include <iostream>
5  #include <string>
6
7  int main() {
8
9      std::cout << '\n';
10
11     std::cout << std::format("{} {}: {}!\n", "Hello", "World", 2020);
12
13     std::cout << std::format("{1} {0}: {2}!\n", "World", "Hello", 2020);
14
15     std::cout << std::format("{0} {0} {1}: {2}!\n", "Hello", "World", 2020);
16
17     std::cout << std::format("{0}: {2}!\n", "Hello", "World", 2020);
18
19     std::cout << '\n';
20
21 }
```

Line 11 displays the argument in the given order. On the contrary, line 13 reorders the first and second argument, line 15 shows the first argument twice, and line 17 ignores the second argument.

For completeness, here is the output of the program:

[39] https://github.com/fmtlib/fmt



Applying the argument id

Applying the argument id with the format specification makes formatting of text in C++20 very powerful.

5.5.2.3 Format Specification

I won't present the formal format specification for basic types, string types, or chrono types. For basic types and `std::string`, read the full details here: standard format specification[40]. Accordingly, you can find the details of chrono types here: chrono format specification[41].

Instead, I present a pragmatic description of the format string for basic types, string types, and chrono types. The presentation of chrono types is in the Calendar and Time Zones chapter.

A simplified format specification for basic types and string types

```
fill_align(opt) sign(opt) #(opt) 0(opt) width(opt) precision(opt) L(opt) type(opt)
```

All parts are optional (opt). The following few sections present the features of this format specification.

5.5.2.3.1 Fill Character and Alignment

The fill character is optional (any character except `{` or `}`) and followed by an alignment specification. To us the fill character you must specify the alignment.

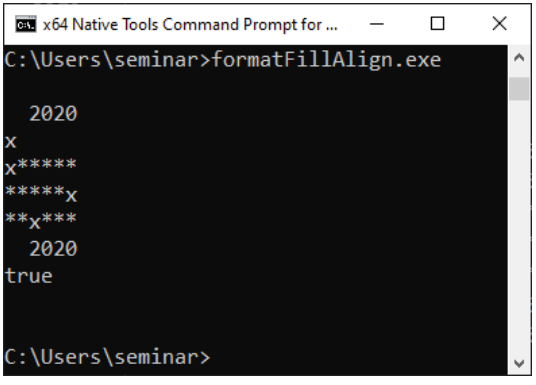
- Fill character: by default, space is used
- Alignment:
  - `<`: left (default for non-numbers)
  - `>`: right (default for numbers)
  - `^`: center

[40] https://en.cppreference.com/w/cpp/utility/format/formatter#Standard_format_specification
[41] https://en.cppreference.com/w/cpp/chrono/system_clock/formatter#Format_specification

Applying the fill character and alignment

```
// formatFillAlign.cpp

#include <format>
#include <iostream>

int main() {

    std::cout << '\n';

    int num = 2020;

    std::cout << std::format("{:6}", num) << '\n';
    std::cout << std::format("{:6}", 'x') << '\n';
    std::cout << std::format("{:*<6}", 'x') << '\n';
    std::cout << std::format("{:*>6}", 'x') << '\n';
    std::cout << std::format("{:*^6}", 'x') << '\n';
    std::cout << std::format("{:6d}", num) << '\n';
    std::cout << std::format("{:6}", true) << '\n';

    std::cout << '\n';

}
```

The default alignment depends on the used types. In contrast to the iostream operator, boolean values are displayed as `true` or `false`.



Applying the fill character and alignment

5.5.2.3.2 Sign, #, and 0

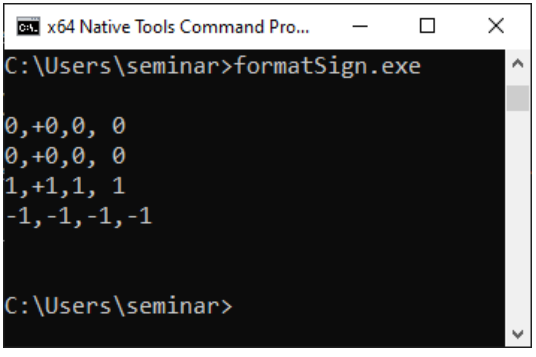
The sign, `#`, and `0` character is only valid when an integer or floating-point type is used.

The sign can have the following values:

- `+`: sign is used for zero and positive numbers
- `-`: sign is only used for negative numbers (default)
- space: leading space is used for non-negative numbers and a minus sign for negative numbers

Applying the sign character

```
// formatSign.cpp

#include <format>
#include <iostream>

int main() {

    std::cout << '\n';

    std::cout << std::format("{0:},{0:+},{0:-},{0: }", 0) << '\n';
    std::cout << std::format("{0:},{0:+},{0:-},{0: }", -0) << '\n';
    std::cout << std::format("{0:},{0:+},{0:-},{0: }", 1) << '\n';
    std::cout << std::format("{0:},{0:+},{0:-},{0: }", -1) << '\n';

    std::cout << '\n';

}
```

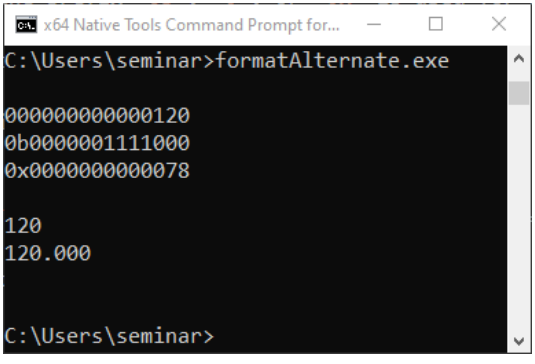


Applying the sign character

The `#` causes the alternative form:

- For integer types, the prefix `0b`, `0`, or `0x` is used for binary, octal, or hexadecimal presented types
- For floating-point types, a decimal point is always used
- `0`: pads with leading zeros

```cpp
// formatAlternate.cpp

#include <format>
#include <iostream>

int main() {

    std::cout << '\n';

    std::cout << std::format("{:#015}", 0x78) << '\n';
    std::cout << std::format("{:#015b}", 0x78) << '\n';
    std::cout << std::format("{:#015x}", 0x78) << '\n';

    std::cout << '\n';

    std::cout << std::format("{:g}", 120.0) << '\n';
    std::cout << std::format("{:#g}", 120.0) << '\n';

    std::cout << '\n';

}
```



Applying the `#` and the `0` characters

5.5.2.3.3 Width and Precision

You can specify the width and the precision of your argument. The width specifier can be applied to numbers, and the precision to floating-point numbers and strings. For floating-point types, the precision specifies the formatting precision; for strings, the precision specifies how many characters are used and so, ultimately, trimming the string. It does not affect a string if the precision is greater than the length of the string.

- width: you can use either a positive decimal number or a replacement field (`{}` or `{n}`). When given, `n` specifies the minimum width.

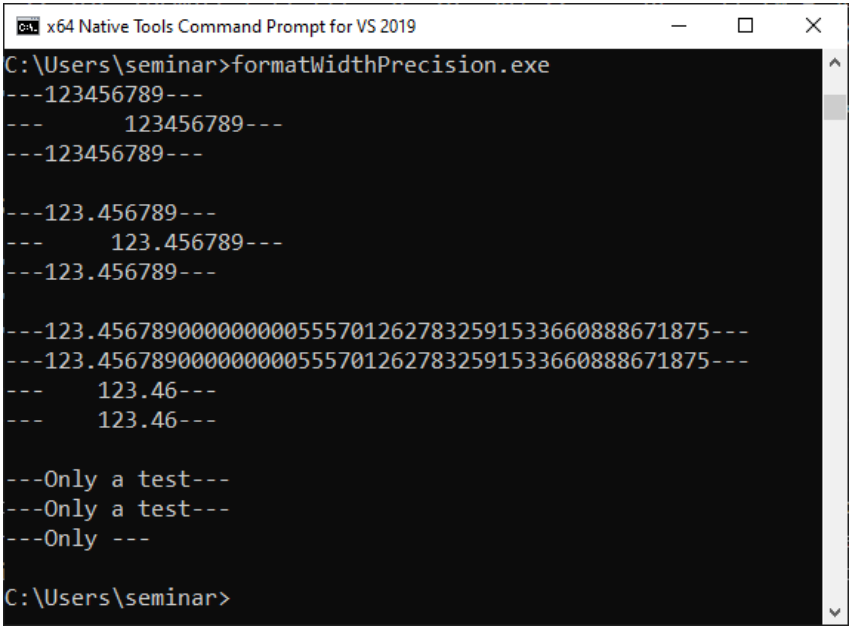
- precision: you can use a period (.) followed by a non-negative decimal number or a replacement field.

A few examples should help you grasp the basics:

Applying the width and precision specifier

```
// formatWidthPrecision.cpp

#include <format>
#include <iostream>
#include <string>

int main() {

    int i = 123456789;
    double d = 123.456789;

    std::cout << "---" << std::format("{}", i) << "---\n";
    std::cout << "---" << std::format("{:15}", i) << "---\n"; // (w = 15)
    std::cout << "---" << std::format("{:}", i) << "---\n";   // (w = 15)

    std::cout << '\n';

    std::cout << "---" << std::format("{}", d) << "---\n";
    std::cout << "---" << std::format("{:15}", d) << "---\n"; // (w = 15)
    std::cout << "---" << std::format("{:}", d) << "---\n";   // (w = 15)

    std::cout << '\n';

    std::string s= "Only a test";

    std::cout << "---" << std::format("{:10.50}", d) << "---\n"; // (w = 10, p = 50)
    std::cout << "---" << std::format("{:{}.{}}", d, 10, 50) << "---\n";  // (w = 10,
                                                                          //  p = 50)
    std::cout << "---" << std::format("{:10.5}", d) << "---\n";  // (w = 10, p = 5)
    std::cout << "---" << std::format("{:{}.{}}", d, 10, 5) << "---\n";  // (w = 10,
                                                                          //  p = 5)

    std::cout << '\n';

    std::cout << "---" << std::format("{:.500}", s) << "---\n";      // (p = 500)
    std::cout << "---" << std::format("{:.{}}", s, 500) << "---\n";  // (p = 500)
    std::cout << "---" << std::format("{:.5}", s) << "---\n";        // (p = 5)

}
```

The `w` character in the source code stands for the width; similarly, the `p` character for the precision. I have a few interesting observations about the program. No extra spaces are added when you specify the width with a replacement field (line 14). When you specify a precision higher than the length of the displayed `double` (lines 26 and 27), the length of the displayed value reflects the precision. This observation does not hold for a string (lines 35 and 36).



Applying the width and precision specifiers

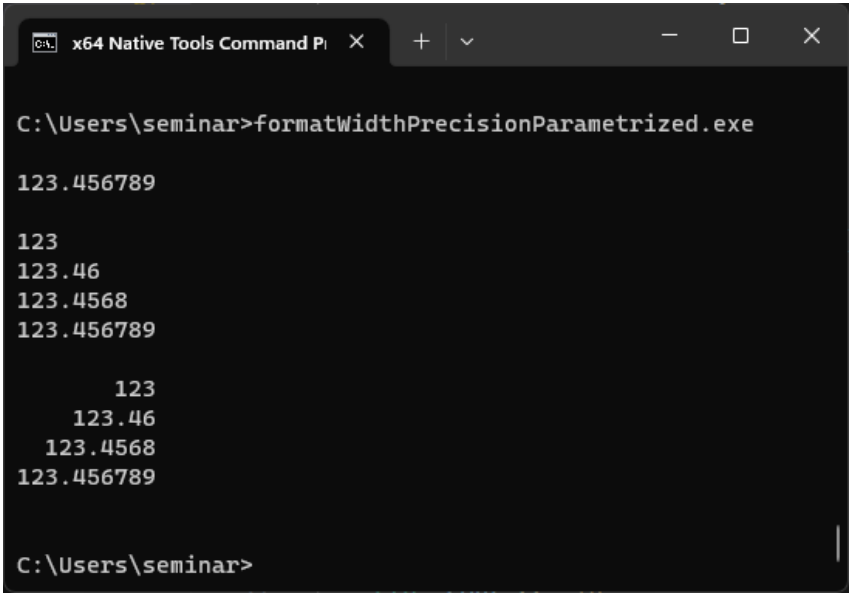
Additionally, you can also parametrize the width and the precision.

Applying the width and precision specifier

```
1  // formatWidthPrecisionParametrized.cpp
2
3  #include <format>
4  #include <iostream>
5
6  int main() {
7
8      std::cout << '\n';
9
10     double doub = 123.456789;
11
12     std::cout << std::format("{:}\n", doub);
13
14     std::cout << '\n';
```

```
15
16      for (auto precision: {3, 5, 7, 9}) {
17         std::cout << std::format("{:.{}}\n", doub, precision);
18      }
19
20      std::cout << '\n';
21
22      int width = 10;
23      for (auto precision: {3, 5, 7, 9}) {
24         std::cout << std::format("{:{}.{}}\n", doub, width, precision);
25      }
26
27      std::cout << '\n';
28
29  }
```

Program `formatWidthPrecisionParametrized.cpp` displays the double `doub` in various ways. Line 12 applies the default. Line 17 varies the `precision` from 3 to 9. The last argument of the format string goes into the inner `{}` of the format specifier `{:.{}}`. Finally, line 24 sets the width of the displayed doubles to 10.



Applying the parametrized width and precision specifiers

5.5.2.3.4 Type

In general, the compiler deduces the type of the value used. But sometimes, you want to specify the type. These are the most important type specifications:

- Strings: `s`
- Integers:
  - `b`: binary format
  - `B`: same as `b` but base Prefix is `0B`
  - `d`: decimal format
  - `o`: octal format
  - `x`: hexadecimal format
  - `X`: same as `x`, but base prefix is `0X`
- `char` and `wchar_t`:
  - `b`, `B`, `d`, `o`, `x`, `X`: such as integers
- `bool`:
  - `s`: `true` or `false`
  - `b`, `B`, `d`, `o`, `x`, `X`: such as integers
- Floating-point:
  - `e`: exponential format
  - `E`: same as `e`, but the exponent is written with `E`
  - `f`, `F`: fixed point; precision is 6
  - `g`, `G`: precision 6 but exponent is written with `E`
- Pointer:
  - `p`: hexadecimal notation of its address

Only `void`, `const void`, and `std::nullptr_t` pointer types are valid. If you want to display the address of an arbitrary pointer, you must cast it to `(const) void*`.

Output of pointers

```
double d = 123.456789;

std::format("{}", &d);                          // ERROR
std::format("{}", static_cast<void*>(&d));      // okay
std::format("{}", static_cast<const void*>(&d)); // okay
std::format("{}", nullptr);                     // okay
```

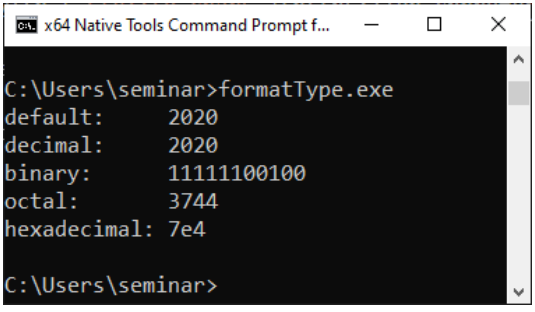
When you don't specify the type, the values are displayed as follows. A string is displayed as a string, an integer in decimal format, a character as a character, and a floating-point value with `std::to_chars`[42].

The type specifiers allow you to easily display an `int` in a different number system.

[42] https://en.cppreference.com/w/cpp/utility/to_chars

Applying the type specifier

```
// formatType.cpp

#include <format>
#include <iostream>

int main() {

    int num{2020};

    std::cout << "default:     " << std::format("{:}", num) << '\n';
    std::cout << "decimal:     " << std::format("{:d}", num) << '\n';
    std::cout << "binary:      " << std::format("{:b}", num) << '\n';
    std::cout << "octal:       " << std::format("{:o}", num) << '\n';
    std::cout << "hexadecimal: " << std::format("{:x}", num) << '\n';

}
```



Applying the type specifier

So far, I've formatted basic types and strings. Additionally, you can format user-defined types.

5.5.3 User-Defined Types

I have to specialize the class `std::formatter`[43] for the user-defined type. In particular, I must implement the member functions `parse`, and `format`.

- **parse**: This function parses the format string and throws a `std::format_error` in case of an error. The function `parse` should be `constexpr` to enable compile-time parsing. It accepts a parse context (`std::format_parse_context`) and should return the last character for the format specifier (the closing `}`). When you don't use a format specifier, this is also the first character of the format specifier.

The following lines show a few examples of the first character of the format specifier:

[43] https://en.cppreference.com/w/cpp/utility/format/formatter

First character of a format specifier

```
"{}"          // context.begin() points to `}`
"{:}"         // context.begin() points to `}`
"{0:d}"       // context.begin() points to `d}`
"{:5.3f}"     // context.begin() points to: `5.3f}`
"{:x}"        // context.begin() points to `x}`
"{0} {0}"     // context.begin() points to: "} {0}"
```

`context.begin()` points to the first character of the format specifier, and `context.end()` to the last character of the entire format string. When you provide a format specifier, you have to parse all between `context.begin()` and `context.end()` and return the position of the closing `}`.

- **format**: This function should be `const`. It gets the value `val` and the format `std::format_context context`. `format` formats the value `val` and writes it, according to the parsed format, to `context.out()`. The `constext.out()` return value can be directly fed into `std::format_to`. `std::format_to` has to return the new position for further output. It returns an iterator that represents the end of the output.

Let me apply the theory and start with the first example.

5.5.3.1 A Formatter for a Single Value

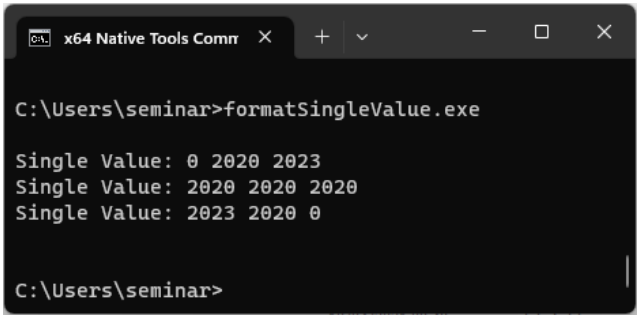
A formatter for a single value

```
1  // formatSingleValue.cpp
2
3  #include <format>
4  #include <iostream>
5
6  class SingleValue {
7   public:
8     SingleValue() = default;
9     explicit SingleValue(int s): singleValue{s} {}
10    int getValue() const {
11      return singleValue;
12    }
13  private:
14    int singleValue{};
15 };
16
17 template<>
18 struct std::formatter<SingleValue> {
19   constexpr auto parse(std::format_parse_context& context) {
20     return context.begin();
21   }
```

```
22    auto format(const SingleValue& sVal, std::format_context& context) const {
23      return std::format_to(context.out(), "{}", sVal.getValue());
24    }
25  };
26
27  int main() {
28
29    std::cout << '\n';
30
31    SingleValue sVal0;
32    SingleValue sVal2020{2020};
33    SingleValue sVal2023{2023};
34
35    std::cout << std::format("Single Value: {} {} {}\n", sVal0, sVal2020, sVal2023);
36    std::cout << std::format("Single Value: {1} {1} {1}\n", sVal0, sVal2020, sVal2023);
37    std::cout << std::format("Single Value: {2} {1} {0}\n", sVal0, sVal2020, sVal2023);
38
39    std::cout << '\n';
40
41  }
```

`SingleValue` (line 6) is a class having only one value. The member function `getValue` (line 10) returns this value. I specialize `std::formatter` (line 17) on `SingleValue`. This specialization has the member functions `parse` (line 19) and `format` (line 22). `parse` returns the end of the format specification. The end of the format specification is the closing `}`. `format` formats the value, and `context.out` creates an object passed to `std::format_to`. `format` returns the new position for further output.

Executing this program gives the expected result.



A formatter for a single value

This formatter has a serious dropback. It does not support a format specifier. Let me improve that.

5.5.3.1.1 A Formatter supporting a Format Specifier

Implementing a formatter for a user-defined type is pretty straightforward when you base your formatter on a standard formatter. Basing a user-defined formatter on a standard formatter can be done in two ways.

5.5.3.1.2 Delegation

The following formatter delegates its job to a standard formatter.

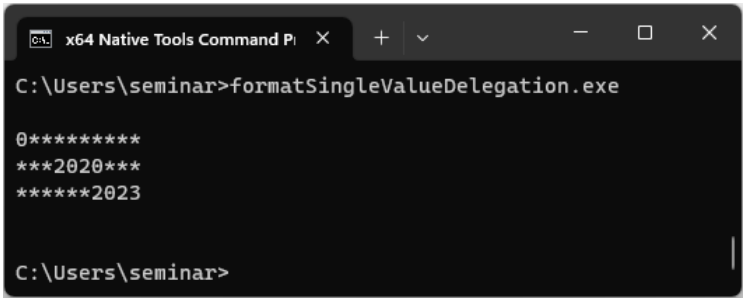
A formatter for a single value supporting a format specifier (delegation)

```
// formatSingleValueDelegation.cpp

#include <format>
#include <iostream>

class SingleValue {
 public:
    SingleValue() = default;
    explicit SingleValue(int s): singleValue{s} {}
    int getValue() const {
        return singleValue;
    }
 private:
    int singleValue{};
};

template<>
struct std::formatter<SingleValue> {

    std::formatter<int> formatter;

    constexpr auto parse(std::format_parse_context& context) {
        return formatter.parse(context);
    }

    auto format(const SingleValue& singleValue, std::format_context& context) const {
        return formatter.format(singleValue.getValue(), context);
    }

};

int main() {

    std::cout << '\n';
```

```
35
36      SingleValue singleValue0;
37      SingleValue singleValue2020{2020};
38      SingleValue singleValue2023{2023};
39
40      std::cout << std::format("{:*<10}", singleValue0) << '\n';
41      std::cout << std::format("{:*^10}", singleValue2020) << '\n';
42      std::cout << std::format("{:*>10}", singleValue2023) << '\n';
43
44      std::cout << '\n';
45
46  }
```

`std::formatter<SingleValue>` (line 17) has a standard formatter for `int:std::formatter<int> formatter` (line 20). I delegate the parsing job (line 23) to the `formatter` (line 23). Accordingly, the formatting job `format` is also delegated to the `formatter` (line 27).

The program's output shows that the formatter supports fill characters and alignment.



A formatter for a single value supporting a format specifier

5.5.3.1.3 Inheritance

Thanks to inheritance, implementing the formatter for the user-defined type `SingleValue` is a piece of cake.

A formatter for a single value supporting a format specifier (inheritance)

```
1  // formatSingleValueInheritance.cpp
2
3  #include <format>
4  #include <iostream>
5
6  class SingleValue {
7   public:
8      SingleValue() = default;
9      explicit SingleValue(int s): singleValue{s} {}
10     int getValue() const {
11         return singleValue;
12     }
13  private:
14     int singleValue{};
15 };
16
17 template<>
18 struct std::formatter<SingleValue> : std::formatter<int> {
19   auto format(const SingleValue& singleValue, std::format_context& context) const {
20     return std::formatter<int>::format(singleValue.getValue(), context);
21   }
22 };
23
24 int main() {
25
26     std::cout << '\n';
27
28     SingleValue singleValue0;
29     SingleValue singleValue2020{2020};
30     SingleValue singleValue2023{2023};
31
32     std::cout << std::format("{:*<10}", singleValue0) << '\n';
33     std::cout << std::format("{:*^10}", singleValue2020) << '\n';
34     std::cout << std::format("{:*>10}", singleValue2023) << '\n';
35
36     std::cout << '\n';
37
38 }
```

I derive `std::formatter<SingleValue>` from `std::formatter<int>` (line 18). Only the `format` functions must be implemented. The output of this program is identical to the output of the previous program `formatSingleValueDelegation.cpp`.

Delegating to a standard formatter or inheriting from one is a straightforward way to implement a user-defined formatter. This strategy only works for user-defined types having one value.

5.5.3.2 A Formatter for More Values

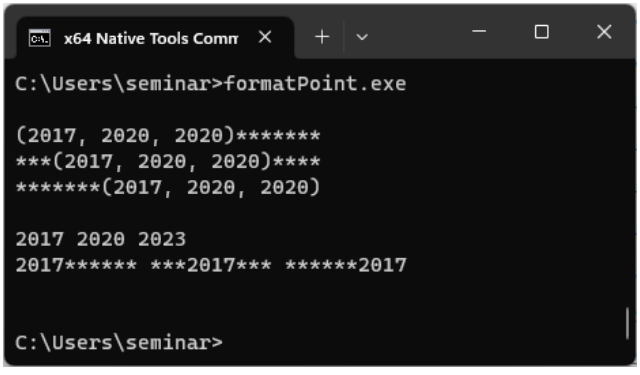
`Point` is a class with three members.

A formatter for a point supporting a format specifier

```
// formatPoint.cpp

#include <format>
#include <iostream>
#include <string>

struct Point {
    int x{2017};
    int y{2020};
    int z{2023};
};

template <>
struct std::formatter<Point> : std::formatter<std::string> {
    auto format(Point point, format_context& context) const {
        return formatter<string>::format(
               std::format("({}, {}, {})", point.x, point.y, point.y), context);
  }
};

int main() {

    std::cout << '\n';

    Point point;

    std::cout << std::format("{:*<25}", point) << '\n';
    std::cout << std::format("{:*^25}", point) << '\n';
    std::cout << std::format("{:*>25}", point) << '\n';

    std::cout << '\n';

    std::cout << std::format("{} {} {}", point.x, point.y, point.z) << '\n';
    std::cout << std::format("{0:*<10} {0:*^10} {0:*>10}", point.x) << '\n';

    std::cout << '\n';
```

```
37
38 }
```

In this case, I derive from the standard formatter `std::formatter<std::string>`. A `std::string_view` is also possible. `std::formatter<Point>` creates the formatted output by calling `format` on `std::formatter`. This function call already gets a formatted string as a value. Consequentially, all format specifiers of `std::string` are applicable (lines 27 - 29). On the contrary, you can also format each value of `Point`. This is exactly happening in lines 33 and 34.



A formatter for a point supporting a format specifier

5.5.4 Internationalization

The formatting functions `std::format*`, and `std::vformat*` have overloads accepting a locale. These overloads allow you to localize your format string.

The following code snippet shows the corresponding overload of `std::format`:

`std::format` overload accepting a locale

```
template< class... Args >
std::string format( const std::locale& loc,
                    std::format_string<Args...> fmt, Args&&... args );
```

To use a given locale, specify `L` before the type specifier in the format string. Now, you apply the locale in each call of `std::format` or set it globally with `std::locale::global`[44].

In the following example, I explicitly apply the German locale on each `std::format` call.

[44] https://en.cppreference.com/w/cpp/locale/locale/global

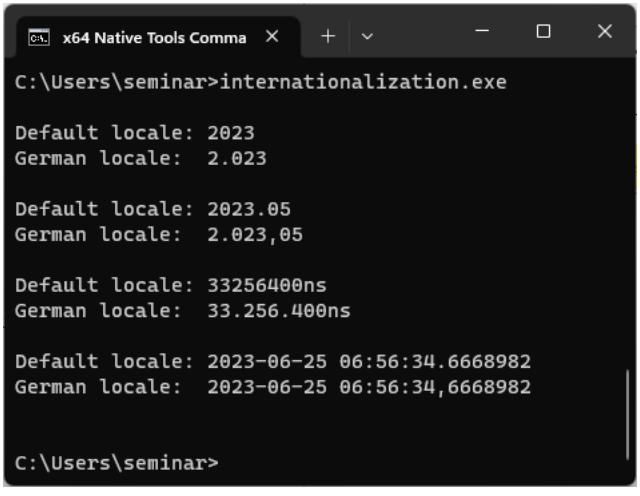
Using the German locale

```cpp
// internationalization.cpp

#include <chrono>
#include <exception>
#include <iostream>
#include <thread>

std::locale createLocale(const std::string& localString) {
  try {
    return std::locale{localString};
  }
  catch (const std::exception& e) {
    return std::locale{""};
  }
}

int main() {

    std::cout << '\n';

    using namespace std::literals;

    std::locale loc = createLocale("de_DE");

    std::cout << "Default locale: " << std::format("{:}", 2023) << '\n';
    std::cout << "German locale:  " << std::format(loc, "{:L}", 2023) << '\n';

    std::cout << '\n';

    std::cout << "Default locale: " << std::format("{:}", 2023.05) << '\n';
    std::cout << "German locale:  " << std::format(loc, "{:L}", 2023.05) << '\n';

    std::cout << '\n';

    auto start = std::chrono::steady_clock::now();
    std::this_thread::sleep_for(33ms);
    auto end = std::chrono::steady_clock::now();

    const auto duration = end - start;

    std::cout << "Default locale: " << std::format("{:}", duration) << '\n';
    std::cout << "German locale:  " << std::format(loc, "{:L}", duration) << '\n';

    std::cout << '\n';
```

```
45
46      const auto now = std::chrono::system_clock::now();
47      std::cout << "Default locale: " << std::format("{}\n", now);
48      std::cout << "German locale:  " << std::format(loc, "{:L}\n", now);
49
50      std::cout << '\n';
51
52  }
```

The function `createLocale` (line 8) creates the German locale. If this fails, it returns the default locale that uses American formatting. I use the German locale in lines 26, 31, 42, and 48. To see the difference, I also applied the `std::format` calls immediately afterward. Consequentially, the local-dependent thousand separator is used for the integral value (line 26), and the locale-dependent decimal point and thousand separator for the floating-point value (line 31). Accordingly, the time duration (line 42) and the time point (line 48) use the given German locale.

The following screenshot shows the output of the program.



Using the German locale

Distilled Information

- The formatting library offers a secure and expandable alternative to the `printf` family and extends the I/O streams.
- The format specification allows you to specify fill letters and text alignment, set the sign, specify the width and the precision of numbers, and specify the data type.
- Thanks to the functions `parse` and `format`, the formatting of a user-defined type can be tailored to your needs.
- The formatting functions `std::format*`, and `std::vformat*` have overloads accepting a locale.

5.6 Calendar and Time Zones

Cippi studies the calendar

To get the most out of the chapter about calendars and time zones, a basic understanding of the chrono terminology is essential.

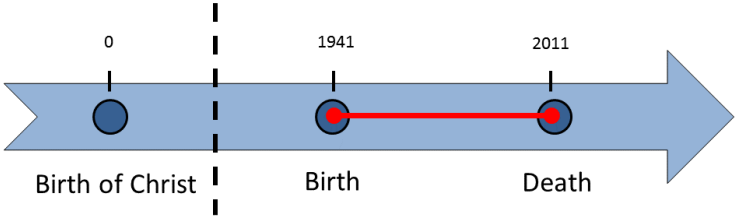
5.6.1 Basic Chrono Terminology

Essentially, the time-zone functionality (C++20) is based on the calendar functionality (C++20) and the calendar functionality (C++20), which are based on the chrono functionality (C++11). Consequently, this basic chrono terminology starts with the three C++11 components time point, time duration, and clock.

- A **time point** is defined by a starting point, the so-called epoch, and additional time duration since the epoch.
- A **time duration** is the difference between two time points. The number of ticks of the clock defines the time duration.
- A **clock** consists of a starting point (epoch) and a tick to calculate the current time point.

You can subtract time points and get a time duration. You get a new time point when you add a time duration to a time point. A year is the typical accuracy of the clock and the measure of the time duration.

I will use the three concepts to present the lifetime of the 2011 died father of the programming language C: Dennis Ritchie. For simplicity reasons, I'm only interested in the years.



Lifetime of Dennis Ritchie

Dennis Ritchie became 70 years old. The birth of Christ is the epoch. Combining the epoch with the time duration gives me the time points 1941 and 2011. Subtraction of the timepoint 1941 from 2011 provides the time duration in Year's accuracy.

C++11 has the three clocks `std::chrono::system_clock`, `std::chrono::steady_clock`, and `std::chrono::high_resolution_clock`. The time duration can be positive and negative. Each of the three clocks has a member function `now` for returning the current time point.

C++20 adds new components to the chrono library:

- The **time of day** is the time duration since midnight, split into hours, minutes, seconds, and fractional seconds.
- **Calendar** stands for various calendar dates such as year, month, weekday, or the nth day of a week.
- A **time zone** represents a time specific to a geographic area.
- A **zoned time** combines a time point with a time zone.

Time is a Mysterium

Honestly, time, for me, is a mystery. On the one hand, each of us has an intuitive idea of time; conversly, defining it formally is exceptionally challenging. For example, the three components, time point, time duration, and clock, depend on each other.

First, let me present the basic types and literals.

5.6.2 Basic Types and Literals

For completeness reasons, this section includes the basic types and literals of the previous C++ standards.

5.6.2.1 Clocks

Beside the wall clock `std::chrono::system_clock`[45], the monotonic clock `std::chrono::steady_clock`[46], and the most precise clock `std::chrono::high_resolution_clock`[47] in C++11, C++20 supports five additional clocks. The following table shows the characteristics of all C++ clocks and their epoch.

Clocks and their Epoch (the namespace `std::chrono` is missing)

Clock	Description	Epoch	Leap Seconds	C++Standard
`steady_clock`	Monotonic clock for measurement	Impl. specific		C++11
`system_clock`	Clock of the operating system	1 January 1970	Not included	C++11
`file_clock`	Alias for `file_time_type`[48]	Impl. specific		C++20
`gps_clock`	GPS time (Global Positioning System[49])	6 January 1980	Not included	C++20
`local_t`	Pseudo clock for local time	Without epoch		C++20
`tai_clock`	TAI time (International Atomic Time[50])	1 January 1958	Not included	C++20
`utc_clock`	Coordinated Universal Time (UTC)	1 January 1970	Included	C++20

The clocks `std::chrono::steady_clock`, and `std::chrono::file_clock` have an implementation specified epoch. The epochs of `std::chrono::system_clock`, `std::chrono::gps_clock`, `std::chrono::tai_clock`, and `std::chrono::utc_clock` start at 00:00:00. `std::chrono::file_clock` is the clock for file system entries.

Additionally, C++11 supports the `std::chrono::high_resolution_clock`. This clock is on all implementations not implemented and is an alias for `std::chrono::steady_clock` or `std::chrono::high_resolution_clock`.

You can convert a time point between the clocks.

5.6.2.1.1 Conversion of time points between clocks.

Thanks to the function `std::chrono::clock_cast`, you can convert time points between the clocks having an epoch. These are the clocks `std::chrono::system_clock`, `std::chrono::utc_clock`, `std::chrono::gps_clock`, and `std::chrono::tai_clock`. Additionally, `std::chrono::file_clock` supports conversion.

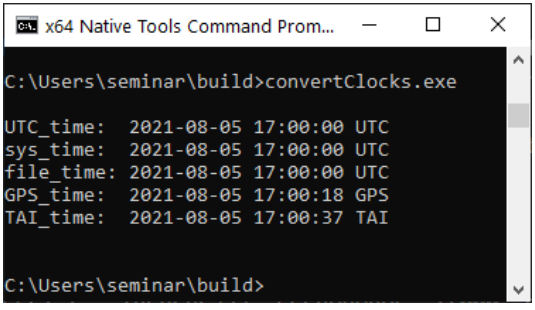
[45] https://www.modernescpp.com/index.php/the-three-clocks
[46] https://www.modernescpp.com/index.php/the-three-clocks
[47] https://www.modernescpp.com/index.php/the-three-clocks
[48] https://en.cppreference.com/w/cpp/filesystem/file_time_type
[49] https://en.wikipedia.org/wiki/Global_Positioning_System
[50] https://en.wikipedia.org/wiki/International_Atomic_Time

The following program converts the time point `2021-8-5 17:00:00` between the various clocks.

Conversion of time point between various clocks

```
1  // convertClocks.cpp
2
3  #include <iostream>
4  #include <sstream>
5  #include <chrono>
6
7  int main() {
8
9    std::cout << '\n';
10
11   using namespace std::literals;
12
13   std::chrono::utc_time<std::chrono::utc_clock::duration> timePoint;
14   std::istringstream{"2021-8-5 17:00:00"} >> std::chrono::parse("%F %T"s, timePoint);
15
16   auto timePointUTC = std::chrono::clock_cast<std::chrono::utc_clock>(timePoint);
17   std::cout << "UTC_time:  " << std::format("{:%F %X %Z}", timePointUTC) << '\n';
18
19   auto timePointSys = std::chrono::clock_cast<std::chrono::system_clock>(timePoint);
20   std::cout << "sys_time:  " << std::format("{:%F %X %Z}", timePointSys) << '\n';
21
22   auto timePointFile = std::chrono::clock_cast<std::chrono::file_clock>(timePoint);
23   std::cout << "file_time: " << std::format("{:%F %X %Z}", timePointFile) << '\n';
24
25   auto timePointGPS = std::chrono::clock_cast<std::chrono::gps_clock>(timePoint);
26   std::cout << "GPS_time:  " << std::format("{:%F %X %Z}", timePointGPS) << '\n';
27
28   auto timePointTAI = std::chrono::clock_cast<std::chrono::tai_clock>(timePoint);
29   std::cout << "TAI_time:  " << std::format("{:%F %X %Z}", timePointTAI) << '\n';
30
31   std::cout << '\n';
32
33 }
```

The function `std::chrono::parse` (line 14) parses the chrono object from the stream. In lines 16, 19, 22, 25, and 28, the `std::chrono::clock_cast` converts the `timePoint` into the specified clock. The following line displays the time point, specifying its date (`%F`), its local time representation (`%X`), and its time zone abbreviation (`%Z`). The section Chrono I/O provides the details about the format string.



Conversion of a time point between various clocks

The output may surprise you. GPS time is 18 seconds ahead of the UTC time. TAI time is 37 seconds ahead of the UTC time and 19 seconds ahead of the GPS time.

5.6.2.2 Time Durations and Literals

C++14 introduced helper types such as `std::chrono::seconds` for time durations and corresponding time literals such as `5s`. C++20 added new helper types. The following table shows all for completeness.

Time Durations and Time Literals

Helper Type	Suffix	Example	Duration	C++ Standard
`std::chrono::nanoseconds`	`ns`	`5ns`		C++14
`std::chrono::microseconds`	`us`	`5us`		C++14
`std::chrono::milliseconds`	`ms`	`5ms`		C++14
`std::chrono::seconds`	`s`	`5s`		C++14
`std::chrono::minutes`	`min`	`5min`		C++14
`std::chrono::hours`	`h`	`5h`		C++14
`std::chrono::days`				C++20
`std::chrono::weeks`				C++20
`std::chrono::months`			30.436875 days	C++20
`std::chrono::years`			365.2425 days (including leap years)	C++20

Often the time duration `std::chrono::days` and the calendar date `std::chrono::day` are mixed up. The same holds for the time duration `std::chrono::years` and the calendar date `std::chrono::year`.

5.6.2.2.1 Distinguish between the time durations `std::chrono::days`, `std::chrono::years`, and the calendar types `std::chrono::day`, `std::chrono::year`

C++20 added two new literals for new calendar types `std::chrono::day` and `std::chrono::year`. The literals `d` and `y` refer to a `std::chrono::day` and `std::chrono::year`

Calendar Type and Literals

Type	Suffix	Example	Available since
`std::chrono::day`	`d`	`5d`	C++20
`std::chrono::year`	`y`	`5y`	C++20

- The day literal represents a day of the month and is unspecified if outside the range [0, 255].
- The year literal represents a year in the Gregorian calendar[51] and is unspecified if outside the range [-32767, 32767].

The following program emphasizes the difference between `std::chrono::days` and `std::chrono::day` and, accordingly, `std::chrono::years` and `std::chrono::year`.

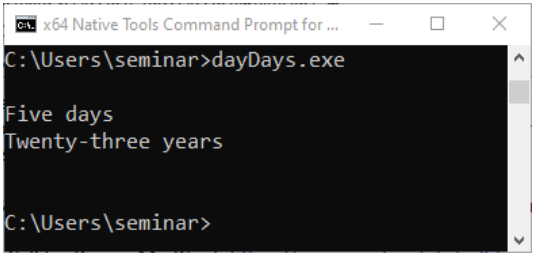
The types day/days and year/years

```
1  // dayDays.cpp
2
3  #include <iostream>
4  #include <chrono>
5
6  int main() {
7
8      std::cout << '\n';
9
10     using namespace std::chrono_literals;
11
12     std::chrono::days days1 = std::chrono::day(30) - std::chrono::day(25);
13     std::chrono::days days2 = 30d - 25d;
14     if ( days1 == days2 &&
15          days1 == std::chrono::days(5)) std::cout << "Five days\n";
16
17     std::chrono::years years1 = std::chrono::year(2021) - std::chrono::year(1998);
18     std::chrono::years years2= 2021y - 1998y;
19     if ( years1 == years2 &&
20          years1 == std::chrono::years(23)) std::cout << "Twenty-three years\n";
21
22     std::cout << '\n';
23
24 }
```

When you subtract two objects of type `std::chrono::day` (line 12), you get an object of type `std::chrono::days`. The same holds for the `std::chrono::year` (lines 17) and `std::chrono::years`.

[51] https://en.wikipedia.org/wiki/Gregorian_calendar

Thanks to the using declaration `using namespace std::chrono_literals` (lines 13 and 18), I can directly specify the time literals for `std::chrono::day` and `std::chrono::year`.



The types day/days and year/years

5.6.2.2.2 Include Literals

There are various ways to include the literals.

Include the time literals

```
using namespace std::literals;
using namespace std::chrono;
using namespace std::chrono_literals;
using namespace std::literals::chrono_literals;
```

- `using namespace std::literals`: includes all C++ literals
- `using namespace std::chrono`: includes the entire namespace `std::chrono`
- `using namespace std::chrono_literals`: includes all time literals
- `using namespace std::literals::chrono_literals`: includes all time literals

The program `literals.cpp` shows the use of different using declarations.

Different using declarations

```
1  // literals.cpp
2
3  #include <chrono>
4  #include <string>
5
6  int main() {
7
8      {
9          using namespace std::literals;
10
11         std::string cppString = "C++ string literal"s;
12         auto aMinute = 60s;
```

```
13          // duration aHour = 0.25h + 15min + 1800s;
14      }
15
16      {
17          using namespace std::chrono;
18
19          // std::string cppString = "C++ string literal"s;
20          auto aMinute = 60s;
21          duration aHour = 0.25h + 15min + 1800s;
22      }
23
24      {
25          using namespace std::chrono_literals;
26
27          // std::string cppString = "C++ String literal"s;
28          auto aMinute = 60s;
29          // duration aHour = 0.25h + 15min + 1800s;
30      }
31
32  }
```

The `using namespace std::literals` declarations enable it to use all built-in literals such as string literal (`"C++ string literal"s` in line 11) or the time literal (`60s` in line 12). `std::chrono::duration` cannot be used unqualified. On the contrary, the using declaration `using namespace std::chrono` allows it to use the time literals and the type `std::chrono::duration` (line 21) unqualified: `duration aHour = 0.25h + 15min + 1800s`. Thanks to the using declaration `using namespace::std::chrono::literals`, all time literals are available.

5.6.2.3 Time Points

Besides the clock and the time duration, the third fundamental type in C++11 was `std::chrono::time_point`.

`std::chrono::time_point`

```
template<typename Clock, typename Duration = typename Clock::duration>
class time_point;
```

A `std::chrono::time_point` depends on the clock and the time duration. C++20 provides aliases for additional time points.

Aliase for time points

Time Point	Description
`std::chrono::local_time<duration>`	Local time point
`std::chrono::local_seconds`	Local time point in seconds
`std::chrono::local_days`	Local time point in days
`std::chrono::sys_time<duration>`	System time point
`std::chrono::sys_seconds`	System time point in seconds
`std::chrono::sys_days`	System time point in days
`std::chrono::utc_time<duration>`	UTC time point
`std::chrono::utc_seconds`	UTC time point in seconds
`std::chrono::tai_time<duration>`	TAI time point
`std::chrono::tai_seconds`	TAI time point in seconds
`std::chrono::gps_time<duration>`	GPS time point
`std::chrono::gps_seconds`	GPS time point in seconds
`std::chrono::file_time<duration>`	Filesystem time point

With the exception of `std::chrono::steady_clock`, you can define a time point with the specified time duration. All but not the clock `std::chrono::file_clock` enables it to specify it for seconds. Additionally, `std::chrono::local_t` and `std::chrono::system_clock` enables to specify it for days.

5.6.3 Time of Day

`std::chrono::hh_mm_ss` is the time duration since midnight, split into hours, minutes, seconds, and fractional seconds. This type is typically used as a formatting tool. First, the following table gives a brief overview of `std::chrono::hh_mm_ss` instance `tOfDay`.

Time of Day

Function	Description
`tOfDay.hours()`	Returns the hour component since midnight
`tOfDay.minutes()`	Returns the minute component since midnight
`tOfDay.seconds()`	Returns the second component since midnight
`tOfDay.subseconds()`	Returns the fractional second component since midnight

Time of Day

Function	Description
`tOfDay.is_negative()`	Returns if the time duration is negative
`tOfDay.to_duration()`	Returns the time duration since midnight If `tOfDay.is_negative()`, returns `-(h + m + s + ss)`, otherwise `h + m + s + ss`
`tOfday.fractional_width`	Number of fractional decimal digits
`std::chrono::make12(hour)`	Returns the 12-hour equivalent of a 24-hour format time
`std::chrono::make24(hour)`	Returns the 24-hour equivalent of a 12-hour format time
`std::chrono::is_am(hour)`	Detects if the 24-hour format time is a.m.
`std::chrono::is_pm(hour)`	Detects if the 24-hour format time is p.m.

Depending on the used time duration, the static member provides the appropriate `tOfDay.fractional_width`. If no such value of `fractional_width` in the range `[0, 18]` exists, then `fractional_width` is `6`. See for example, `std::chrono::duration<int, std::ratio<1, 3>>` in the following table.

fractional_width

Time Duration	Value of `fractional_with`
`std::chrono::hours`	0
`std::chrono::minutes`	0
`std::chrono::seconds`	0
`std::chrono::milliseconds`	3
`std::chrono::microseconds`	6
`std::chrono::nanoseconds`	9
`std::chrono::duration<int, std::ratio<1, 2>>`	1
`std::chrono::duration<int, std::ratio<1, 3>>`	6

The use of the chrono type `std::chrono::hh_mm_ss` is straightforward.

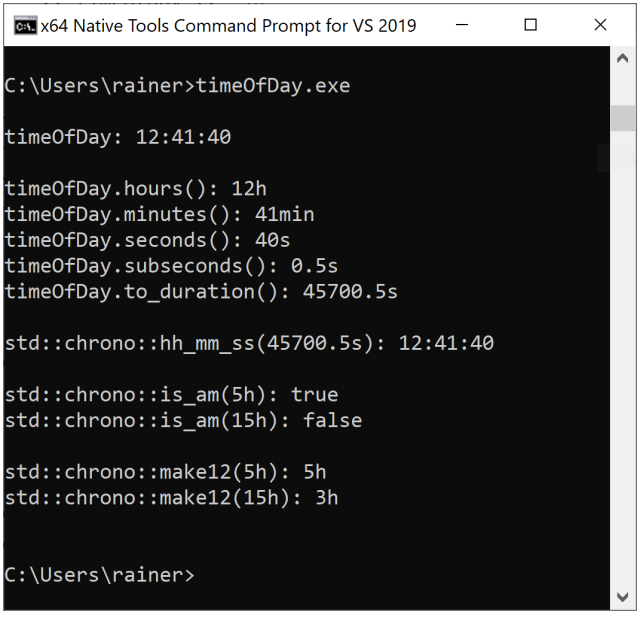
Time of day

```
// timeOfDay.cpp

#include <chrono>
#include <iostream>

int main() {

     using namespace std::chrono_literals;

     std::cout << std::boolalpha << '\n';

     auto timeOfDay = std::chrono::hh_mm_ss(10.5h + 98min + 2020s + 0.5s);

     std::cout<< "timeOfDay: " << timeOfDay << '\n';

     std::cout << '\n';

     std::cout << "timeOfDay.hours(): " << timeOfDay.hours() << '\n';
     std::cout << "timeOfDay.minutes(): " << timeOfDay.minutes() << '\n';
     std::cout << "timeOfDay.seconds(): " << timeOfDay.seconds() << '\n';
     std::cout << "timeOfDay.subseconds(): " << timeOfDay.subseconds() << '\n';
     std::cout << "timeOfDay.to_duration(): " << timeOfDay.to_duration() << '\n';

     std::cout << '\n';

     std::cout << "std::chrono::hh_mm_ss(45700.5s): "
               << std::chrono::hh_mm_ss(45700.5s) << '\n';

     std::cout << '\n';

     std::cout << "std::chrono::is_am(5h): " << std::chrono::is_am(5h) << '\n';
     std::cout << "std::chrono::is_am(15h): " << std::chrono::is_am(15h) << '\n';

     std::cout << '\n';

     std::cout << "std::chrono::make12(5h): " << std::chrono::make12(5h) << '\n';
     std::cout << "std::chrono::make12(15h): " << std::chrono::make12(15h) << '\n';

     std::cout << '\n';

}
```

First, I create in line 12 a new instance of `std::chrono::hh_mm_ss: timeOfDay`. Thanks to the chrono

literals from C++14, I can add a few time durations to initialize a time of day object. With C++20, you can directly output `timeOfDay` (line 14). The rest should be straightforward to read. Lines 18 - 21 display the components of the time since midnight in hours, minutes, seconds, and fractional seconds. Line 22 returns the time duration since midnight in seconds. Line 26 is more interesting: the given seconds correspond to the time displayed in line 15. Lines 31 and 32 return if the given hour is a.m. Lines 36 and 37 return the 12-hour equivalent of the given hour.

Here is the output of the program:



Time of day

5.6.4 Calendar Dates

A new type of the chrono extension in C++20 is a calendar date. C++20 supports various ways to create a calendar date and interact with them. First of all: What is a calendar date?

- A **calendar date** is a date that consists of a year, a month, and a day. Consequently, C++20 has a specific data type `std::chrono::year_month_day`. C++20 has way more to offer. The following table should give you the overview of calendar types before I show you various use cases.

Various calendar types

Calendar Type	Description
`std::chrono::day`	Represents a day of a month
`std::chrono::month`	Represents a month of a year
`std::chrono::year`	Represents a year in the Gregorian calendar
`std::chrono::weekday`	Represents a day of the week in the Gregorian calendar
`std::chrono::weekday_indexed`	Represents the n-th weekday of a month
`std::chrono::weekday_last`	Represents the last weekday of a month
`std::chrono::month_day`	Represents a specific day of a specific month
`std::chrono::month_day_last`	Represents the last day of a specific month
`std::chrono::month_weekday`	Represents the n-th weekday of a specific month
`std::chrono::month_weekday_last`	Represents the last weekday of a specific month
`std::chrono::year_month`	Represents a specific month of a specific year
`std::chrono::year_month_day`	Represents a specific year, month, and day
`std::chrono::year_month_day_last`	Represents the last day of a specific year and month
`std::chrono::year_month_weekday`	Represents the n-th weekday of a specific year and month
`std::chrono::year_month_day_weekday_last`	Represents the last weekday of a specific year and month
`std::chrono::last`	Indicates the last day or weekday of a month

Thanks to the cute syntax, you can use `std::chrono::operator /` to create Gregorian calendar dates.

The calendar data types support various operations. The following table gives an overview. For readability reasons, I ignore the namespace `std::chrono`.

Operations on the calendar types

Calendar Type	`++/--`	`+/-`	difference	`==/!=`	`<=>`
`std::chrono::day`	yes	`days`	yes	yes	yes
`std::chrono::month`	yes	`months`, `years`	yes	yes	yes
`std::chrono::year`	yes	`years`	yes	yes	yes
`std::chrono::weekday`	yes	`days`	yes	yes	yes
`std::chrono::weekday_indexed`				yes	
`std::chrono::weekday_last`				yes	
`std::chrono::month_day`				yes	yes
`std::chrono::month_day_last`			yes	yes	
`std::chrono::month_weekday`				yes	
`std::chrono::month_weekday_last`				yes	

Operations on the calendar types

Calendar Type	++/--	+/-	difference	==/!=	<=>
std::chrono::year_month		months , years	yes	yes	yes
std::chrono::year_month_day		months, years		yes	yes
std::chrono::year_month_day_last		months, years		yes	yes
std::chrono::year_month_weekday		months, years		yes	
std::chrono::year_month_day_weekday_last		months, years		yes	

The increment and decrement operations[52] ++/-- are supported in the prefix and postfix version. Adding or subtraction +/- requires objects of type std::chrono::duration as arguments. You can calculate the **difference** of two objects having the same calendar type. The result is a object of type std::chrono::duration. That means when you build the difference of two objects of calendar type std::chrono::day you get an object of type std::chrono::days. <=> is the new three-way comparison operator.

The following program uses the operations on the calendar types.

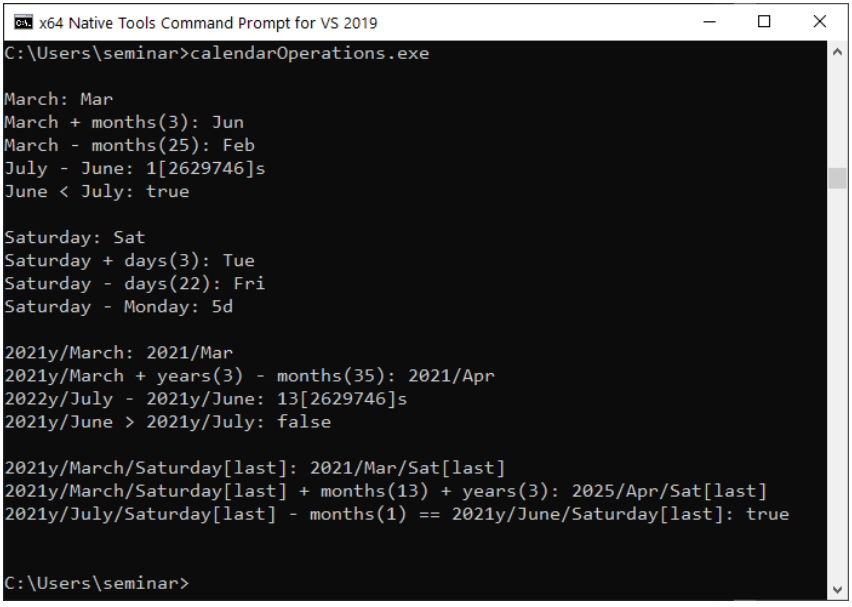
Operations on calendar types

```
1  // calendarOperations.cpp
2
3  #include <chrono>
4  #include <iostream>
5
6  int main() {
7
8    std::cout << '\n';
9
10   using std::chrono::Monday;
11   using std::chrono::Saturday;
12
13   using std::chrono::March;
14   using std::chrono::June;
15   using std::chrono::July;
16
17   using std::chrono::days;
18   using std::chrono::months;
19   using std::chrono::years;
20
21   using std::chrono::last;
22
23   using namespace std::chrono_literals;
24
25   std::cout << std::boolalpha;
```

[52] https://en.cppreference.com/w/cpp/language/operator_incdec

```
26
27    std::cout << "March: " << March << '\n';
28    std::cout << "March + months(3): " << March + months(3) << '\n';
29    std::cout << "March - months(25): " << March - months(25) << '\n';
30    std::cout << "July - June: " <<  July - June << '\n';
31    std::cout << "June < July: " << (June < July) << '\n';
32
33    std::cout << '\n';
34
35    std::cout << "Saturday: " << Saturday << '\n';
36    std::cout << "Saturday + days(3): " << Saturday + days(3) << '\n';
37    std::cout << "Saturday - days(22): " << Saturday - days(22) << '\n';
38    std::cout << "Saturday - Monday: " <<  Saturday - Monday << '\n';
39
40    std::cout << '\n';
41
42    std::cout << "2021y/March: " << 2021y/March << '\n';
43    std::cout << "2021y/March + years(3) - months(35): "
44              << 2021y/March + years(3) - months(35) << '\n';
45    std::cout << "2022y/July - 2021y/June: " << 2022y/July - 2021y/June << '\n';
46    std::cout << "2021y/June > 2021y/July: " << (2021y/June > 2021y/July) << '\n';
47
48    std::cout << '\n';
49
50    std::cout << "2021y/March/Saturday[last]: " << 2021y/March/Saturday[last] << '\n';
51    std::cout << "2021y/March/Saturday[last] + months(13) + years(3): "
52              << 2021y/March/Saturday[last] + months(13) + years(3) << '\n';
53    std::cout << "2021y/July/Saturday[last] - months(1) == 2021y/June/Saturday[last]: "
54              << (2021y/July/Saturday[last] - months(1) == 2021y/June/Saturday[last])
55              << '\n';
56
57    std::cout << '\n';
58
59 }
```

The program performs operations on `std::chrono::month` (line 27), `std::chrono::weekday` (line 35), `std::chrono::year_month` (line 42), and `std::chrono::year_month_weekday_last` (line 50).



Operations with Calendar Type

Adding or subtracting the time duration `std::chrono::months` automatically applies modulo operations (lines 28 and 29). Subtracting two `std::chrono::month` objects returns `1` month. One month has `2629746` seconds (line 31). Accordingly, you can add or subtract a time duration `std::chrono::days` to or from a calendar data `std::chrono::day` (lines 36 and 37). Subtracting two `std::chrono::day` objects returns a `std::chrono::days` object. `std::chrono::year_month` allows the subtraction (line 44), the difference (line 45), and the comparison of time points (line 46). Objects of type `std::chrono::weekday_last` allow the addition/subtraction of the time durations `std::chrono::months` and `std::chrono::years`. In addition, these `std::chrono::weekday_last` objects can be compared.

C++20 supports constants and literals to make using calendar-date types more convenient.

5.6.4.1 Constants and Literals for Calendar Types

Let me start with the constants for `std::chrono::weekday`, and `std::chrono::month`.

std::chrono::weekday

std::chrono::Monday
std::chrono::Thuesday
std::chrono::Wednesday
std::chrono::Thursday
std::chrono::Friday
std::chrono::Saturday
std::chrono::Sunday

std::chrono::month

std::chrono::January
std::chrono::February
std::chrono::March
std::chrono::April
std::chrono::May
std::chrono::June
std::chrono::July
std::chrono::August
std::chrono::September
std::chrono::October
std::chrono::November
std::chrono::December

C++20 supports for calendar types `std::chrono::day` and `std::chrono::year` two new literals: `d' and y'`. You can read more details about it in the section Time Durations and Literals.

Let me start and create a few calendar dates.

5.6.4.2 Create Calendar Dates

The program `createCalendar.cpp` shows various ways to create calendar-related dates.

Create calendar dates

```
1  // createCalendar.cpp
2
3  #include <chrono>
4  #include <iostream>
5
6  int main() {
7
8      std::cout << '\n';
9
```

```
10      using namespace std::chrono_literals;
11
12      using std::chrono::last;
13
14      using std::chrono::year;
15      using std::chrono::month;
16      using std::chrono::day;
17
18      using std::chrono::year_month;
19      using std::chrono::year_month_day;
20      using std::chrono::year_month_day_last;
21      using std::chrono::year_month_weekday;
22      using std::chrono::year_month_weekday_last;
23      using std::chrono::month_weekday;
24      using std::chrono::month_weekday_last;
25      using std::chrono::month_day;
26      using std::chrono::month_day_last;
27      using std::chrono::weekday_last;
28      using std::chrono::weekday;
29
30      using std::chrono::January;
31      using std::chrono::February;
32      using std::chrono::June;
33      using std::chrono::March;
34      using std::chrono::October;
35
36      using std::chrono::Monday;
37      using std::chrono::Thursday;
38      using std::chrono::Sunday;
39
40      constexpr auto yearMonthDay{year(1940)/month(6)/day(26)};
41      std::cout << yearMonthDay << " ";
42      std::cout << year_month_day(1940y, June, 26d) << '\n';
43
44      std::cout << '\n';
45
46      constexpr auto yearMonthDayLast{year(2010)/March/last};
47      std::cout << yearMonthDayLast << " ";
48      std::cout << year_month_day_last(2010y, month_day_last(month(3))) << '\n';
49
50      constexpr auto yearMonthWeekday{year(2020)/March/Thursday[2]};
51      std::cout << yearMonthWeekday << " ";
52      std::cout << year_month_weekday(2020y, month(March), Thursday[2]) << '\n';
53
54      constexpr auto yearMonthWeekdayLast{year(2010)/March/Monday[last]};
```

```
55      std::cout << yearMonthWeekdayLast << " ";
56      std::cout << year_month_weekday_last(2010y, month(March), weekday_last(Monday));
57
58      std::cout << "\n\n";
59
60      constexpr auto day_{day(19)};
61      std::cout << day_  << " ";
62      std::cout << day(19) << '\n';
63
64      constexpr auto month_{month(1)};
65      std::cout << month_  << " ";
66      std::cout << month(1) << '\n';
67
68      constexpr auto year_{year(1988)};
69      std::cout << year_  << " ";
70      std::cout << year(1988) << '\n';
71
72      constexpr auto weekday_{weekday(5)};
73      std::cout << weekday_  << " ";
74      std::cout << weekday(5) << '\n';
75
76      constexpr auto yearMonth{year(1988)/1};
77      std::cout << yearMonth  << " ";
78      std::cout << year_month(year(1988), January) << '\n';
79
80      constexpr auto monthDay{10/day(22)};
81      std::cout << monthDay <<  " ";
82      std::cout << month_day(October, day(22)) << '\n';
83
84      constexpr auto monthDayLast{June/last};
85      std::cout << monthDayLast << " ";
86      std::cout << month_day_last(month(6)) << '\n';
87
88      constexpr auto monthWeekday{2/Monday[3]};
89      std::cout << monthWeekday << " ";
90      std::cout << month_weekday(February, Monday[3]) << '\n';
91
92      constexpr auto monthWeekDayLast{June/Sunday[last]};
93      std::cout << monthWeekDayLast << " ";
94      std::cout << month_weekday_last(June, weekday_last(Sunday)) << '\n';
95
96      std::cout << '\n';
97
98  }
```

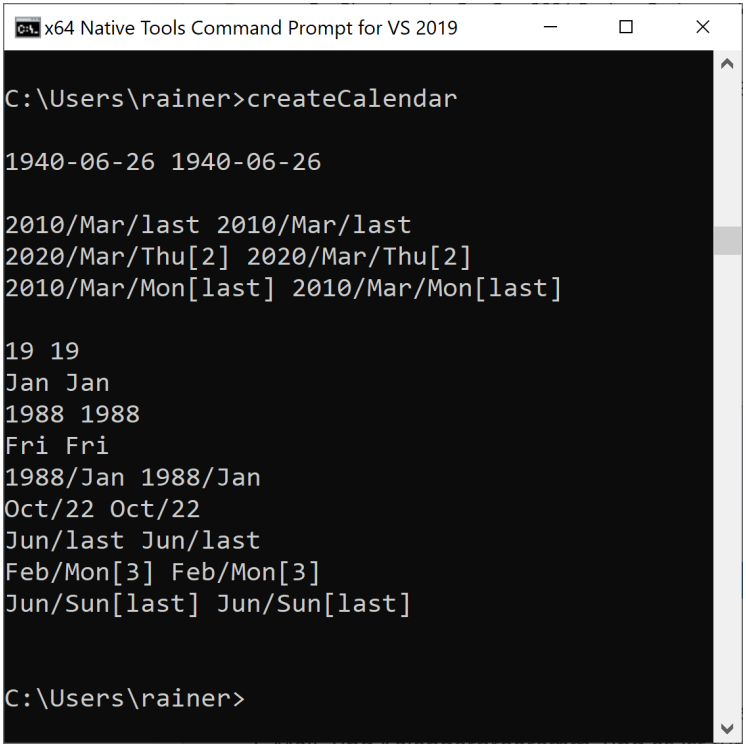
There are two ways to create a calendar date. You can use the so-called cute syntax `yearMonthDay{year(1940)/month(6` (line 40), or you can use the explicit type `date::year_month_day(1940y, June, 26d)` (line 42). To avoid overwhelming you, I will delay my explanation of the cute syntax to the next section. The explicit type is interesting because it uses the date-time literals `1940y`, `26d`, and the predefined constant `June`. This was the obvious part of the program.

Line 46, line 50, and line 54 offer additional ways to create calendar dates.

- Line 46: the last day of March 2010: `{year(2010)/March/last}` or `year_month_day_last(2010y, month_day_last(month(3)))`
- Line 50: the second Thursday of March 2020: `{year(2020)/March/Thursday[2]}` or `year_month_weekday(2020y, month(March), Thursday[2])`
- Line 54: the last Monday of March 2010: `{year(2010)/March/Monday[last]}` or `year_month_weekday_last(2010y, month(March), weekday_last(Monday))`

The remaining calendar types stand for a day (line 60), a month (line 64), or a year (line 68). You can combine them as basic building blocks for fully specified calendar dates, such as in lines 46, 50, or 54

This is the output of the program:



Various calendar days

As promised, let me write about the cute syntax.

5.6.4.3 Cute Syntax

The cute syntax consists of overloaded division operators to specify a calendar date. The overloaded operators support time literals (e.g., `2020y`, `31d`) and `std::chrono::month` constants such as `std::chrono::January`, `std::chrono::February`, ..., `std::chrono::December`).

The following three combinations of year, month, and day are possible using the cute syntax.

Cute syntax

```
year/month/day
day/month/year
month/day/year
```

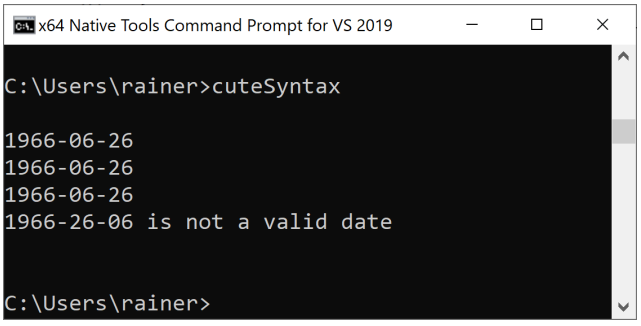
These combinations are not chosen arbitrarily. They are the ones used worldwide. Any other combination is not allowed.

Consequently, when you choose the type `year`, `month`, or `day` for the first argument, the type for the remaining two arguments is no longer necessary, and a number does the job.

Cute syntax

```
// cuteSyntax.cpp

#include <chrono>
#include <iostream>

int main() {

    std::cout << '\n';

    constexpr auto yearMonthDay{std::chrono::year(1966)/6/26};
    std::cout << yearMonthDay << '\n';

    constexpr auto dayMonthYear{std::chrono::day(26)/6/1966};
    std::cout << dayMonthYear << '\n';

    constexpr auto monthDayYear{std::chrono::month(6)/26/1966};
    std::cout << monthDayYear << '\n';

    constexpr auto yearDayMonth{std::chrono::year(1966)/std::chrono::month(26)/6};
    std::cout << yearDayMonth << '\n';

    std::cout << '\n';

}
```

The combination year/day/month (line 19) is not allowed and causes a run-time message.



Use of cute syntax

I assume you want to display a calendar date `{year(2010)/March/last}` in a readable form, for example, 2020-03-31. This is a job for the `local_days` or `sys_days` operator.

5.6.4.4 Displaying Calendar Dates

Thanks to `std::chrono::local_days` or `std::chrono::sys_days`, you can convert calendar dates to a local or a system `std::chrono::time_point`. I use `std::chrono::sys_days` in my example. `std::chrono::sys_days` is based on `std::chrono::system_clock`[53]. Let me convert the calendar dates (lines 18, 22, and 26) from the previous program `createCalendar.cpp`.

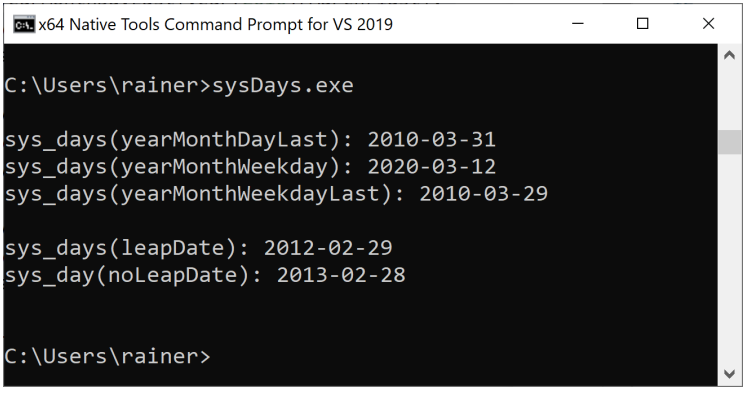
Displaying calendar dates

```
1  // sysDays.cpp
2
3  #include <chrono>
4  #include <iostream>
5
6  int main() {
7
8      std::cout << '\n';
9
10     using std::chrono::last;
11
12     using std::chrono::year;
13     using std::chrono::sys_days;
14
15     using std::chrono::March;
16     using std::chrono::February;
17
18     using std::chrono::Monday;
```

[53] https://en.cppreference.com/w/cpp/chrono/system_clock

```
19      using std::chrono::Thursday;
20
21      constexpr auto yearMonthDayLast{year(2010)/March/last};
22      std::cout << "sys_days(yearMonthDayLast): "
23                << sys_days(yearMonthDayLast)  <<  '\n';
24
25      constexpr auto yearMonthWeekday{year(2020)/March/Thursday[2]};
26      std::cout << "sys_days(yearMonthWeekday): "
27                <<  sys_days(yearMonthWeekday) << '\n';
28
29      constexpr auto yearMonthWeekdayLast{year(2010)/March/Monday[last]};
30      std::cout << "sys_days(yearMonthWeekdayLast): "
31                << sys_days(yearMonthWeekdayLast) << '\n';
32
33      std::cout << '\n';
34
35      constexpr auto leapDate{year(2012)/February/last};
36      std::cout << "sys_days(leapDate): " << sys_days(leapDate) << '\n';
37
38      constexpr auto noLeapDate{year(2013)/February/last};
39      std::cout << "sys_day(noLeapDate): " << sys_days(noLeapDate) << '\n';
40
41      std::cout << '\n';
42
43  }
```

The `std::chrono::last` constant (line 21) lets me quickly determine how many days a month has. The output shows that 2012 is a leap year (line 36), but not 2013 (line 39).



Displaying calendar dates

Suppose you have a calendar date like `year(2100)/2/29`. Your first question may be: Is this date valid?

5.6.4.5 Check if a Date is Valid

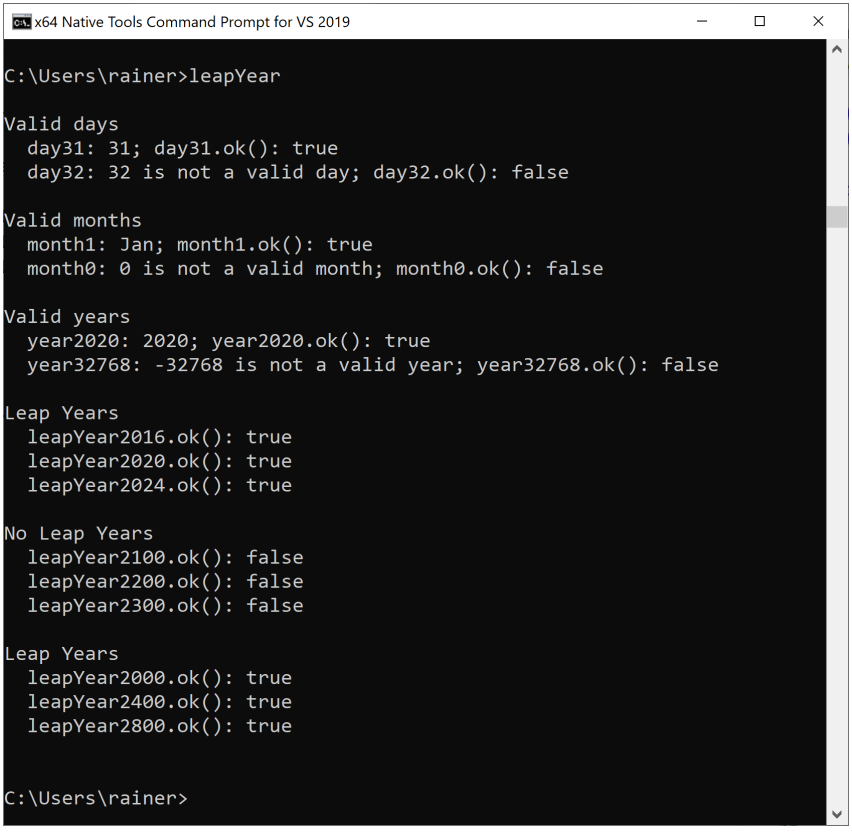
The various calendar types in C++20 have the function `ok`. This function returns `true` if the date is valid.

Checking if a date is valid

```
//  leapYear.cpp

#include <chrono>
#include <iostream>

int main() {

    std::cout << std::boolalpha << '\n';

    std::cout << "Valid days" << '\n';
    std::chrono::day day31(31);
    std::chrono::day day32 = day31 + std::chrono::days(1);
    std::cout << "  day31: " << day31 << "; ";
    std::cout << "day31.ok(): "  << day31.ok() << '\n';
    std::cout << "  day32: " << day32 << "; ";
    std::cout << "day32.ok(): "  << day32.ok() << '\n';


    std::cout << '\n';

    std::cout << "Valid months" << '\n';
    std::chrono::month month1(1);
    std::chrono::month month0(0);
    std::cout << "  month1: " << month1 << "; ";
    std::cout << "month1.ok(): "  << month1.ok() << '\n';
    std::cout << "  month0: " << month0 << "; ";
    std::cout << "month0.ok(): "  << month0.ok() << '\n';

    std::cout << '\n';

    std::cout << "Valid years" << '\n';
    std::chrono::year year2020(2020);
    std::chrono::year year32768(-32768);
    std::cout << "  year2020: " << year2020 << "; ";
    std::cout << "year2020.ok(): "  << year2020.ok() << '\n';
    std::cout << "  year32768: " << year32768 << "; ";
    std::cout << "year32768.ok(): "  << year32768.ok() << '\n';

    std::cout << '\n';
```

```
40
41      std::cout << "Leap Years"  << '\n';
42
43      constexpr auto leapYear2016{std::chrono::year(2016)/2/29};
44      constexpr auto leapYear2020{std::chrono::year(2020)/2/29};
45      constexpr auto leapYear2024{std::chrono::year(2024)/2/29};
46
47      std::cout << "  leapYear2016.ok(): " << leapYear2016.ok() << '\n';
48      std::cout << "  leapYear2020.ok(): " << leapYear2020.ok() << '\n';
49      std::cout << "  leapYear2024.ok(): " << leapYear2024.ok() << '\n';
50
51      std::cout << '\n';
52
53       std::cout << "No Leap Years"  << '\n';
54
55      constexpr auto leapYear2100{std::chrono::year(2100)/2/29};
56      constexpr auto leapYear2200{std::chrono::year(2200)/2/29};
57      constexpr auto leapYear2300{std::chrono::year(2300)/2/29};
58
59      std::cout << "  leapYear2100.ok(): " << leapYear2100.ok() << '\n';
60      std::cout << "  leapYear2200.ok(): " << leapYear2200.ok() << '\n';
61      std::cout << "  leapYear2300.ok(): " << leapYear2300.ok() << '\n';
62
63      std::cout << '\n';
64
65      std::cout << "Leap Years"  << '\n';
66
67      constexpr auto leapYear2000{std::chrono::year(2000)/2/29};
68      constexpr auto leapYear2400{std::chrono::year(2400)/2/29};
69      constexpr auto leapYear2800{std::chrono::year(2800)/2/29};
70
71      std::cout << "  leapYear2000.ok(): " << leapYear2000.ok() << '\n';
72      std::cout << "  leapYear2400.ok(): " << leapYear2400.ok() << '\n';
73      std::cout << "  leapYear2800.ok(): " << leapYear2800.ok() << '\n';
74
75      std::cout << '\n';
76
77  }
```

I check in the program if a given day (line 10), a given month (line 21), or a given year (line 31) is valid. The range of a day is [1, 31], of a month [1, 12], and a year [-32767, 32767]. Consequently, the `ok()` calls on the corresponding values return false. Two facts are interesting when I display various values. First, if the value is not valid, the output shows: “is not a valid day”, “is not a valid month”, “is not a valid year”. Second, month values are displayed in string representation.



Check if a data is valid

You can apply the `ok`-call on a calendar date. Now it's pretty easy to check if a specific calendar date is a leap day and, therefore, the corresponding year a leap year. In the worldwide used Gregorian calendar[54], the following rules apply:

Each year that is exactly divisible by **4** is a **leap year**.

- Except for years that are exactly divisible by **100**. They are **not leap years**.
  - Except for years that are exactly divisible by **400**. They are **leap years**.

Too complicated? The program `leapYears.cpp` exemplifies this rule.

The extended chrono library makes it relatively easy to ask for the time duration between calendar dates.

5.6.4.6 Query Calendar Dates

Without further ado, the following program `queryCalendarDates.cpp` queries a few calendar dates.

[54] https://en.wikipedia.org/wiki/Gregorian_calendar

Query calendar dates

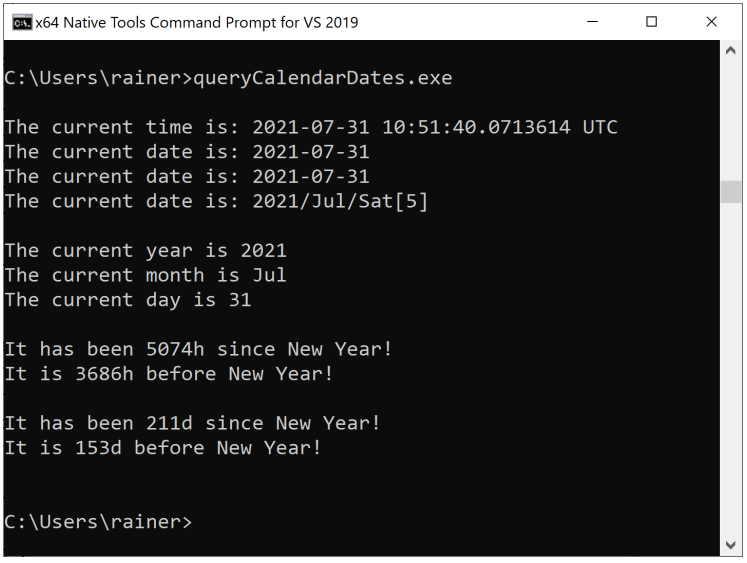
```cpp
// queryCalendarDates.cpp

#include <chrono>
#include <iostream>

int main() {

    std::cout << '\n';

    using std::chrono::floor;

    using std::chrono::January;

    using std::chrono::years;
    using std::chrono::days;
    using std::chrono::hours;

    using std::chrono::year_month_day;
    using std::chrono::year_month_weekday;

    using std::chrono::sys_days;

    auto now = std::chrono::system_clock::now();
    std::cout << "The current time is: " << now << " UTC\n";
    std::cout << "The current date is: " << floor<days>(now) << '\n';
    std::cout << "The current date is: " << year_month_day{floor<days>(now)}
              << '\n';
    std::cout << "The current date is: " << year_month_weekday{floor<days>(now)}
              << '\n';

    std::cout << '\n';


    auto currentDate = year_month_day(floor<days>(now));
    auto currentYear = currentDate.year();
    std::cout << "The current year is " << currentYear << '\n';
    auto currentMonth = currentDate.month();
    std::cout << "The current month is " << currentMonth << '\n';
    auto currentDay = currentDate.day();
    std::cout << "The current day is " << currentDay << '\n';

    std::cout << '\n';

    auto hAfter = floor<hours>(now) - sys_days(January/1/currentYear);
```

```
45      std::cout << "It has been " << hAfter << " since New Year!\n";
46      auto nextYear = currentDate.year() + years(1);
47      auto nextNewYear = sys_days(January/1/nextYear);
48      auto hBefore =  sys_days(January/1/nextYear) - floor<hours>(now);
49      std::cout << "It is " << hBefore << " before New Year!\n";
50
51      std::cout << '\n';
52
53      std::cout << "It has been " << floor<days>(hAfter) << " since New Year!\n";
54      std::cout << "It is " << floor<days>(hBefore) << " before New Year!\n";
55
56      std::cout << '\n';
57
58  }
```

With the C++20 extension, you can directly display a time point, such as `now` (line 24). `std::chrono::floor` rounds the time point down to a day `std::chrono::sys_days`. This value can be used to initialize the calendar type `std::chrono::year_month_day`. Finally, when I put the value into a `std::chrono::year_month_weekday` calendar type, I get the answer that this specific day is the 3rd Tuesday in October.

Of course, I can also ask a calendar date for its components, such as the current year, month, or day (line 34).

Line 44 is the most interesting one. When I subtract from the current date, using hour resolution, the first of January of the current year, I get the number of hours since the new year. Conversely, when I subtract from the first of January of the next year (line 48) the current date, I get the hours to the new year using hour resolution. Maybe you don't like hour resolution. Lines 54 and 54 display the values using day resolution.



Query calendar days

Now, I want to know the weekday of my birthdays.

5.6.4.7 Query Weekdays

Thanks to the extended chrono library, getting the weekday of a given calendar date is pretty easy.

Weekdays of given calendar dates

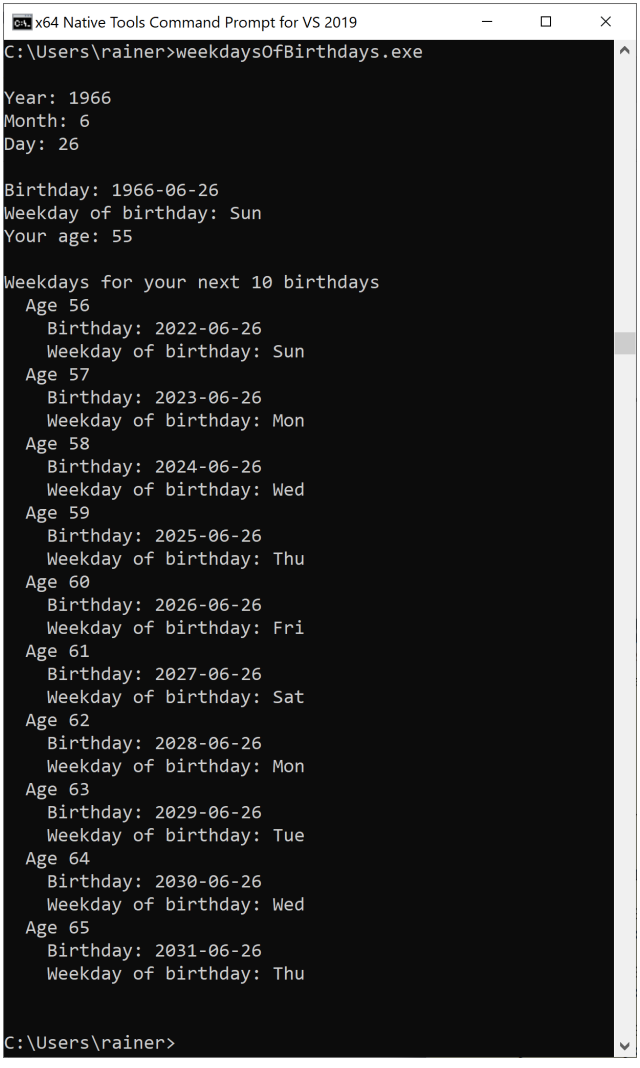
```
1  // weekdaysOfBirthdays.cpp
2
3  #include <chrono>
4  #include <cstdlib>
5  #include <iostream>
6
7  int main() {
8
9      std::cout << '\n';
10
11     int y;
12     int m;
13     int d;
14
15     std::cout << "Year: ";
16     std::cin >> y;
17     std::cout << "Month: ";
18     std::cin >> m;
```

```
19      std::cout << "Day: ";
20      std::cin >> d;
21
22      std::cout << '\n';
23
24      auto birthday = std::chrono::year(y)/std::chrono::month(m)/std::chrono::day(d);
25
26      if (not birthday.ok()) {
27          std::cout << birthday << '\n';
28          std::exit(EXIT_FAILURE);
29      }
30
31      std::cout << "Birthday: " << birthday << '\n';
32      auto birthdayWeekday = std::chrono::year_month_weekday(birthday);
33      std::cout << "Weekday of birthday: " << birthdayWeekday.weekday() << '\n';
34
35      auto currentDate = std::chrono::year_month_day(
36          std::chrono::floor<std::chrono::days>(std::chrono::system_clock::now()));
37      auto currentYear = currentDate.year();
38
39      auto age = (int)currentDate.year() - (int)birthday.year();
40      std::cout << "Your age: " << age << '\n';
41
42      std::cout << '\n';
43
44      std::cout << "Weekdays for your next 10 birthdays" << '\n';
45
46      for (int i = 1, newYear = (int)currentYear; i <= 10;  ++i ) {
47          std::cout << "  Age " <<  ++age << '\n';
48          auto newBirthday = std::chrono::year(++newYear)/
49                             std::chrono::month(m)/std::chrono::day(d);
50          std::cout << "    Birthday: " << newBirthday << '\n';
51          std::cout << "    Weekday of birthday: "
52                    << std::chrono::year_month_weekday(newBirthday).weekday() << '\n';
53      }
54
55      std::cout << '\n';
56
57  }
```

First, the program asks you for your birthday's Year, month, and day (line 15). Based on the input, a calendar date is created (line 24) and checked for validity (line 26). Now I display the weekday of your birthday. I use the calendar date to fill the calendar type `std::chrono::year_month_weekday` (line 32). To get the `int` representation of the calendar type `year`, I must convert it to `int` (line 39). Now

I can display your age. Finally, for each of your next ten birthdays (line 46), the for loop shows the following information: your age, the calendar date, and the weekday. I have to increment the `age` and `newYear` variables.

Here is a run of the program with my birthday.



Weekdays of birthdays

5.6.4.8 Calculating Ordinal Dates

As a last example of the new calendar facility, I want to present the online resource Examples and Recipes[55] from Howard Hinnant, which has about 40 examples of the new chrono functionality. Presumably, the chrono extension in C++20 is not easy to get; therefore, it's essential to have so many examples. You should use these examples as a starting point for further experiments and, therefore, sharpen your understanding. You can also add your recipes.

To get an idea of Examples and Recipes, I want to present a slightly modified program by Roland Bock[56] that calculates ordinal dates.

"*An ordinal date consists of a year and a day of year (1st of January being day 1, 31st of December being day 365 or day 366). The year can be obtained directly from year_month_day. And calculating the day is wonderfully easy. In the code below, we make us of the fact that year_month_day can deal with invalid dates like the 0th of January:*" (Roland Bock)

I added the necessary headers to Roland's program.

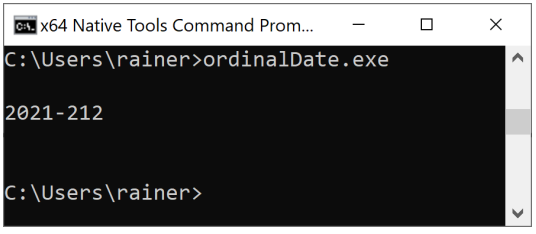
Calculating ordinal dates

```
// ordinalDate.cpp

#include <cassert>
#include <chrono>
#include <iomanip>
#include <iostream>

int main() {

   std::cout << '\n';

   using std::chrono::system_clock;

   using std::chrono::floor;

   using std::chrono::days;

   using std::chrono::January;

   using std::chrono::year_month_day;
   using std::chrono::sys_days;

   const auto time = system_clock::now();
   const auto daypoint = floor<days>(time);
   const auto ymd = year_month_day{daypoint};
```

[55] https://github.com/HowardHinnant/date/wiki/Examples-and-Recipes
[56] https://github.com/rbock

```
26
27      // calculating the year and the day of the year
28      const auto year = ymd.year();
29      const auto year_day = daypoint - sys_days{year/January/0};
30
31      std::cout << year << '-' << std::setfill('0') << std::setw(3)
32                << year_day.count() << '\n';
33
34      // inverse calculation and check
35      assert(ymd == year_month_day{sys_days{year/January/0} + year_day});
36
37      std::cout << '\n';
38
39  }
```

I want to make a few remarks about the program. Line 24 truncates the current time point. The value is used in the following line to initialize a calendar date. Line 29 calculates the time duration between the two time points. Both time points have the resolution day. Finally, `year_day.count()` in line 31 returns the time duration in days.



Caculating ordinal dates

5.6.5 Time Zones

First, a time zone is a region and its entire date history, such as daylight saving time or leap seconds.

Challenges

Dealing with time zones has a few inherent challenges.

- Wintertime and summertime: Many European countries, such as Germany, use a so-called summertime (daylight saving time) and wintertime. The daylight saving time is one hour ahead of the wintertime in Germany.
- More time zones: Countries like China or the United States have different time zones. For example, in the United States, between the Hawaii Standard Time (`UTC-10`) and the Easter Daylight Time (`UTC-4`) is a difference of six hours.
- Time zone differences: Time zone differences are often fractions of hours, such as 30 or 45 minutes. The Australian Central Time is `UTC+9:30`, and the Australian Central Western Standard Time is `UTC+8:45`.
- Time zone abbreviations are ambiguous: The time zone abbreviations are not unique. `ADT` can be Arabic Daylight Time (`UTC+4`) or Atlantic Daylight Time (`UTC-3`).

The time zone library in C++20 is a complete parser of the IANA time-zone database[57]. The following table gives you an overview of the new functionality.

The time-zone data types

Type	Description
`std::chrono::tzdb`	Describes a copy of the IANA time-zone database
`std::chrono::tdzb_list`	Represents a linked list of the `tzdb`

`std::chrono::get_tzdb`
`std::chrono::get_tzdb_list`
`std::chrono::reload_tzdb`
`std::chrono::remote_version`
	Accesses and controls the global time-zone database

`std::chrono::locate_zone`	Locates the time zone based on its name
`std::chrono::current_zone`	Returns the current time zone
`std::chrono::time_zone`	Represents a time zone
`std::chrono::sys_info`	Represents information about a time zone at a specific time point
`std::chrono::local_info`	Represents information about a local time to UNIX time conversion

[57] https://www.iana.org/timezones

The time-zone data types

Type	Description
`std::chrono::zoned_traits`	Class for time zone pointers
`std::chrono::zoned_time`	Represents a time zone and a time point
`std::chrono::leap_second`	Contains information about a leap-second insertion
`std::chrono::time_zone_link`	Represents an alternative name for a time zone
`std::chrono::nonexistent_local_time`	Exception which is thrown if a local time does not exist

The use of the time zone database requires an operating system. Consequently, using the time zone database on a freestanding system typically results in an exception. The time-zone database is updated during the operating system's update, such as a reboot. When your system supports updating the IANA time-zone database[58] without rebooting, you can use `std::chrono::reload_tzdb()`. The new database is atomically added to the front of the linked list. Calls such as `std::chrono::get_tzdb_list()` or `std::chrono::get_tzdb()` parse the front of the list. Consequently, the database queries get the updated database entries. `std::chrono::get_tzdb().version` returns the version of the used database.

The two elementary types for time zones are `std::chrono::time_zone` and `std::chrono::zoned_time`.

The possible time zones are predefined by the IANA time-zone database[59]. The calls `std::chrono::current_zone()`, and `std::chrono::locate_zone(name)` return a pointer to the current or by `name` requested time zone. The call `std::chrono::locate_zone(name)` causes a search for `name` in the database. If the search is unsuccessful, you get a `std::runtime_error` exception.

`std::chrono::zoned_time()` represents a time zone combined with a time point. You can use a system time point, or a local time point as time point. A system time point uses `std::chrono::system_clock` and a local time point uses the pseudo clock `std::chrono::local_t`).

My first example is straightforward. It displays the UTC time and the local time.

5.6.5.1 UTC Time and Local Time

The UTC time or Coordinated Universal Time[60] is the primary time standard worldwide. A computer uses Unix time[61], a very close approximation of UTC. The UNIX time is the number of seconds since the Unix epoch. The Unix epoch is 00:00:00 UTC on 1 January 1970.

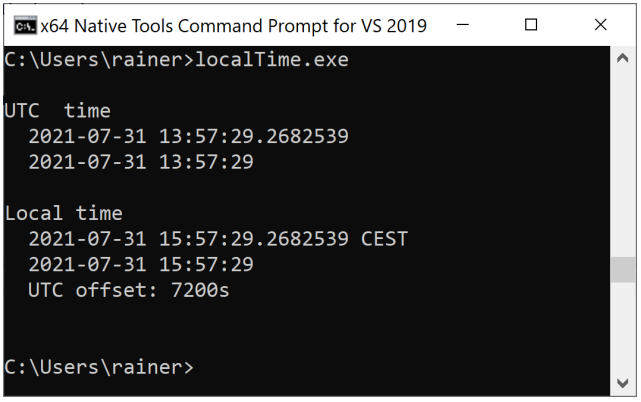
`std::chrono::system_clock::now()` returns in the program `localTime.cpp` the Unix time.

[58] https://www.iana.org/timezones
[59] https://www.iana.org/timezones
[60] https://en.wikipedia.org/wiki/Coordinated_Universal_Time
[61] https://en.wikipedia.org/wiki/Unix_time

Getting the UTC time and local time

```
1  // localTime.cpp
2
3  #include <chrono>
4  #include <iostream>
5
6  int main() {
7
8      std::cout << '\n';
9
10     using std::chrono::floor;
11
12     std::cout << "UTC  time" << '\n';
13     auto utcTime = std::chrono::system_clock::now();
14     std::cout << "   " << utcTime << '\n';
15     std::cout << "   " << floor<std::chrono::seconds>(utcTime) << '\n';
16
17     std::cout << '\n';
18
19     std::cout << "Local time" << '\n';
20     auto localTime = std::chrono::zoned_time(std::chrono::current_zone(), utcTime);
21
22     std::cout << "   " << localTime << '\n';
23     std::cout << "   " << floor<std::chrono::seconds>(localTime.get_local_time())
24                       << '\n';
25
26     auto offset = localTime.get_info().offset;
27     std::cout << "  UTC offset: "  << offset << '\n';
28
29     std::cout << '\n';
30
31 }
```

The code block beginning with line 12 gets the current time point, truncates it to seconds, and displays it. Line 20 creates a `std::chrono::zoned_time localTime`. After that, the call `localTime.get_local_time()` returns the stored time point as a local time. This time point is also truncated to seconds. `localTime` (line 26) can also be used to get information about the time zone. In this case, I'm interested in the offset to the UTC time.



Displaying UTC time and local time

My last example answers a crucial question when I teach in a different time zone: When should I start my online class?

5.6.5.2 Various Time Zones for Online Classes

The program `onlineClass.cpp` answers the following question: How late is it in given time zones when I start an online class at the 7h, 13h, or 17h local time (Germany)?

The online class should start on the 1st of February 2021, taking four hours. Because of daylight saving time, the calendar date is essential to get the correct answer.

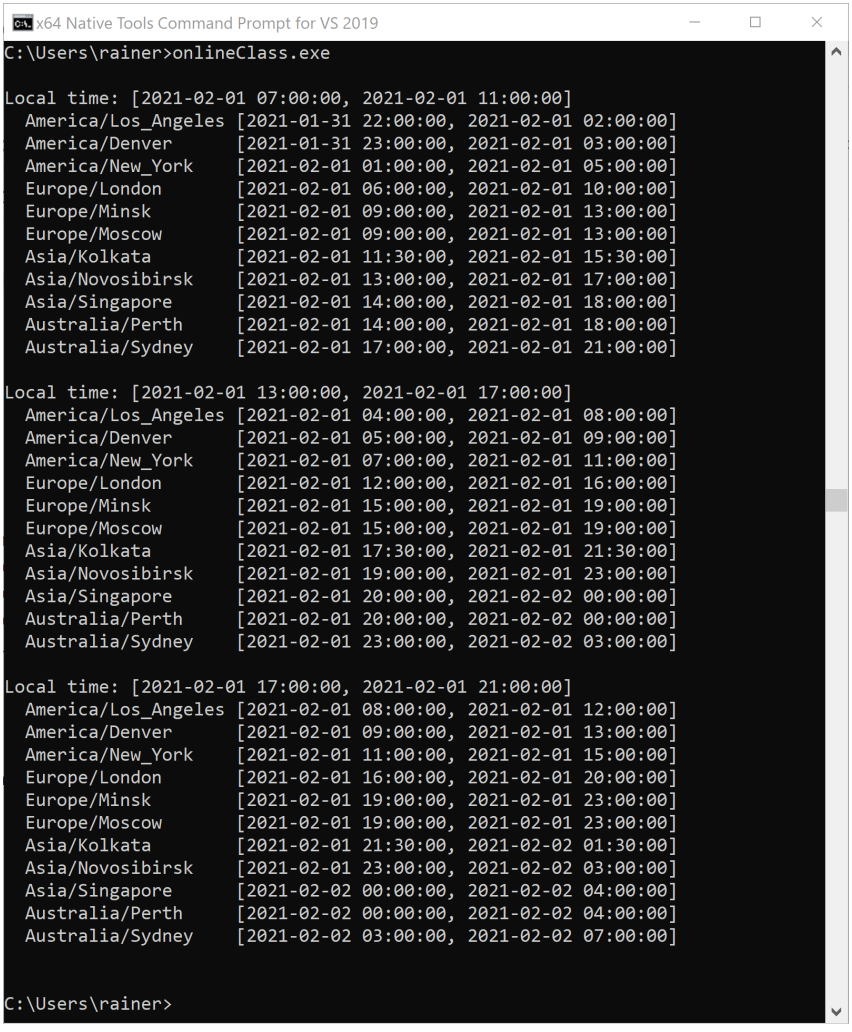
Calculating the time in different time zones

```
1  // onlineClass.cpp
2
3  #include <chrono>
4  #include <algorithm>
5  #include <iomanip>
6  #include <iostream>
7
8  using namespace std::chrono_literals;
9
10 template <typename ZonedTime>
11 auto getMinutes(const ZonedTime& zonedTime) {
12     return std::chrono::floor<std::chrono::minutes>(zonedTime.get_local_time());
13 }
14
15 void printStartEndTimes(const std::chrono::local_days& localDay,
16                         const std::chrono::hours& h,
17                         const std::chrono::hours& durationClass,
18                         const std::initializer_list<std::string>& timeZones ) {
```

```
19
20      std::chrono::zoned_time startDate{std::chrono::current_zone(), localDay + h};
21      std::chrono::zoned_time endDate{std::chrono::current_zone(),
22                                      localDay + h + durationClass};
23      std::cout << "Local time: ["  << getMinutes(startDate) << ", "
24                                     << getMinutes(endDate) << "]" << '\n';
25
26    auto longestStringSize = std::max(timeZones, [](const std::string& a,
27                          const std::string& b) { return a.size() < b.size(); }).size();
28      for (auto timeZone: timeZones) {
29          std::cout << "  " << std::setw(longestStringSize + 1) << std::left
30                    << timeZone
31                    << "[" << getMinutes(std::chrono::zoned_time(timeZone, startDate))
32                    << ", " << getMinutes(std::chrono::zoned_time(timeZone, endDate))
33                    << "]" << '\n';
34
35      }
36  }
37
38  int main() {
39
40      using namespace std::string_literals;
41
42      std::cout << '\n';
43
44      constexpr auto classDay{std::chrono::year(2021)/2/1};
45      constexpr auto durationClass = 4h;
46      auto timeZones = {"America/Los_Angeles"s, "America/Denver"s,
47                        "America/New_York"s, "Europe/London"s,
48                        "Europe/Minsk"s, "Europe/Moscow"s,
49                        "Asia/Kolkata"s, "Asia/Novosibirsk"s,
50                        "Asia/Singapore"s, "Australia/Perth"s,
51                        "Australia/Sydney"s};
52
53      for (auto startTime: {7h, 13h, 17h}) {
54          printStartEndTimes(std::chrono::local_days{classDay}, startTime,
55                             durationClass, timeZones);
56          std::cout << '\n';
57      }
58
59      std::cout << '\n';
60
61  }
```

Before I dive into the functions `getMinutes` (line 10) and `printStartEndTimes` (line 15), let me say a few words about the main function. The main function defines the day of the class, the duration of the class, and all time zones. Finally, the range-based for loop (line 53) iterates through all potential starting points for an online class. All necessary information is displayed thanks to the function `printStartEndTimes` (line 15).

The lines beginning with line 20 calculate the `startDate` and `endDate` of my training by adding the start time and the class duration to the calendar date. Both values are displayed with the help of the function `getMinutes` (line 10). `floor<std::chrono::minutes>(zonedTime.get_local_time())` gets the stored timepoint out of the `std::chrono::zoned_time` and rounds the value down to the minute resolution. Line 26 determines the size of the longest of all time-zone names to align the program's output properly. Line 28 iterates through all time zones and displays the name of the time zone and the beginning and end of each online class. A few calendar dates even cross the day boundaries.



Displaying start and end times in various time zones

5.6.6 Chrono I/O

I/O consists of the reading and writing of the chrono types. The various chrono types support the unformatted writing and the formatted one with the new formatting library. This library also has the function `std::chrono::parse()` that makes reading from a stream quite powerful.

5.6.6.1 Output

Most chrono types, such as time duration, time points, and calendar dates, support direct writing without format specification.

5.6.6.1.1 Unformatted

The following tables show the default output format. Let's start with time durations.

5.6.6.1.2 Time Durations

Time Durations and Time Literals

Time Duration	Literal	Output
std::chrono::nanoseconds	ns	5ns
std::chrono::microseconds	us	5us
std::chrono::milliseconds	ms	5ms
std::chrono::seconds	s	5s
std::chrono::minutes	min	5min
std::chrono::hours	h	5h
std::chrono::days		
std::chrono::weeks		5[604800]s
std::chrono::months		5[2629746]
std::chrono::years		5[31556952]

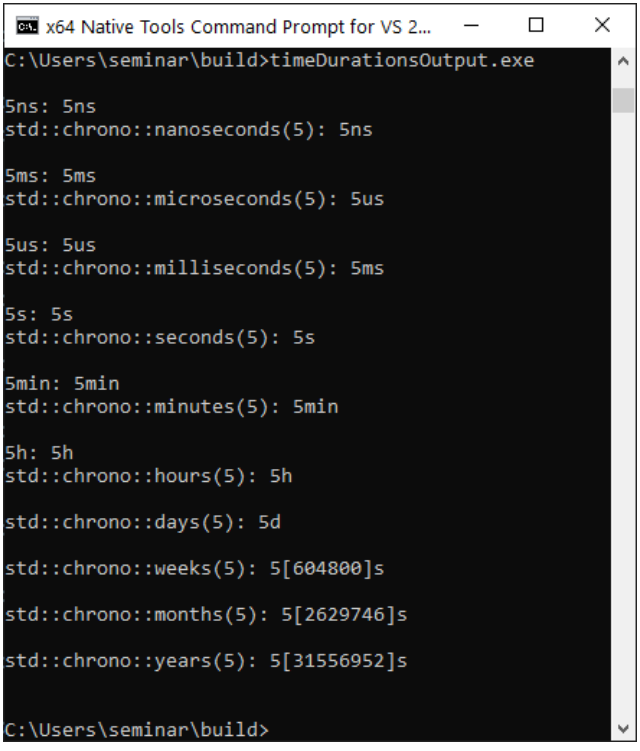
The program displays values for each time duration.

Time durations and their literals

```
1  // timeDurationsOutput.cpp
2
3  #include <chrono>
4  #include <iostream>
5
6  int main() {
7
8      std::cout << '\n';
9
10     using namespace std::chrono_literals;
11
12     std::cout << "5ns: " << 5ns << '\n';
13     std::cout << "std::chrono::nanoseconds(5): "
14               << std::chrono::nanoseconds(5) << '\n';
15
```

```
16      std::cout << '\n';
17
18      std::cout << "5ms: " << 5ms << '\n';
19      std::cout << "std::chrono::microseconds(5): "
20                << std::chrono::microseconds(5) << '\n';
21
22      std::cout << '\n';
23
24      std::cout << "5us: " << 5us << '\n';
25      std::cout << "std::chrono::milliseconds(5): "
26                << std::chrono::milliseconds(5) << '\n';
27
28      std::cout << '\n';
29
30      std::cout << "5s: " << 5s << '\n';
31      std::cout << "std::chrono::seconds(5): " << std::chrono::seconds(5) << '\n';
32
33      std::cout << '\n';
34
35      std::cout << "5min: " << 5min << '\n';
36      std::cout << "std::chrono::minutes(5): " << std::chrono::minutes(5) << '\n';
37
38      std::cout << '\n';
39
40      std::cout << "5h: " << 5h << '\n';
41      std::cout << "std::chrono::hours(5): " << std::chrono::hours(5) << '\n';
42
43      std::cout << '\n';
44
45      std::cout << "std::chrono::days(5): " << std::chrono::days(5) << '\n';
46
47      std::cout << '\n';
48
49      std::cout << "std::chrono::weeks(5): " << std::chrono::weeks(5) << '\n';
50
51      std::cout << '\n';
52
53      std::cout << "std::chrono::months(5): " << std::chrono::months(5) << '\n';
54
55      std::cout << '\n';
56
57      std::cout << "std::chrono::years(5): " << std::chrono::years(5) << '\n';
58
59      std::cout << '\n';
60
```

```
61  }
```



Time durations and their literals

The natural numbers in the square braces of `std::chrono::weeks`, `std::chrono::months`, and `std::chrono::years` represent the number of seconds.

5.6.6.1.3 Time Points

When you use the C++20 clocks static member function `now`, you get the date and the time in the following format.

Current Time with the C++20 clocks

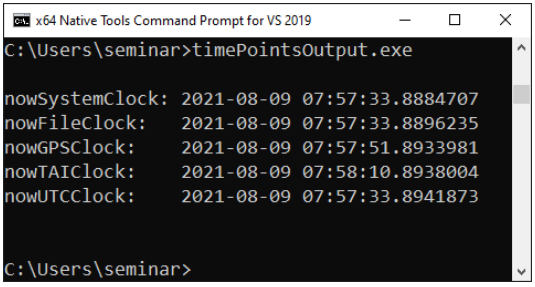
```
year-month-day hours:minutes:seconds
```

The following program shows the current time using all C++20 clocks.

The current time displayed with the C++20 clocks

```
1  // timePointsOutput.cpp
2
3  #include <chrono>
4  #include <iostream>
5
6  int main() {
7
8      std::cout << '\n';
9
10     auto nowSystemClock = std::chrono::system_clock::now();
11     std::cout << "nowSystemClock: " << nowSystemClock << '\n';
12
13     auto nowSteadyClock = std::chrono::steady_clock::now();
14     // std::cout << "nowSteadyClock: " << nowSteadyClock << '\n';    ERROR
15
16     auto nowFileClock = std::chrono::file_clock::now();
17     std::cout << "nowFileClock:   " << nowFileClock << '\n';
18
19     auto nowGPSClock = std::chrono::gps_clock::now();
20     std::cout << "nowGPSClock:    " << nowGPSClock << '\n';
21
22     // auto nowlocal_tClock = std::chrono::local_t::now();           ERROR
23
24     auto nowTAIClock = std::chrono::tai_clock::now();
25     std::cout << "nowTAIClock:    " << nowTAIClock << '\n';
26
27     auto nowUTCClock = std::chrono::utc_clock::now();
28     std::cout << "nowUTCClock:    " << nowUTCClock << '\n';
29
30     std::cout << '\n';
31
32 }
```

The program shows two interesting facts. First, the current time given by the `std::chrono::steady_clock::now()` cannot be displayed (line 14). Second, the pseudo clock `std::chrono::local_t` has not static member function `now()` (line 22).



Current time with the C++20 clocks

The GPS time is 18 seconds ahead of the UTC time. The TAI time is 37 seconds ahead of the UTC time and 19 seconds ahead of the GPS time.

Thanks to the C++17 `std::chrono::floor`[62], you can display the time point in different granularities. In this case, the time point has to be of type `std::chrono::local_time`.

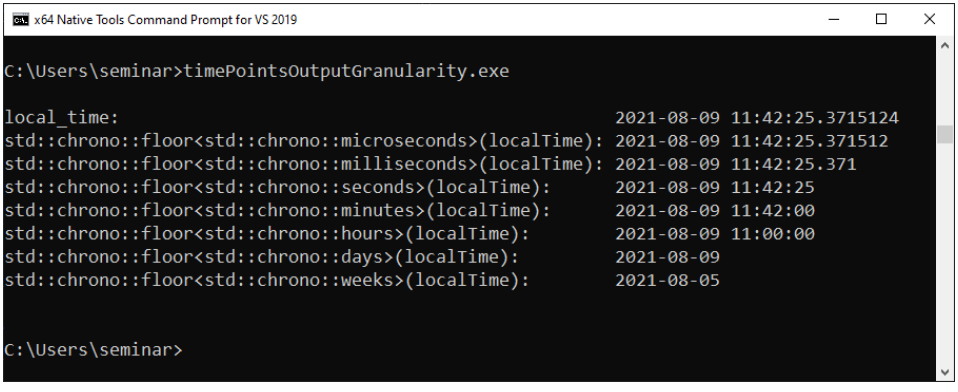
The local time displayed in different resolutions

```
// timePointsOutputGranularity.cpp

#include <chrono>
#include <iostream>

int main() {

    std::cout << '\n';

    auto now = std::chrono::system_clock::now();

    auto zonedTime = std::chrono::zoned_time(std::chrono::current_zone(), now);
    auto localTime = zonedTime.get_local_time();

    std::cout << "local_time:                                          "
              << localTime << '\n';

    std::cout << "std::chrono::floor<std::chrono::microseconds>(localTime): "
              << std::chrono::floor<std::chrono::microseconds>(localTime) << '\n';

    std::cout << "std::chrono::floor<std::chrono::milliseconds>(localTime): "
              << std::chrono::floor<std::chrono::milliseconds>(localTime) << '\n';

    std::cout << "std::chrono::floor<std::chrono::seconds>(localTime):      "
              << std::chrono::floor<std::chrono::seconds>(localTime) << '\n';

```

[62] https://en.cppreference.com/w/cpp/chrono/duration/floor

```
27      std::cout << "std::chrono::floor<std::chrono::minutes>(localTime):      "
28                << std::chrono::floor<std::chrono::minutes>(localTime) << '\n';
29
30      std::cout << "std::chrono::floor<std::chrono::hours>(localTime):        "
31                << std::chrono::floor<std::chrono::hours>(localTime) << '\n';
32
33      std::cout << "std::chrono::floor<std::chrono::days>(localTime):         "
34                << std::chrono::floor<std::chrono::days>(localTime) << '\n';
35
36      std::cout << "std::chrono::floor<std::chrono::weeks>(localTime):        "
37                << std::chrono::floor<std::chrono::weeks>(localTime) << '\n';
38
39      // std::cout << std::chrono::floor<std::chrono::months>(localTime) << '\n'; ERROR
40      // std::cout << std::chrono::floor<std::chrono::years>(localTime) << '\n';  ERROR
41
42      std::cout << '\n';
43
44  }
```

The program displays localTime in different accuracies, starting with the time duration `std::chrono::microseconds` (line 18) and ending with `std::chrono::weeks` (line 36). Curiously, the time durations for `std::chrono::months`, and `std::chrono::years` cannot be displayed, but this will be fixed with C++23.



The local time displayed in different resolutions

5.6.6.1.4 Calendar Dates

The following table shows the format specifiers, including a short description and an example. For the full description, refer to the cppreference.com/chrono/parse[63] page.

[63] https://en.cppreference.com/w/cpp/chrono/parse

Various Calendar Types

Calendar Type	Description	Output Example
`std::chrono::day`	Day	`09`
`std::chrono::month`	Month	`Aug`
`std::chrono::year`	Year	`2021`
`std::chrono::weekday`	Weekday	`Mon`
`std::chrono::weekday_indexed`	nth weekday	`Mon[2]`
`std::chrono::weekday_last`	Last weekday	`Mon[last]`
`std::chrono::month_day`	Day of a month	`Aug/09`
`std::chrono::month_day_last`	Last day of a month	`Aug/last`
`std::chrono::month_weekday`	nth weekday of a month	`Aug/Mon[2]`
`std::chrono::month_weekday_last`	Last weekday of a month	`Aug/Mon[last]`
`std::chrono::year_month`	Month of a years	`2021/Aug`
`std::chrono::year_month_day`	A day of a month of a year	`2021-08-09`
`std::chrono::year_month_day_last`	Last day of a month of a year	`2021/Aug/last`
`std::chrono::year_month_weekday`	nth weekday of a month of a year	`2021/Aug/Mon[2]`
`std::chrono::year_month_day_weekday_last`	Last weekday of a month of a year	`2021/Aug/Mon[last]`

The program `createCalender.cpp` outputs the various calendar dates.

5.6.6.1.5 Formatted

The following table shows the format specifiers, including a short description and an example. Refer to the cppreference.com/chrono/parse[64] page for the full description.

Format Specifiers for Calendar Dates

Specifier	Description	Example
Calendar Date:		
`%c`	Locale's date and time representation	`Mon Aug 9 22:58:04 2021`
`%x`	Locale's date representation	`09/08/21`
`%F`	year-month-day	`2021-08-08`
`%D`	month/day/year	`09/08/21`
Year		
`%Y`	Year	`2021`
`%y`	Year without century	`21`
`%C`	Century as two digits	`20`

[64] https://en.cppreference.com/w/cpp/chrono/parse

Format Specifiers for Calendar Dates

Specifier	Description	Example
Month:		
`%b`, or `%h`	Abbreviated month name	`Aug`
`%B`	Month name	`August`
`%m`	Month	`08`
Week:		
`%W`	Week of the year (01 until 53, week starts Monday)	`31`
`%U`	Week of the year (01 until 53, week starts Sunday)	`31`
Weekday:		
`%a`	Abbreviated weekday name	`Mon`
`%A`	Weekday name	`Monday`
`%w`	Weekday as number (Sunday (0) until Saturday (6))	`1`
`%u`	Weekday as number (Monday (1) until Sunday (7))	`1`
Day:		
`%e`	Day (leading space if necessary)	`9`
`%d`	Day with two digits	`09`

Format Specifiers for Time

Specifier	Description	Example
`%c`	Date and time representation	`Mon Aug 9 22:58:04 2021`
`%X`	Time representation	`22:58:04`
`%r`	12-hour clock time	`10:58:04 PM`
`%T`	hours:minutes:seconds	`22:58:04.435`
`%R`	hours:minutes	`22:58`
`%H`	24-hour clock	`22`
`%I`	12-hour clock	`10`
`%p`	AM or PM (12-hour clock)	`PM`
`%M`	Minute	`58`
`%S`	seconds.subseconds	`04.453`

Other Format Specifier for Chrono

Specifier	Description	Example
%Z	Time zone abbreviation	CEST
%z	Offset (hours and minutes) from UTC	+0200
%j	Day of the year (Starting wiht 001)	221
%q	Unit suffix according to the time's duration	ms
%n	Newline character	\n
%t	Tabulator character	\t
%%	% character	%

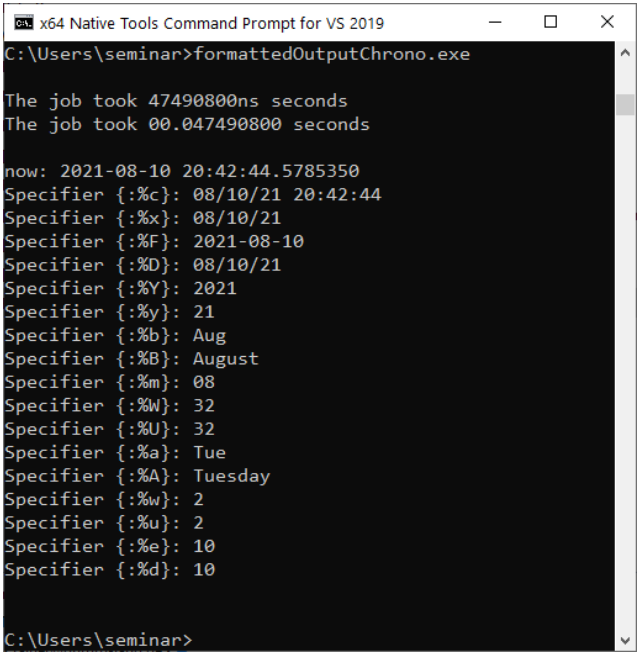
The following program uses the time specifier and calendar date specifiers.

Use of the time specifier and calendar date specifiers

```
// formattedOutputChrono.cpp

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

int main() {

    std::cout << '\n';

    using namespace std::literals;

    auto start = std::chrono::steady_clock::now();
    std::this_thread::sleep_for(33ms);
    auto end = std::chrono::steady_clock::now();
    std::cout << std::format("The job took {} seconds\n", end - start);
    std::cout << std::format("The job took {:%S} seconds\n", end - start);

    std::cout << '\n';

    auto now = std::chrono::system_clock::now();
    std::cout << "now: " << now << '\n';
    std::cout << "Specifier {:%c}: " << std::format("{:%c}\n", now);
    std::cout << "Specifier {:%x}: " << std::format("{:%x}\n", now);
    std::cout << "Specifier {:%F}: " << std::format("{:%F}\n", now);
    std::cout << "Specifier {:%D}: " << std::format("{:%D}\n", now);
    std::cout << "Specifier {:%Y}: " << std::format("{:%Y}\n", now);
    std::cout << "Specifier {:%y}: " << std::format("{:%y}\n", now);
    std::cout << "Specifier {:%b}: " << std::format("{:%b}\n", now);
    std::cout << "Specifier {:%B}: " << std::format("{:%B}\n", now);
```

```
31      std::cout << "Specifier {:%m}: " << std::format("{:%m}\n", now);
32      std::cout << "Specifier {:%W}: " << std::format("{:%W}\n", now);
33      std::cout << "Specifier {:%U}: " << std::format("{:%U}\n", now);
34      std::cout << "Specifier {:%a}: " << std::format("{:%a}\n", now);
35      std::cout << "Specifier {:%A}: " << std::format("{:%A}\n", now);
36      std::cout << "Specifier {:%w}: " << std::format("{:%w}\n", now);
37      std::cout << "Specifier {:%u}: " << std::format("{:%u}\n", now);
38      std::cout << "Specifier {:%e}: " << std::format("{:%e}\n", now);
39      std::cout << "Specifier {:%d}: " << std::format("{:%d}\n", now);
40
41      std::cout << '\n';
42
43  }
```

The call `std::chrono::steady_clock::now()` (lines 13 and 15) determines the current time. You should use the `std::chrono::steady_clock` for measurements because this clock is monotonic and cannot be adjusted, such as `std::chrono::system_clock` (line 21)



Use of the time specifier and calendar date specifiers

You can also apply the format specifier for formatted input.

5.6.6.2 Input

The chrono library supports formatted input in two ways. You can use the function `std::chrono::from_stream`[65] or `std::chrono::parse`[66]. Both functions require an input stream and parse the input into a time point according to the format specification. All format specifier except `%q` for unit suffixed according to the literals for time durations can be used.

5.6.6.2.1 `std::chrono::from_stream`

`std::chrono::from_stream` has overloads for the various clocks and calendar dates.

- Clocks
  - `std::chrono::system_time`
  - `std::chrono::utc_time`
  - `std::chrono::tai_time`
  - `std::chrono::gps_time`
  - `std::chrono::file_time`
  - `std::chrono::local_time`
- Calendar dates
  - `std::chrono::year_month_day`
  - `std::chrono::year_month`
  - `std::chrono::month_day`
  - `std::chrono::weekday`
  - `std::chrono::year`
  - `std::chrono::month`
  - `std::chrono::day`

The various overloads require in the elementary form an input stream `is`, a format string `fmt`, and a time point or a calendar object `chro`: `std::chrono::from_stream(is, fmt, chro)`. The chrono object from the input stream is then parsed according to the format string.

You can also provide an abbreviation `abb` for a time zone and an offset `off` from the UTC time: `std::chrono::from_stream(is, fmt, chro, abb, off)`. The offset has the type `std::chrono::minutes`.

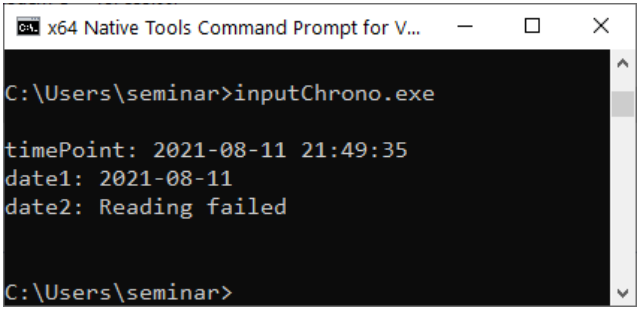
The program `inputChrono.cpp` uses formatted input to read a time point and a calendar date from an input stream.

[65] https://en.cppreference.com/w/cpp/chrono/system_clock/from_stream
[66] https://en.cppreference.com/w/cpp/chrono/parse

Reading chrono objects from an input stream using `std::chrono::from_stream`

```
1  // inputChrono.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <string>
6  #include <sstream>
7
8  int main() {
9
10     std::cout << '\n';
11
12     std::chrono::sys_seconds timePoint;
13     std::istringstream iStream1{"2021-08-11 21:49:35"};
14     std::chrono::from_stream(iStream1, "%F %T", timePoint);
15     if (iStream1) std::cout << "timePoint: "  << timePoint << '\n';
16     else std::cerr << "timepoint: Reading failed\n";
17
18     std::chrono::year_month_day date1;
19     std::istringstream iStream2{"11/08/21"};
20     std::chrono::from_stream(iStream2, "%x", date1);
21     if (iStream2) std::cout << "date1: "  << date1 << '\n';
22     else std::cerr << "date1: Reading failed\n";
23
24     std::chrono::year_month_day date2;
25     std::istringstream iStream3{"11/15/21"};
26     std::chrono::from_stream(iStream3, "%x", date2);
27     if (iStream3) std::cout << "date2: "  << date2 << '\n';
28     else std::cerr << "date2: Reading failed\n";
29
30     std::cout << '\n';
31
32 }
```

On lines 13 and 14, the data on the input stream (`iStream1`) matches the format string (`"%F %T"`). The same holds for input stream `iStream2` (line 19) and the corresponding format string `"%x"` (line 20). On the contrary, there is no 15th month, and the parse step in line 26 fails. Consequentially, the failbit of the `iStream3` is set. Using the `iStream3` in a boolean expression evaluates to false.



Reading chrono objects from an input stream using `std::chrono::from_stream`

5.6.6.2.2 `std::chrono::parse`

Accordingly to `std::chrono::from_stream`, you can use the function `std::chrono::parse` for parsing input. The following code snippet shows their equivalence.

Equivalence of `std::chrono::from_stream` and `std::chrono::parse`

```
std::chrono::from_stream(is, fmt, chro)
is >> std::chrono::parse(fmt, chro)
```

Instead of `std::chrono::from_stream`, `std::chrono::parse` is directly invoked on the input stream `is`. `std::chrono::parse` also needs a format string `fmt` and a chrono object `chro`.

Consequently, I can directly rewrite the previous program `inputChrono.cpp` using `std::chrono::from_stream` into the program `inputChronoParse.cpp` using `std::chrono::parse`.

Reading chrono objects from an input stream using `std::chrono::parse`

```
1  // inputChronoParse.cpp
2
3  #include <chrono>
4  #include <iostream>
5  #include <string>
6  #include <sstream>
7
8  int main() {
9
10     std::cout << '\n';
11
12     std::chrono::sys_seconds timePoint;
13     std::istringstream iStream1{"2021-08-11 21:49:35"};
14     iStream1 >> std::chrono::parse("%F %T", timePoint);
15     if (iStream1) std::cout << "timePoint: "  << timePoint << '\n';
16     else std::cerr << "timepoint: Reading failed\n";
```

```
17
18      std::chrono::year_month_day date1;
19      std::istringstream iStream2{"11/08/21"};
20      iStream2 >> std::chrono::parse("%x", date1);
21      if (iStream2) std::cout << "date1: "  << date1 << '\n';
22      else std::cerr << "date1: Reading failed\n";
23
24      std::chrono::year_month_day date2;
25      std::istringstream iStream3{"11/15/21"};
26      iStream3 >> std::chrono::parse("%x", date2);
27      if (iStream3) std::cout << "date2: "  << date2 << '\n';
28      else std::cerr << "date2: Reading failed\n";
29
30      std::cout << '\n';
31
32  }
```

Format String must be a C++ String

The format string in `std::chrono::parse` must be a C++ string and cannot be a C string. This issue is already regarded as a oversight and will be fixed.

```
1   std::chrono::parse("%F %T", timePoint);
2
3   std::chrono::parse(std::string("%F %T"), timePoint);
4
5   using namespace std::literals;
6   std::chrono::parse("%F %T"s, timePoint);
```

A straightforward fix of this issue (line 1) is to use a `std::string` (line 3), or a string literal (line 6).

Distilled Information

- C++20 adds new components to the chrono library: time of day, calendar, and time zone.
- Time of day is the time duration since midnight, split into hours, minutes, seconds, and fractional seconds.
- Calendar stands for various calendar dates such as year, a month, a weekday, or the n-th day of a week.
- A time zone represents time specific to a geographic area.
- Thanks to the new formatting library, you can read and write chrono objects formatted to and from input streams.

5.7 Further Improvements

Cippi goes up

5.7.1 `std::bind_front`

`std::bind_front (Func&& func, Args&& ... args)` creates a callable wrapper for a callable `func`. `std::bind_front` can have an arbitrary number of arguments and binds its arguments to the front.

`std::bind_front` versus `std::bind`

Since C++11, we have had `std::bind`[67] and lambda expressions[68]. With C++20, we get `std::bind_front`[69]. This may make you wonder. To be pedantic `std::bind` is available since the Technical Report 1[70] (TR1). `std::bind` and lambda expressions can be used as a replacement of `std::bind_front`. Furthermore, `std::bind_front` seems like the little sister of `std::bind`, because only `std::bind` supports the rearranging of arguments. Of course, there is a reason to use `std::bind_front` in the future: in contrast to `std::bind`, `std::bind_front` propagates the exception specification of the underlying call operator.

The following program shows that you can replace `std::bind_front` with `std::bind` or lambda expressions.

[67] https://en.cppreference.com/w/cpp/utility/functional/bind
[68] https://en.cppreference.com/w/cpp/language/lambda
[69] https://en.cppreference.com/w/cpp/utility/functional/bind_front
[70] https://en.wikipedia.org/wiki/C%2B%2B_Technical_Report_1

Comparing `std::bind_front`, `std::bind`, and a lambda expression

```
1  // bindFront.cpp
2
3  #include <functional>
4  #include <iostream>
5
6  int plusFunction(int a, int b) {
7      return a + b;
8  }
9
10 auto plusLambda = [](int a, int b) {
11     return a + b;
12 };
13
14 int main() {
15
16     std::cout << '\n';
17
18     auto twoThousandPlus1 = std::bind_front(plusFunction, 2000);
19     std::cout << "twoThousandPlus1(20): " << twoThousandPlus1(20) << '\n';
20
21     auto twoThousandPlus2 = std::bind_front(plusLambda, 2000);
22     std::cout << "twoThousandPlus2(20): " << twoThousandPlus2(20) << '\n';
23
24     auto twoThousandPlus3 = std::bind_front(std::plus<int>(), 2000);
25     std::cout << "twoThousandPlus3(20): " << twoThousandPlus3(20) << '\n';
26
27     std::cout << "\n\n";
28
29     using namespace std::placeholders;
30
31     auto twoThousandPlus4 = std::bind(plusFunction, 2000, _1);
32     std::cout << "twoThousandPlus4(20): " << twoThousandPlus4(20) << '\n';
33
34     auto twoThousandPlus5 =  [](int b) { return plusLambda(2000, b); };
35     std::cout << "twoThousandPlus5(20): " << twoThousandPlus5(20) << '\n';
36
37     std::cout << '\n';
38
39 }
```

Each call (lines 18, 21, 24, 31, and 34) gets a callable taking two arguments and returns a callable taking only one argument because the first argument is bound to `2000`. The callable is a function (line

18), a lambda expression (line 21), and a predefined function object (line 24). Parameter `_1` is a so-called placeholder (line 31) and stands for the missing argument. With lambda expression (line 34), you can directly apply one argument and provide an argument `b` for the missing parameter. From the readability perspective, `std::bind_front` may be easier to read than `std::bind` or a lambda expression.

```
twoThousandPlus1(20): 2020
twoThousandPlus2(20): 2020
twoThousandPlus3(20): 2020


twoThousandPlus4(20): 2020
twoThousandPlus5(20): 2020
```

Applying `std::bind`, `std::bind_front`, and a lambda expression

5.7.2 `std::is_constant_evaluated`

The function `std::is_constant_evaluted` determines whether the function call occurs within a constant-evaluated context or not. Why do we need this function from the type-traits library? In C++20, we have roughly spoken three kinds of functions:

- `consteval` declared functions run at compile time: `consteval int alwaysCompiletime();`
- `constexpr` declared functions can run at compile time or run time: `constexpr int itDepends();`
- usual functions run at run time: `int alwaysRuntime();`

Now, I must write about the complicated case: `constexpr`. A `constexpr` function can run at compile time or run time. Sometimes these functions should behave differently, depending on whether the function call occurs within a constant-evaluated context or not. A constexpr function such as `getSum` has the potential to run at compile time.

A `constexpr`-declared function

```
constexpr int getSum(int l, int r) {
    return l + r;
}
```

A `constexpr` function can be called within a constant-evaluated context or not.

1. A constant-evaluated context
    - Implicit: A call inside a `constexpr` function or a `static_assert`.
    - Explicit: The client of the function explicitly wants to have the result at compile time: `constexpr auto res = getSum(2000, 11)`. Now, `getSum()` has to run at compile time.

2. A non-constant-evaluated context

    - A `constexpr` function can only be performed at run time if the arguments are not `constexpr`. This would be the case if the function `getSum(a, 11)` is invoked with a variable that was not declared as `constexpr : int a = 2000`.

You can detect if the function call occurs within a constant-evaluated context and perform different operations. cppreference.com/is_constant_evaluated[71] shows a smart use case. At compile time, you explicitly calculate the power of two numbers; at run time, you use `std::pow`.

Executing different code at compile time and run time

```
// constantEvaluated.cpp

#include <type_traits>
#include <cmath>
#include <iostream>

constexpr double power(double b, int x) {
    if (std::is_constant_evaluated() && !(b == 0.0 && x < 0)) {

        if (x == 0)
            return 1.0;
        double r = 1.0, p = x > 0 ? b : 1.0 / b;
        auto u = unsigned(x > 0 ? x : -x);
        while (u != 0) {
            if (u & 1) r *= p;
            u /= 2;
            p *= p;
        }
        return r;
    }
    else {
        return std::pow(b, double(x));
    }
}

int main() {

    std::cout << '\n';

    constexpr double kilo1 = power(10.0, 3);
    std::cout << "kilo1: " << kilo1 << '\n';

    int n = 3;
```

[71] https://en.cppreference.com/w/cpp/types/is_constant_evaluated

```
    double kilo2 = power(10.0, n);
    std::cout << "kilo2: " << kilo2 << '\n';

    std::cout << '\n';

}
```

There are two interesting observation I want to share.

- You can use a non-`constexpr` function such as `std::pow`[72] in the run-time branch of the function `constantEvaluated.cpp`.
- It is possible to use `std::is_constant_evaluated` in a `consteval` declared function or in a function that can only run at run time. In this case, the compile-time branch or run-time branch is performed.

5.7.3 `std::ssize`

Accordingly to `std::ranges::ssize`, `std::ssize` in C++20 returns a signed value. In contrast, the container's member function `size` returns an unsigned value.

`size`, `std::size`, and `std::ssize`

```
1 std::vector myVec{1, 2, 3};
2
3 for (int i = 0; i <= myVec.size(); i++) { }
4 for (int i = 0; i <= std::size(myVec); i++) { }
5 for (int i = 0; i <= std::ssize(myVec); i++) { }
```

The expressions `myVec.size()` (line 3) and `std::size(myVec)` (line 4) return an unsigned value, but `std::ssize(myVec)` (line 5) a signed value. Consequentially, the compiler may produce a warning for lines 3 and 4, depending on your compiler options. This warning is due to comparing the signed value of `i` and the unsigned value of the vector size. Furthermore, this comparison fails if the container's size exceeds the maximum value of the signed `int i`.

Thanks to argument-dependent lookup[73], you can use `std::size`, and `std::ssize` unqualified:

[72] https://en.cppreference.com/w/cpp/numeric/math/pow
[73] https://www.modernescpp.com/index.php/argument-dependent-lookup-and-hidden-friends/

Unqualified use of `std::size`, and `std::ssize`

```
std::vector myVec{1, 2, 3};

for (int i = 0; i <= size(myVec); i++) { }
for (int i = 0; i <= ssize(myVec); i++) { }
```

5.7.4 `std::source_location`

`std::source_location` represents implementation-defined information about the source code. This information includes the file name, line number, column number, and function name. The information is precious if you need information about the call site for debugging, logging, or testing purposes. The class `std::source_location` is a better alternative than the predefined C++11 macros `__FILE__` and `__LINE__` and should be used instead.

`std::source_location` can give you the following information.

`std::source_location src`

Function	Description
`std::source_location::current()`	Creates a new `source_location` object `src`
`src.line()`	Returns the line number
`src.column()`	Returns the column number
`src.file_name()`	Returns the file name
`src.function_name()`	Returns the function name

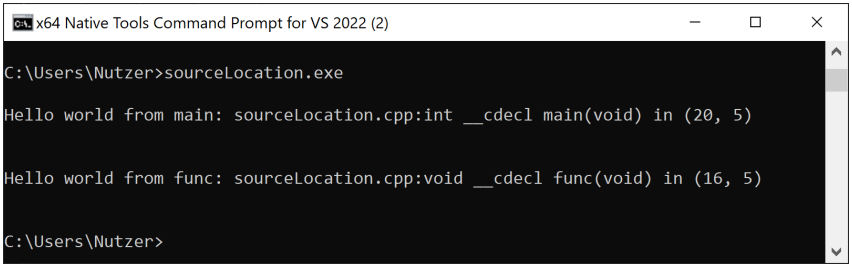
The `static consteval` function `std::source_location::current()` creates a new source location object `src` that represents the information of the call site. You can store this object in a container `std::vector<std::source_location>` or display its information.

Displaying information about the call site with `std::source_location`

```
1  // sourceLocation.cpp
2
3  #include <format>
4  #include <iostream>
5  #include <string_view>
6  #include <source_location>
7
8  void log(std::string_view message,
9           const std::source_location& logMessage = std::source_location::current()) {
10     std::cerr << std::format("\n{}: {}:{} in ({}, {}) \n\n",
11                   message,
12                   logMessage.file_name(), logMessage.function_name(),
13                   logMessage.line(), logMessage.column());
14 }
15
16 void func() {
17     log("Hello world from func");
18 }
19
20 int main() {
21     log("Hello world from main");
22     func();
23 }
```

The program `sourceLocation.cpp` displays for each log message (lines 17 and 21) the source file, the function name, and the line and column number.

First, here is the output of the MSVC compiler:



Displaying log information with the MSVC compiler

Additionally, here is GCC's output on the Compiler Explorer:

```
Hello world from main: /app/example.cpp:int main() in (21, 8)


Hello world from func: /app/example.cpp:void func() in (17, 8)
```

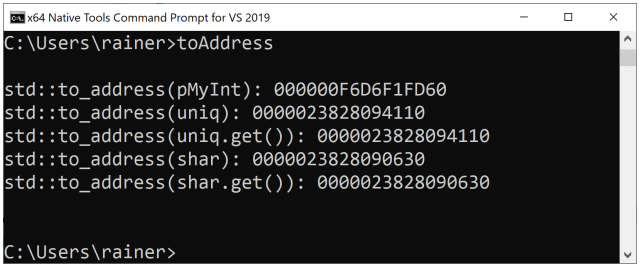
Displaying log information with the GCC compiler

5.7.5 `std::to_address`

`std::to_address(p)` returns the address of p without forming a reference to the object pointed to by `p`. The utility function `std::to_address` can handle raw pointers and fancy pointers (smart pointers) uniquely.

Displaying the address of raw pointers and smart pointers in an unique way

```
// toAddress.cpp

#include <iostream>
#include <memory>

int main() {

    std::cout << '\n';

    int myInt{5};

    int* pMyInt{&myInt};
    std::cout << "std::to_address(pMyInt): "  << std::to_address(pMyInt) << '\n';

    auto uniq = std::make_unique<int>(5);
    std::cout << "std::to_address(uniq): "  << std::to_address(uniq) << '\n';
    std::cout << "std::to_address(uniq.get()): "  << std::to_address(uniq.get()) << '\n';

    auto shar = std::make_shared<int>(5);
    std::cout << "std::to_address(shar): "  << std::to_address(shar) << '\n';
    std::cout << "std::to_address(shar.get()): "  << std::to_address(shar.get()) << '\n';

    std::cout << '\n';

}
```



Displaying the address of raw pointers and smart pointers in a unique way

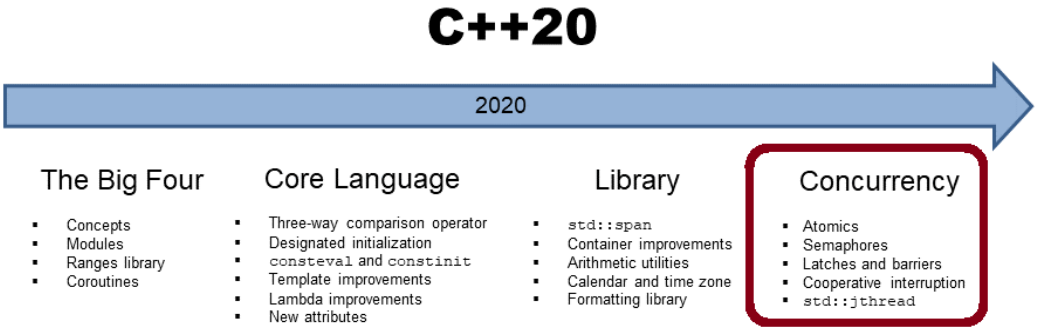
In contrast to `std::addressof(*p)`[74], `std::to_address(p)` can be used even when `p` does not reference storage that has an object constructed in it.

Distilled Information

- `std::bind_front` is the easier-to-use variant for `std::bind` (C++11). In contrast to `std::bind`, `std::bind_front` does not enable the rearranging of its arguments.
- The function `std::is_constant_evaluted` determines whether the function is executed at compile time or run time.
- `std::source_location` represents information about the source code. This information includes file names, line numbers, and function names, and is highly valuable for debugging, logging, or testing.
- `std::to_address(p)` returns the address of p without forming a reference to the object pointed to by `p`.

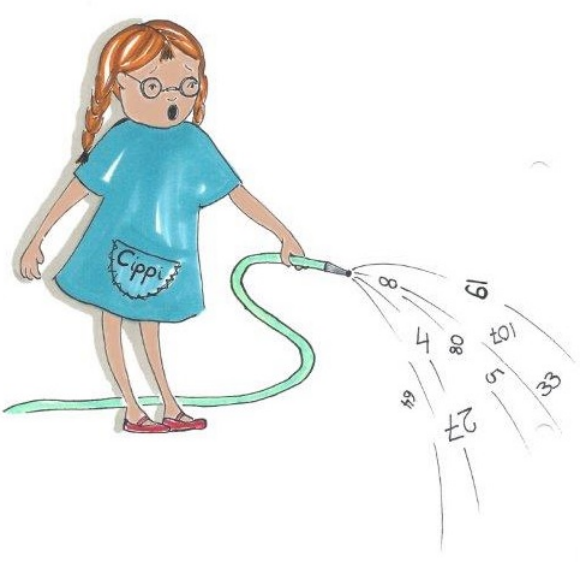
[74] https://en.cppreference.com/w/cpp/memory/addressof

6. Concurrency



With the publishing of the C++11 standard, C++ got a multithreading library and a memory model. This library has basic building blocks like atomic variables, threads, locks, and condition variables. That's the foundation on which C++ standards such as C++20 can establish higher-level abstractions.

6.1 Coroutines



Cippi waters the flowers

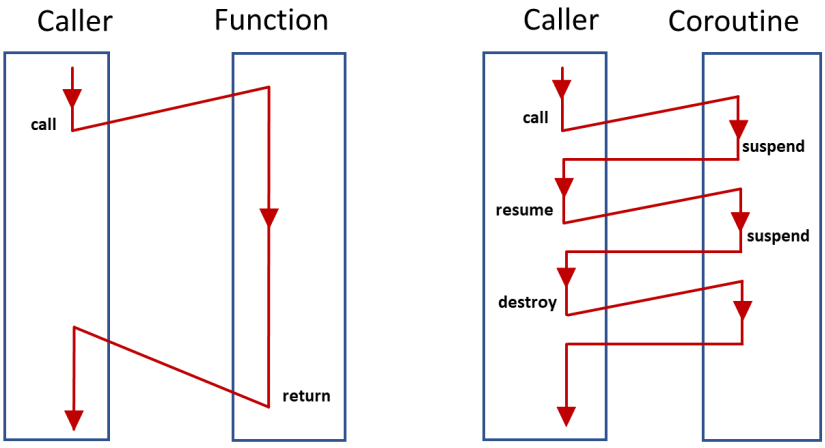
Coroutines are functions that can suspend and resume their execution while keeping their state. The evolution of functions in C++ goes one step further.

The Challenge of Understanding Coroutines

It was quite a challenge for me to understand coroutines. I strongly suggest that you should not read the sections in the chapter in sequence. Skip in your first iteration the sections The Framework, Awaitable and Awaiters, and The Workflow. Furthermore, read the case studies Variations of Futures, Modification and Generalization of a Generator, and Various Job Workflows. Reading, studying, and playing with the examples provided should give you the initial intuition needed to dive into details and the workflow of coroutines.

What I present in this section as a new idea in C++20 is quite old. The term coroutine was coined by Melvin Conway[1]. He used it in his publication on compiler construction in 1963. Donald Knuth[2] called procedures a special case of coroutines. Sometimes, it just takes a while to get your ideas accepted.

[1] https://en.wikipedia.org/wiki/Melvin_Conway
[2] https://en.wikipedia.org/wiki/Donald_Knuth



Functions versus Coroutines

While you can only call a function and return from it, you can call a coroutine, suspend and resume it, and destroy a suspended coroutine.

With the new keywords `co_await` and `co_yield`, C++20 extends the execution of C++ functions with two new concepts.

Thanks to the `co_await expression`, it is possible to suspend and resume the execution of the `expression`. If you use `co_await expression` in a function `func`, the call `auto getResult = func()` does not block if the result of the function is not available. Instead of resource-consuming blocking, you have resource-friendly waiting.

`co_yield expression` supports generator functions. The generator function returns a new value each time you call it. A generator function is a kind of data stream from which you can extract values. The data stream can be infinite. Therefore, we are at the center of lazy evaluation with C++.

6.1.1 A Generator Function

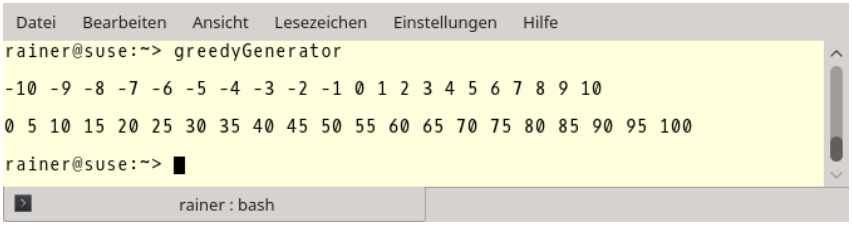
The following program is as simple as possible. The function `getNumbers` returns all integers from `begin` to `end`, incremented by `inc`. Value `begin` has to be smaller than `end`, and `inc` has to be positive.

A greedy generator function

```
// greedyGenerator.cpp

#include <iostream>
#include <vector>

std::vector<int> getNumbers(int begin, int end, int inc = 1) {

    std::vector<int> numbers;
    for (int i = begin; i < end; i += inc) {
        numbers.push_back(i);
    }

    return numbers;

}

int main() {

    std::cout << '\n';

    const auto numbers= getNumbers(-10, 11);

    for (auto n: numbers) std::cout << n << " ";

    std::cout << "\n\n";

    for (auto n: getNumbers(0, 101, 5)) std::cout << n << " ";

    std::cout << "\n\n";

}
```

Of course, I'm reinventing the wheel with `getNumbers`, because this task could be done with `std::iota`[3]. For completeness, here is the output.

[3] http://en.cppreference.com/w/cpp/algorithm/iota



A generator function

Two observations of the program `greedyGenerator.cpp` are essential. On the one hand, the vector `numbers` in line 8 always gets all values. This holds even if I'm only interested in the first 5 elements of a vector with 1000 elements. On the other hand, it's easy to transform the function `getNumbers` into a lazy generator. The following program is intentionally not complete. The definition of the `generator` is still missing.

A lazy generator function

```
1  // lazyGenerator.cpp
2
3  #include <iostream>
4
5  generator<int> generatorForNumbers(int begin, int inc = 1) {
6
7      for (int i = begin;; i += inc) {
8          co_yield i;
9      }
10
11 }
12
13 int main() {
14
15     std::cout << '\n';
16
17     const auto numbers = generatorForNumbers(-10);
18
19     for (int i= 1; i <= 20; ++i) std::cout << numbers() << " ";
20
21     std::cout << "\n\n";
22
23     for (auto n: generatorForNumbers(0, 5)) std::cout << n << " ";
24
25     std::cout << "\n\n";
26
27 }
```

While the function `getNumbers` in the file `greedyGenerator.cpp` returns a `std::vector<int>`, the

coroutine `generatorForNumbers` in `lazyGenerator.cpp` returns a generator. The generator `numbers` in line 17 or `generatorForNumbers(0, 5)` in line 23 returns a new number on request. The range-based for loop triggers the query. Precisely, the query of the coroutine returns the value `i` via `co_yield i` and immediately suspends its execution. If a new value is requested, the coroutine resumes execution exactly at that place.

The expression `generatorForNumbers(0, 5)` in line 23 is a just-in-place use of a generator.

I want to stress one point explicitly. The coroutine `generatorForNumbers` creates an infinite data stream because the for loop in line 8 has no end condition. This is fine if I only ask for a finite number of values, such as in line 20. This does not hold for line 23 since there is no end condition. Therefore, the expression runs *forever*.

6.1.2 Characteristics

Coroutines have a few unique characteristics.

6.1.2.1 Typical Use Cases

Coroutines are the usual way to write event-driven applications[4], which can be simulations, games, servers, user interfaces, or even algorithms. Coroutines are also typically used for cooperative multitasking[5]. The key to cooperative multitasking is that each task takes as much time as it needs but avoids sleeping or waiting and instead allows some other task to run. Cooperative multitasking stands in contrast to pre-emptive multitasking, for which we have a scheduler that decides how long each task gets the CPU.

There are different kinds of coroutines.

6.1.2.2 Underlying Concepts

Coroutines in C++20 are asymmetric, first-class, and stackless.

The workflow of an **asymmetric** coroutine goes back to the caller. This does not hold for a symmetric coroutine. A symmetric coroutine can delegate its workflow to another coroutine.

First-class coroutines are similar to first-class functions, since coroutines behave like data. Behaving like data means that you can use them as arguments to or return values from functions, or store them in a variable.

A **stackless** coroutine can suspend and resume the top-level coroutine. The execution of the coroutine and the yielding from the coroutine comes back to the caller. The coroutine stores its state for resumption separate from the stack. Stackless coroutines are often called resumable functions.

[4] https://en.wikipedia.org/wiki/Event-driven_programming
[5] https://en.wikipedia.org/wiki/Computer_multitasking

6.1.2.3 Design Goals

Gor Nishanov describes in proposal N4402[6] the design goals of coroutines.

Coroutines should

- be highly scalable (to billions of concurrent coroutines)
- have highly efficient resume and suspend operations comparable in cost to the overhead of a function
- seamlessly interact with existing facilities with no overhead
- have open-ended coroutine machinery allowing library designers to develop coroutine libraries exposing various high-level semantics such as generators, goroutines[7], tasks and more
- usable in environments where exceptions are forbidden or not available

Due to the design goals of scalability and seamless interaction with existing facilities, the coroutines are stackless. In contrast, a stackful coroutine reserves a default stack of 1MB on Windows and 2MB on Linux.

There are four ways for a function to become a coroutine.

6.1.2.4 Becoming a Coroutine

A function becomes a coroutine if it uses

- `co_return`, or
- `co_await`, or
- `co_yield`, or a
- `co_await` expression in a range-based for loop.

[6] https://isocpp.org/files/papers/N4402.pdf
[7] https://tour.golang.org/concurrency/1

Distinguish Between the Coroutine and the Coroutine Handle

The term coroutine is often used for two different things: the function invoking `co_return`, `co_await`, or `co_yield`, and the coroutine handle. Using one term for two different entities may puzzle you (such as it did me). Let me clarify both terms.

A simple coroutine producing 2021

```
MyFuture<int> createFuture() {
   co_return 2021;
}

int main() {

   auto fut = createFuture();
   std::cout << "fut.get(): " << fut.get() << '\n';

}
```

This straightforward example has a function `createFuture` and returns an object of type `MyFuture<int>`. Both are called coroutines. To be specific, the function `createFuture` is the coroutine that returns a coroutine handle `MyFuture<int>`. The coroutine handle is a handle to a objects that implements the framework to model a specific behavior. I present in the section `co_return` the implementation and the use of this straightforward coroutine.

6.1.2.4.1 Restrictions

Coroutines cannot have `return` statements or placeholder return types. This applies for unconstrained placeholders (`auto`) and constrained placeholders (concepts).

Additionally, functions having variadic arguments[8], `constexpr` functions, `consteval` functions, constructors, destructors, and the `main` function cannot be coroutines.

6.1.3 The Framework

The framework for implementing coroutines consists of more than 20 functions, some of which you must implement and some of which you may overwrite. Therefore, you can tailor the coroutine to your needs.

A coroutine is associated with three parts: the promise object, the coroutine handle, and the coroutine frame. The client gets the coroutine handle to interact with the promise object, which keeps its state in the coroutine frame.

[8] https://en.cppreference.com/w/cpp/language/variadic_arguments

6.1.3.1 Promise Object

The promise object is manipulated inside the coroutine, and it delivers its result or exception via the promise object.

The promise object supports the following interface. Some of the functions must be implemented, some of them are optional.

Promise object

Member Function	Description
Constructor	A promise needs a constructor.
`initial_suspend()`	Determines if the coroutine suspends before it runs.
`final_suspend noexcept()`	Determines if the coroutine suspends before it ends.
`unhandled_exception()`	Called when an unhandled exception happens.
`get_return_object()`	Initializes the coroutine handle that is returned to the caller.
`get_return_object_on_allocation_failure()`	Defines if memory allocation fails.
`return_value(val)`	Is invoked by `co_return val`.
`return_void()`	Is invoked by `co_return` or the end of the coroutine.
`yield_value(val)`	Is invoked by `co_yield val`.
`await_transform(val)`	Returns an Awaitable.
`operator new(size)`	Defines how the coroutine allocates memory.
`operator delete(ptr, size)`	Defines how the coroutine frees memory.

The compiler automatically invokes these functions during its execution of the coroutine. The section workflow presents this workflow in detail.

The function `get_return_object` initializes the coroutine handle that the client uses to interact with the coroutine.

The three functions `yield_value`, `initial_suspend` and `final_suspend` return an awaiter. Often, the predefined Awaiters `std::suspend_always` and `std::suspend_never` are used. This Awaiter can start eager or lazy. The function `final_suspend` can execute some logic if the coroutine is finally suspended.

A promise needs at least one of the member functions `return_value`, `return_void`, or `yield_value`. Additionally, either `return_value` or `return_void` must be available. When you overload `yield_value` or `return_value` of the promise object for different types, the coroutine can return different values using `co_yield`, or `co_return`.

The `static` function `get_return_object_on_allocation_failure` guarantees that memory allocation of the coroutine never throws. When this `static` member function is implemented, the overloaded `::operator new(std::size_t sz, std::nothrow_t)` is called. Consequentially, when you implement your `operator new`, it must be `noexcept` and return a `nullptr` when memory allocation fails.

Thanks to the two functions `operator new(size)`, and `operator delete(ptr, size)`, you can implement a coroutine-specific memory allocation strategy. This strategy may avoid memory allocation on the heap.

For an unhandled exception in the coroutine, the function `unhandled_exception` is called. Now, you can handle your exception in various ways: you can ignore the exception, terminate your program using, for example `std::exit`[9], handle the exception, or store it using `std::current_exception`[10]. You have to rethrow the exception in the `try` block to handle it:

Handling an exception in the coroutine

```
void unhandled_exception() {
    try {
        throw; // rethrow the exception
    }
    catch (const std::exception& excep) {
        std::cerr << "Exception in the coroutine: " << excep.what() << '\n';
    }
}
```

6.1.3.2 Coroutine Handle

The coroutine handle is a non-owning handle to resume or destroy the coroutine frame from the outside. The coroutine handle is part of the resumable function. The following table shows the coroutine handles interface.

[9] https://en.cppreference.com/w/cpp/utility/program/exit
[10] https://en.cppreference.com/w/cpp/error/current_exception

Functions	Description
Constructor{} Constructor{`nullptr`}	Creates a handle not referring to a coroutine
Constructor{`handle`}	Copies the `handle`.
`handle1 = handle2`	Assigns the `handle2`.
`coroutine_handle<PromType>::from_promise(prom)`	Creates a handle with the promise `prom`.
`coroutine_handle<PromType>::from_address(addr)`	Returns the handle for the address `addr`.
`operator bool`	Checks if the handle represents a coroutine.
`operator coroutine_handle<>`	Creates a type-erased coroutine handle.
`handle.done()`	Checks if the coroutine has completed.
`==`, `!=`	Checks if two handles refer the same coroutine.
`<`, `<=`, `>`, `>=`, `<=>`	Enables the ordering of handles.
`handle.resume()`	Resumes the coroutine.
`handle()`	Resumes the coroutine.
`handle.destroy()`	Destroys the coroutine.
`handle.promise()`	Returns the promise of the coroutine.
`handle.address()`	Returns the underlying address of the coroutine data.

Typically, the promise invokes in its member function `get_return_object` the static member function `coroutine_handle<PromType>::from_promise(prom)` to initialize the coroutine handle.

Creating the coroutine handle from the promise

```
struct promise_type {
    ...
    auto get_return_object() {
        return Generator{handle_type::from_promise(*this)};
    }
    ...
};
```

This call `get_return_object` creates and returns a `Generator`, initialized with the promise. Finally, the class `Generator` uses the handle.

A coroutine handle

```
1  template<typename T>
2  struct Generator {
3
4      struct promise_type;
5      using handle_type = std::coroutine_handle<promise_type>;
6
7      Generator(handle_type h): coro(h) {}
8      handle_type coro;
9
10     ~Generator() {
11         if ( coro ) coro.destroy();
12     }
13     T getValue() {
14         return coro.promise().current_value;
15     }
16     bool next() {
17         coro.resume();
18         return not coro.done();
19     }
20 ...
21 }
```

The constructor (line 7) gets the coroutine handle to the promise that has type `std::coroutine_handle<promise_type>`[11]. The member functions `next` (line 16) and `getValue` (line 13) enables a client to resume the promise (`gen.next()`) or ask for its value (`gen.getValue()`) using the coroutine handle.

Internally, both functions trigger the coroutine handle `coro` (line 8) to

- resume the coroutine: `coro.resume()` (line 17) or `coro()`;

[11] https://en.cppreference.com/w/cpp/coroutine/coroutine_handle

- destroy the coroutine: `coro.destroy()` (line 11);
- check the state of the coroutine: `coro` (line 11).

The coroutine is automatically destroyed when its function body ends. Its call `coro` returns `true` at its final suspension point.

A default initialized or with a `nullptr` initialized coroutine handle does not refer to a coroutine. Using this handle in a logical expression, return false. Coroutine handles are typically copied because copying or assigning them is cheap.

The function `address` returns a void pointer (`void*`) to the coroutine. This void pointer can be used to create the handle using the static function `from_address`:

Creating the coroutine handle from the coroutine address

```
auto handle = std::coroutine_handle<Promise>::from_promise(prm);
void* handlePointer = handle.address();
auto handle2 = std::coroutine_handle<Promise>::from_address(handlePointer);
```

Thanks to the overloaded `coroutine_handle<>` operator, you can convert a coroutine handle to `std::coroutine_handle<void>`. The type-erased coroutine handle can be used to accept any coroutine handle but misses the promise.

Creating the coroutine handle from the coroutine address

```
auto handle = std::coroutine_handle<Promise>::from_promise(prm);
std::coroutine_handle<> handle2 = handle;
handle2.resume();
```

Calling `promise` on the type-erased coroutine handle `handle2` is an error: `handle2.promise()`.

Typically, the coroutine returns the coroutine handle.

6.1.3.2.1 `std::coroutine_traits`

`coroutine_traits` allows it to inject the coroutine handle into a coroutine.

Injecting the coroutine handle

```cpp
// coroutineTraits.cpp

#include <coroutine>
#include <memory>
#include <iostream>

struct GeneratorVerbose {

    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;

    handle_type coro;
    GeneratorVerbose() {
        std::cout << "        GeneratorVerbose::GeneratorVerbose" << '\n';
    }

    ~GeneratorVerbose() {
        std::cout << "        GeneratorVerbose::~GeneratorVerbose" << '\n';
        if ( coro ) coro.destroy();
    }

    int getNextValue() {
        std::cout << "        GeneratorVerbose::getNextValue" << '\n';
        coro.resume();
        return coro.promise().current_value;
    }
    struct promise_type {
        promise_type(int, GeneratorVerbose& genVerbose) {
            std::cout << "            promise_type::promise_type" << '\n';
            genVerbose.coro =  handle_type::from_promise(*this);
        }

        ~promise_type() {
            std::cout << "            promise_type::~promise_type" << '\n';
        }

        std::suspend_always initial_suspend() {
            std::cout << "            promise_type::initial_suspend" << '\n';
            return {};
        }
        std::suspend_always final_suspend() noexcept {
            std::cout << "            promise_type::final_suspend" << '\n';
            return {};
        }
```

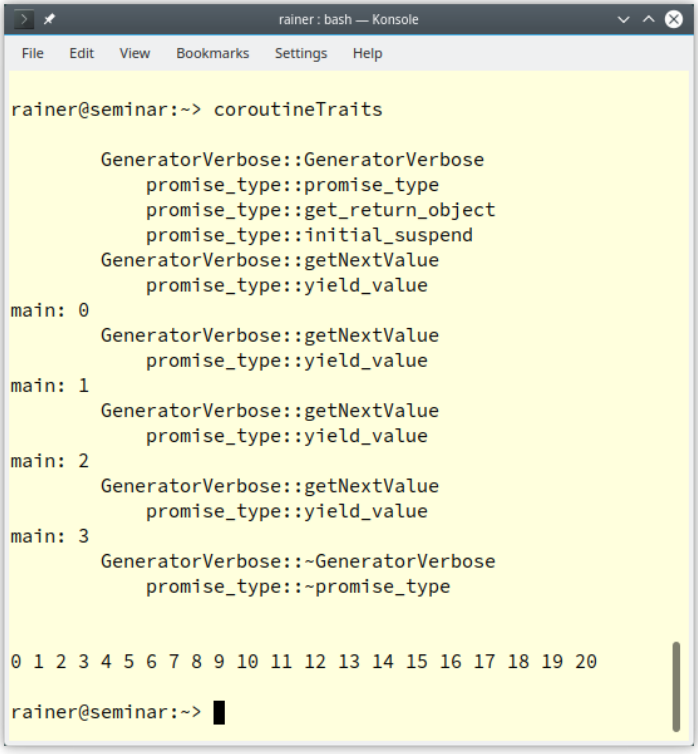
```
45          auto get_return_object() {
46              std::cout << "            promise_type::get_return_object" << '\n';          \
47
48          }
49
50          std::suspend_always yield_value(int value) {
51              std::cout << "            promise_type::yield_value" << '\n';                \
52
53              current_value = value;
54              return {};
55          }
56          void return_void() {}
57          void unhandled_exception() {
58              std::exit(1);
59          }
60
61          int current_value;
62      };
63
64  };
65
66  struct Generator {
67
68      struct promise_type;
69      using handle_type = std::coroutine_handle<promise_type>;
70
71      handle_type coro;
72
73      ~Generator() {
74          if ( coro ) coro.destroy();
75      }
76
77      int getNextValue() {
78          coro.resume();
79          return coro.promise().current_value;
80      }
81      struct promise_type {
82          promise_type(int, Generator& gen) {
83              gen.coro =  handle_type::from_promise(*this);
84          }
85
86          std::suspend_always initial_suspend() {
87              return {};
88          }
89          std::suspend_always final_suspend() noexcept {
```

```
90              return {};
91          }
92
93          auto get_return_object() { }
94
95          std::suspend_always yield_value(int value) {
96              current_value = value;
97              return {};
98          }
99          void return_void() {}
100         void unhandled_exception() {
101             std::exit(1);
102         }
103
104         int current_value;
105     };
106
107 };
108
109 template<>
110 struct std::coroutine_traits<void, int, GeneratorVerbose&> {
111     using promise_type = GeneratorVerbose::promise_type;
112 };
113
114 template<>
115 struct std::coroutine_traits<void, int, Generator&> {
116     using promise_type = Generator::promise_type;
117 };
118
119
120 template <typename CoroutineInterface>
121 void getNext(int start, CoroutineInterface&) {
122     auto value = start;
123     while (true) {
124         co_yield value;
125         value += 1;
126     }
127 }
128
129 int main() {
130
131     std::cout << '\n';
132
133     {
134         GeneratorVerbose genVerbose;
```

```
135         getNext(0, genVerbose);
136         for (int i = 0; i <= 3; ++i) {
137             auto val = genVerbose.getNextValue();
138             std::cout << "main: " << val << '\n';
139         }
140     }
141
142     std::cout << "\n\n";
143
144     Generator gen;
145     getNext(0, gen);
146     for (int i = 0; i <= 20; ++i) {
147         auto val = gen.getNextValue();
148         std::cout << val << ' ';
149     }
150
151     std::cout << "\n\n";
152
153 }
```

The program `coroutineTraits.cpp` has two generators. Both are default constructed (lines 132 and 142). `GeneratorVerbose` (line 7) and `Generator` (line 64) have minimal functionality to be used inside a coroutine. For simplicity, I will call them coroutine interface. `GeneratorVerbose` displays when each function is called. The function template `getNext` is the coroutine. This coroutine doesn't return the coroutine, but it is injected. In line 133, I inject `GeneratorVerbose`, and in line 143 `Generator`. Thanks to the fully specialized coroutine traits (line 107) and (line 112), the compiler knows which promise type it should use. `void, int, GeneratorVerbose&` is the signature of the first instantiation of the function template `getNext` and `void, int, Generator&` the signature of the second. The first argument, `void`, stands for the return type of `getNext`. The `promise_type` of `GeneratorVerbose` and `Generator` need a constructor that takes exactly the same arguments as the function `getNext`. Usually, the member function `get_return_object` initializes and returns the coroutine handle. This step is not necessary when the coroutine interface is injected.

The following program shows the output of the program. In the first case, you can study the various function invocation.



A special Awaitable

6.1.3.3 Coroutine Frame

The coroutine frame is an internal, typically heap-allocated state. It consists of the already mentioned promise object, the coroutine's copied parameters, the representation of the suspension points, local variables whose lifetime ends before the current suspension point, and local variables whose lifetime exceeds the current suspension point.

The coroutine is typically heap-allocated, but compilers may avoid heap allocation. The following properties increase the likelihood that the coroutine is not heap allocated.

1. The lifetime of the coroutine has to be nested inside the lifetime of the caller.
2. Inline function give the compiler more insight to see the size of the frame.
3. `std::final_suspend` returns `std::suspend_always`. This simplifies lifetime management.

The crucial abstractions in the coroutine framework are Awaitables and Awaiters.

6.1.4 Awaitables and Awaiters

The three functions of a promise object `prom` `yield_value`, `initial_suspend`, and `final_suspend` return Awaitables.

6.1.4.1 Awaitables

An Awaitable is something you can await on. It is the argument of `co_await`: `co_await Awaitable`. The Awaitable determines if the coroutine pauses or not.

Essentially, the compiler generates the following function calls using the promise `prom` and the `co_await` operator.

Compiler-generated function calls

Call	Compiler generated call
Start coroutine execution	`co_await prom.initial_suspend()`
`co_yield value`	`co_await prom.yield_value(value)`
`co_return value`	`co_await prom.return_value(value)`
End coroutine execution	`co_await prom.final_suspend()`

Thanks to the member function `await_transform`, the promise can create the Awaitable.

6.1.4.1.1 `await_transform`

The following Awaitable suspends never. This example also shows how the Awaitable can get an argument.

A special Awaitable

```
1  // suspendsNeverWithSleep.cpp
2
3  #include <coroutine>
4  #include <chrono>
5  #include <format>
6  #include <iostream>
7  #include <thread>
8
9  struct MySuspendNever {
10   MySuspendNever(std::chrono::duration<double, std::milli> sleep): sleepDuration(sleep) {}
11   std::chrono::duration<double, std::milli> sleepDuration;
12
```

```
13    bool await_ready() const noexcept {
14      std::cout << "        MySuspendNever::await_ready"  << '\n';
15      std::this_thread::sleep_for(sleepDuration);
16      return true;
17    }
18    void await_suspend(std::coroutine_handle<>) const noexcept {
19      std::cout << "        MySuspendNever::await_suspend"  << '\n';
20    }
21    void await_resume() const noexcept {
22      std::cout << "        MySuspendNever::await_resume"  << '\n';
23    }
24  };
25
26  struct Job {
27    struct promise_type;
28    using handle_type = std::coroutine_handle<promise_type>;
29    handle_type coro;
30    Job(handle_type h): coro(h){}
31    ~Job() {
32      if ( coro ) coro.destroy();
33    }
34    void start() {
35      coro.resume();
36    }
37
38
39    struct promise_type {
40      auto get_return_object() {
41        return Job{handle_type::from_promise(*this)};
42      }
43      std::suspend_always initial_suspend() {
44        std::cout << "    Job prepared" << '\n';
45        return {};
46      }
47      std::suspend_always final_suspend() noexcept {
48        std::cout << "    Job finished" << '\n';
49        return {};
50      }
51      void return_void() {}
52      void unhandled_exception() {}
53
54    };
55  };
56
57  Job prepareJob() {
```

```
58    using namespace std::chrono_literals;
59    co_await MySuspendNever(0.5ms);
60  }
61
62  int main() {
63
64    std::cout <<  "Before job" << '\n';
65
66    auto start = std::chrono::steady_clock::now();
67    auto job = prepareJob();
68    job.start();
69    auto end = std::chrono::steady_clock::now();
70    std::cout << std::format("The job took {}\n", end - start);
71
72    std::cout <<  "After job" <<  '\n';
73
74  }
```

The Awaitable `MySuspendNever` (line 9) suspends never and sleeps for the `sleepDuration` (line 14). Its constructor takes the `sleepDuration` (line 10). Line 59 uses this special constructor taking the `sleepDuration` as argument. The following screenshot shows the output of the program.

```
Before job
    Job prepared
        MySuspendNever::await_ready
        MySuspendNever::await_resume
    Job finished
The job took 572975ns
After job
```

A special Awaitable

Thanks to `await_transform`, there is another way to get an Awaitable. First, the `co_wait` call of the coroutine `prepareJob` gets the time duration.

co_await has a time duration

```
// suspendsNeverWithSleepAwaitTransform.cpp

...

Job prepareJob() {
    using namespace std::chrono_literals;
    co_await 0.5ms;
}
```

Second, the promise type supports a member function `await_transform` that gets the time duration and returns the Awaitable.

The promise with a member function `await_transform`

```
// suspendsNeverWithSleepAwaitTransform.cpp

...

struct promise_type {
    ...
    auto await_transform(std::chrono::duration<double, std::milli> sleepDuration) {
        return MySuspendNever(sleepDuration);
    }
    ...
};
```

The `co_await` operator needs an Awaitable as an argument. Typically, the Awaitable becomes the Awaiter.

6.1.4.2 Awaiter

The concept Awaiter requires three member functions `await_ready`, `await_suspend`, and `await_resume`.

The Awaiter

Member Function	Description
Constructor	Initializes the Awaiter. Can take arguments.
`await_ready`	Indicates if the coroutine is ready for suspension.
`await_suspend(coroutineHandle)`	Handles the suspension of the coroutine.
`await_resume`	Handles the resumption of the coroutine.

The three member functions `await_ready`, `await_suspend`, and `await_resume` are typically `const`, `noexcept`, and `constexpr`.

6.1.4.2.1 `await_ready`

The function `await_ready` returns a boolean and is immediately called before the coroutine is suspended. The coroutine is suspended if `await_ready` returns `true`; otherwise, it is not suspended and continues its control flow. Typically, `await_ready` returns `false`, but it may return `true` if the reason for suspension no longer exists.

6.1.4.2.2 `await_suspend`

The function `await_suspend(coroutineHandle)` handles the suspension of the coroutine. `await_suspend` is the crucial function of the Awaiter. It supports various control flows based on its parameters and return types:

Parameter

- `std::coroutine_handle<PromiseType>`: the coroutine handle
- `std::coroutine_handle<>`: a type-erased coroutine handle that doesn't have access to the promise
- `auto`: automatically deduced return type

Return type

- `void`: the control flow immediately returns to the caller. The coroutine remains suspended.
- `bool`:
  - `true`: the control flow immediately returns to the caller
  - `false`: the coroutine is resumed
- `std::coroutine_handle<>`:
  - if it returns a coroutine handle `coroHandle` of another coroutine, this coroutine is resumed: `coroHandle.resume()`. This strategy is called `symmetric transfer`.
  - `std::noop_coroutine`: no coroutine is resumed. Equivalent to returning `true`.

6.1.4.2.3 `await_resume`

Typically, after the call `await_suspend`,, `await_resume` is called. `await_resume` return value is the result of the `co_wait awaitable` expression. The return value can also be `void`.

6.1.4.2.4 Symmetric Transfer

The symmetric transfer is if the call `std::await_suspend` returns a coroutine handle. This means that the coroutine is suspended, but the returned coroutine immediately resumed. Thanks to this technique, the immediately resumed coroutine uses the stack from the given coroutine.

The following code snippet shows the critical ideas of coroutine and an Awaiter implementing the continuation of coroutines.

Continuation with symmetric transfer

```
struct Task {
    struct promise_type;
    using handleType = std::coroutine_handle<promise_type>;
    handle_type origHandle;

    struct promise_type {
        std::coroutine_handle<> continuationHandle = contHandle;
        ...
        auto final_suspend() noexcept {
            return ContinuationAwaiter{};
        }
    };
...
};

struct ContinuationAwaiter {
    bool await_ready() noexcept {
        return false;
    }

    std::coroutine_handle<> await_suspend(Task::handleType handle) noexcept {
        if (handle.promise().continuationHandle) {
            return handle.promise().continuationHandle;
        }
        else {
            return std::noop_coroutine();
        }
    }

    void await_resume() noexcept { }
};
```

If a coroutine should continue with another coroutine, it must return a special Awaiter in its `final_suspend` call (line 9). Additionally, the `promise_type` must store the handle to the coroutine to be continued (line 7). If there is no continuation, you should set the continuation handle to a `nullptr`: `std::coroutine_handle<> continuationHandle = contHandle`.

The special Awaiter `ContinuationAwaiter` continues in the member function `await_suspend`. It gets the coroutine handle `handle`, checks if the continuation handle is set (line 22), and returns the continuation handle. The coroutine is suspended in this case, but the returned coroutine is immediately resumed. If the coroutine handle is not set, `std::noop_coroutine` is returned. This means the control flow immediately returns to the caller without resuming a coroutine. A `std::noop_coroutine` call creates a coroutine handle `std::noop_coroutine_handle`. Calling `resume`, `destroy`, `address`, or `done` on a `std::noop_coroutine_handle` has no effect.

6.1.4.3 `std::suspend_always` and `std::suspend_never`

The C++20 standard already defines two basic Awaiters: `std::suspend_always`, and `std::suspend_never`.

As its name suggests, the Awaiter `suspend_always` always suspends. Therefore, the call `await_ready` returns `false`.

The Awaiter `std::suspend_always`

```
struct suspend_always {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(std::coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

The opposite holds for `suspend_never`. It never suspends and, hence, the call `await_ready` returns `true`.

The Awaiter `std::suspend_never`

```
struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(std::coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

The Awaiters `std::suspend_always` and `std::suspend_never` are the basic building blocks for functions, such as `initial_suspend` and `final_suspend`. Both functions are automatically executed when the coroutine is executed: `initial_suspend` at the beginning and `final_suspend` at the end end of the coroutine.

6.1.4.4 Get the Awaitable and the Awaiter

There are essentially two ways to get an Awaiter.

- A `co_await` operator is defined.
- The Awaitable becomes the Awaiter.

Remember, when `co_await expression` is invoked, the `expression` is converted to an Awaitable:

1. if `expression` is created by an initial suspension point (`prom.initial_suspend()`), final suspension point (`prom.final_suspend()`), or yield expression (`prom.yield_value(value)`), the Awaitable is the `expression`.
2. if the current coroutine's promise type has a member function `await_transform`, the Awaitable is `prom.await_transform(expression)`.
3. the Awaitable is the `expression`.

Now, the compiler performs the following lookup rule to get an Awaiter:

1. It looks for the `co_await` operator on the promise object and returns an Awaiter:

   ```
   awaiter = awaitable.operator co_await();
   ```

2. It looks for a freestanding `co_await` operator and returns an Awaiter:

   ```
   awaiter = operator co_await(awaitable);
   ```

3. If there is no `co_await` operator defined, the Awaitable becomes the Awaiter:

   ```
   awaiter = awaitable;
   ```

`awaiter = awaitable`

When you study my coroutine implementations in this chapter, you may notice that I use most of the time that an Awaitable implicitly becomes an Awaiter. Only the example to thread synchronization uses the `co_await` operator to get the Awaiter.

After these static aspects of coroutines, I want to continue with their dynamic aspects.

6.1.5 The Workflows

The compiler transforms your coroutine and runs two workflows: the outer promise workflow and the inner Awaiter workflow.

6.1.5.1 The Promise Workflow

When you use `co_yield`, `co_await`, or `co_return` in a function, the function becomes a coroutine, and the compiler transforms its body to something equivalent to the following lines.

The transformed coroutine

```
1  {
2      Promise prom;
3      co_await prom.initial_suspend();
4      try {
5          <function body having co_return, co_yield, or co_await>
6      }
7      catch (...) {
8          prom.unhandled_exception();
9      }
10 FinalSuspend:
11     co_await prom.final_suspend();
12 }
```

The compiler automatically runs the transformed code using the functions of the promise object. In short, I call this workflow the promise workflow. Here are the main steps of this workflow.

- Coroutine begins execution
  - allocates the coroutine frame if necessary
  - copies all function parameters to the coroutine frame
  - creates the prom object `prom` (line 2)
  - calls `prom.get_return_object()` to create the coroutine handle and keeps it in a local variable. The result of the call will be returned to the caller when the coroutine first suspends.
  - calls `prom.initial_suspend()` and `co_await`s its result. The promise type typically returns `suspend_never` for eagerly-started coroutines or `suspend_always` for lazily-started coroutines. (line 3)
  - the body of the coroutine is executed when `co_await prom.initial_suspend()` resumes
- Coroutine reaches a suspension point
  - the return object (`prom.get_return_object()`) is returned to the caller which resumed the coroutine
- Coroutine reaches `co_return`
  - calls `prom.return_void()` for `co_return` or `co_return expression`, where `expression` has type `void`
  - calls `prom.return_value(expression)` for `co_return expression`, where `expression` has non-`void` type.
  - destroys all stack-created variables
  - calls `prom.final_suspend()` and `co_await`s its result
- Coroutine is destroyed (by terminating via `co_return` or via uncaught exception, or via the coroutine handle)

- calls the destructor of the promise object
- calls the destructor of the function parameters
- frees the memory used by the coroutine frame
- transfers control back to the caller

When a coroutine ends with an uncaught exception, the following happens:

- catches the exception and calls `prom.unhandled_exception()` from the catch block
- calls `prom.final_suspend()` and `co_await`s the result (line 11)

When you use `co_await expr` in a coroutine, or the compiler implicitly invokes `co_await prom.initial_suspend()`, `co_await prom.final.suspend()`, or `co_await prom.yield_value(value)`, a second, inner Awaitable workflow starts.

6.1.5.2 The Awaiter Workflow

Using `co_await expr` causes the compiler to transform the code based on the functions `await_ready`, `await_suspend`, and `await_resume`. Consequently, I call the execution of the transformed code the Awaiter workflow.

The compiler generates approximately the following code using the `awaiter`. For simplicity, I ignore exception handling and describe the workflow with comments.

The generated Awaiter Workflow

```
1  awaiter.await_ready() returns false:
2
3      suspend coroutine
4
5      awaiter.await_suspend(coroutineHandle) returns:
6
7          void:
8              awaiter.await_suspend(coroutineHandle);
9              coroutine keeps suspended
10             return to caller
11
12         bool:
13             bool result = awaiter.await_suspend(coroutineHandle);
14             if result:
15                 coroutine keep suspended
16                 return to caller
17             else:
18                 go to resumptionPoint
19
20         another coroutine handle:
21             auto anotherCoroutineHandle = awaiter.await_suspend(coroutineHandle);
22             anotherCoroutineHandle.resume();
```

```
23              return to caller
24
25  resumptionPoint:
26
27  return awaiter.await_resume();
```

The workflow is only executed if `awaiter.await_ready()` returns `false` (line 1). In case it returns `true`, the coroutine is ready and returns with the result of the call `awaiter.await_resume()` (line 27).

Let me assume that `awaiter.await_ready()` returns `false`. First, the coroutine is suspended (line 3), and immediately the return value of `awaiter.await_suspend()` is evaluated. The return type can be `void` (line 7), a boolean (line 12), or another coroutine handle (line 20), such as `anotherCoroutineHandle`. Depending on the return type, the program flow returns or another coroutine is executed.

Return value of `awaiter.await_suspend()`

Type	Description
`void`	The coroutine keeps suspended and returns to the caller.
`bool`	`bool == true`: The coroutine keeps suspended and returns to the caller. `bool == false`: The coroutine is resumed and does not return to the caller.
`anotherCoroutineHandle`	The other coroutine is resumed and returns to the caller.

Whats happens in case an exception is thrown? It makes a difference if the exception occurs in `await_read`, `await_suspend`, or `await_resume`.

- `await_ready`: The coroutine is not suspended, and the calls `await_suspend` or `await_resume` are not evaluated.
- `await_suspend`: The exception is caught, the coroutine is resumed, and the exception rethrown. `await_resume` is not called.
- `await_resume`: `await_ready` and `await_suspend` are evaluated, and all values are returned. Of course, the call `await_resume` does not return a result.

Let me put theory into practice.

6.1.6 `co_return`

A coroutine uses **co_return** as its return statement.

6.1.6.1 A Future

Admittedly, the coroutine in the following program `eagerFuture.cpp` is the simplest coroutine I can imagine. Still it does something meaningful: it automatically stores the result of its invocation.

An eager future

```cpp
// eagerFuture.cpp

#include <coroutine>
#include <iostream>
#include <memory>

template<typename T>
struct MyFuture {
    std::shared_ptr<T> value;
    MyFuture(std::shared_ptr<T> p): value(p) {}
    ~MyFuture() { }
    T get() {
        return *value;
    }

    struct promise_type {
        std::shared_ptr<T> ptr = std::make_shared<T>();
        ~promise_type() { }
        MyFuture<T> get_return_object() {
            return ptr;
        }
        void return_value(T v) {
            *ptr = v;
        }
        std::suspend_never initial_suspend() {
            return {};
        }
        std::suspend_never final_suspend() noexcept {
            return {};
        }
        void unhandled_exception() {
            std::exit(1);
        }
    };
};

MyFuture<int> createFuture() {
    co_return 2021;
}

int main() {

    std::cout << '\n';

```

```
45      auto fut = createFuture();
46      std::cout << "fut.get(): " << fut.get() << '\n';
47
48      std::cout << '\n';
49
50  }
```

`MyFuture` behaves as a future[12] which runs immediately. The call of the coroutine `createFuture` (line 45) returns the future, and the call `fut.get` (line 46) picks up the result of the associated promise.

There is one subtle difference to a future, the return value of the coroutine `createFuture` is available after its invocation. Due to the lifetime issues, the return value is managed by a `std::shared_ptr` (lines 9 and 17). The coroutine always uses `std::suspend_never` (lines 25 and 28) and, therefore, neither suspends before it runs nor after. This means the coroutine is executed when the function `createFuture` is invoked. The member function `get_return_object` (line 19) creates and stores the handle to the coroutine object, and `return_value` (lines 22) stores the result of the coroutine, which was provided by `co_return 2021` (line 38). The client invokes `fut.get` (line 46) and uses the future as a handle to the promise. The member function `get` returns the result to the client (line 13).

```
fut.get(): 2021
```

An eager future

You may think it is not worth the effort of implementing a coroutine that behaves just like a function. You are right! However, this simple coroutine is an ideal starting point for writing various implementations of futures. Read more about Variations of Futures in the chapter case studies.

6.1.7 `co_yield`

Thanks to `co_yield` you can implement a generator generating an infinite data stream from which you can successively query values. The return type of the generator `generatorForNumbers(int begin, int inc= 1)` is `generator<int>`, where the generator internally holds a special promise `p` such that a call `co_yield i` is equivalent to a call `co_await p.yield_value(i)`. Statement `co_yield i` can be called an arbitrary number of times. Immediately after each call, the execution of the coroutine is suspended.

6.1.7.1 An Infinite Data Stream

The program `infiniteDataStream.cpp` produces an infinite data stream. The coroutine `getNext` uses `co_yield` to create a data stream that starts at `start` and gives on request the next value, incremented by `step`.

[12] https://en.cppreference.com/w/cpp/thread/future

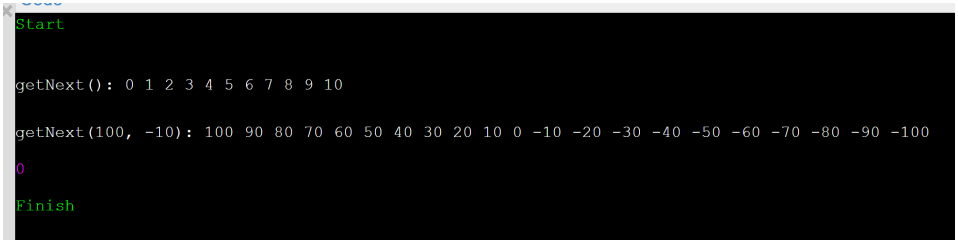
An infinite data stream

```
// infiniteDataStream.cpp

#include <coroutine>
#include <memory>
#include <iostream>

template<typename T>
struct Generator {

    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;

    Generator(handle_type h): coro(h) {}                          // (3)
    handle_type coro;

    ~Generator() {
        if ( coro ) coro.destroy();
    }
    Generator(const Generator&) = delete;
    Generator& operator = (const Generator&) = delete;
    Generator(Generator&& oth) noexcept : coro(oth.coro) {
        oth.coro = nullptr;
    }
    Generator& operator = (Generator&& oth) noexcept {
        coro = oth.coro;
        oth.coro = nullptr;
        return *this;
    }
    T getValue() {
        return coro.promise().current_value;
    }
    bool next() {                                                 // (5)
        coro.resume();
        return not coro.done();
    }
    struct promise_type {
        promise_type()  = default;                                // (1)

        ~promise_type() = default;

        auto initial_suspend() {                                  // (4)
            return std::suspend_always{};
        }
        auto final_suspend() noexcept {
```

```
45              return std::suspend_always{};
46          }
47          auto get_return_object() {                                  // (2)
48              return Generator{handle_type::from_promise(*this)};
49          }
50          auto return_void() {
51              return std::suspend_never{};
52          }
53
54          auto yield_value(const T value) {                           // (6)
55              current_value = value;
56              return std::suspend_always{};
57          }
58          void unhandled_exception() {
59              std::exit(1);
60          }
61          T current_value;
62      };
63
64  };
65
66  Generator<int> getNext(int start = 0, int step = 1) {
67      auto value = start;
68      while (true) {
69          co_yield value;
70          value += step;
71      }
72  }
73
74  int main() {
75
76      std::cout << '\n';
77
78      std::cout << "getNext():";
79      auto gen = getNext();
80      for (int i = 0; i <= 10; ++i) {
81          gen.next();
82          std::cout << " " << gen.getValue();                         // (7)
83      }
84
85      std::cout << "\n\n";
86
87      std::cout << "getNext(100, -10):";
88      auto gen2 = getNext(100, -10);
89      for (int i = 0; i <= 20; ++i) {
```

```
90          gen2.next();
91          std::cout << " " << gen2.getValue();
92      }
93
94      std::cout << '\n';
95
96  }
```

The `main` program creates two coroutines. The first one `gen` (line 79) returns the values from 0 to 10, and the second one `gen2` (line 88) the values from 100 to -100. Before I dive into the workflow, thanks to the online compiler Wandbox[13], here is the output of the program.



An infinite data stream

The numbers in the program `infiniteDataStream.cpp` stand for the steps in the first iteration of the workflow.

1. creates the promise
2. calls `promise.get_return_object()` and keeps the result in a local variable
3. creates the generator
4. calls `promise.initial_suspend()`. The generator is lazy and, therefore, always suspends.
5. asks for the next value and returns if the generator is consumed
6. triggered by the `co_yield` call. The next value is available thereafter.
7. gets the next value

In further iterations, only steps 5, 6, and 7 are performed.

Section Modification and Generalization of Threads in chapter case studies discusses further improvements and modifications of the generator `infiniteDataStream.cpp`.

6.1.8 `co_await`

`co_await` eventually causes the execution of the coroutine to be suspended or resumed. The expression `exp` in `co_await exp` has to be a so-called Awaitable expression, i.e. which must implement a specific interface, consisting of the three functions `await_ready`, `await_suspend`, and `await_resume`.

A typical use case for `co_await` is a server that waits for events.

[13] https://wandbox.org/

A blocking server

```
1  Acceptor acceptor{443};
2  while (true) {
3      Socket socket = acceptor.accept();              // blocking
4      auto request = socket.read();                   // blocking
5      auto response = handleRequest(request);
6      socket.write(response);                         // blocking
7  }
```

The server is quite simple because it sequentially answers each request in the same thread. The server listens on port 443 (line 1), accepts the connection (line 3), reads the incoming data from the client (line 4), and writes its answer to the client (line 6). The calls in lines 3, 4, and 6 are blocking.

Thanks to `co_await`, the blocking calls can now be suspended and resumed.

A waiting server

```
1  Acceptor acceptor{443};
2  while (true) {
3      Socket socket = co_await acceptor.accept();
4      auto request = co_await socket.read();
5      auto response = handleRequest(request);
6      co_await socket.write(response);
7  }
```

Before I present the challenging example of thread synchronization with coroutines, I want to start with something straightforward: starting a job on request.

6.1.8.1 Starting a Job on Request

The coroutine in the following example is as simple as it can be. It awaits on the predefined Awaitable `std::suspend_never()`.

Starting a job on request

```
1  // startJob.cpp
2
3  #include <coroutine>
4  #include <iostream>
5
6  struct Job {
7      struct promise_type;
8      using handle_type = std::coroutine_handle<promise_type>;
9      handle_type coro;
10     Job(handle_type h): coro(h){}
```

```
11      ~Job() {
12          if ( coro ) coro.destroy();
13      }
14      void start() {
15          coro.resume();
16      }
17
18
19      struct promise_type {
20          auto get_return_object() {
21              return Job{handle_type::from_promise(*this)};
22          }
23          std::suspend_always initial_suspend() {
24              std::cout << "    Preparing job" << '\n';
25              return {};
26          }
27          std::suspend_always final_suspend() noexcept {
28              std::cout << "    Performing job" << '\n';
29              return {};
30          }
31          void return_void() {}
32          void unhandled_exception() {}
33
34      };
35  };
36
37  Job prepareJob() {
38      co_await std::suspend_never();
39  }
40
41  int main() {
42
43      std::cout <<  "Before job" << '\n';
44
45      auto job = prepareJob();
46      job.start();
47
48      std::cout <<  "After job" <<  '\n';
49
50  }
```

You may think that the coroutine `prepareJob` (line 37) is meaningless because the Awaitable never suspends. No! The function `prepareJob` is at least a coroutine factory using `co_await` (line 38) and returning a coroutine object. The function call `prepareJob()` in line 45 creates the coroutine object of

type Job. When you study the data type Job, you recognize that the coroutine object is immediately suspended, because the member function of the promise returns the Awaitable std::suspend_always (line 23). This is exactly the reason why the function call job.start (line 46) is necessary to resume the coroutine (line 15). The member function final_suspend also returns std::suspend_always (line 27).

```
Before job
    Preparing job
    Performing job
After job
```

Starting a Job on Request

In the case studies' section various job flows, I use the program startJob as a starting point for further experiments.

6.1.8.2 Thread Synchronization

It's typical for threads to synchronize themselves. One thread prepares a work package, another thread awaits. Condition variables[14], promises and futures[15], and also an atomic boolean[16] can be used to create a sender-receiver workflow. Thanks to coroutines, thread synchronization is quite easy, without the inherent risks of condition variables such as spurious wakeups and lost wakeups.

Thread Synchronization

```
1  // senderReceiver.cpp
2
3  #include <coroutine>
4  #include <chrono>
5  #include <iostream>
6  #include <functional>
7  #include <string>
8  #include <stdexcept>
9  #include <atomic>
10 #include <thread>
11
12 class Event {
13  public:
14
15     Event() = default;
16
17     Event(const Event&) = delete;
```

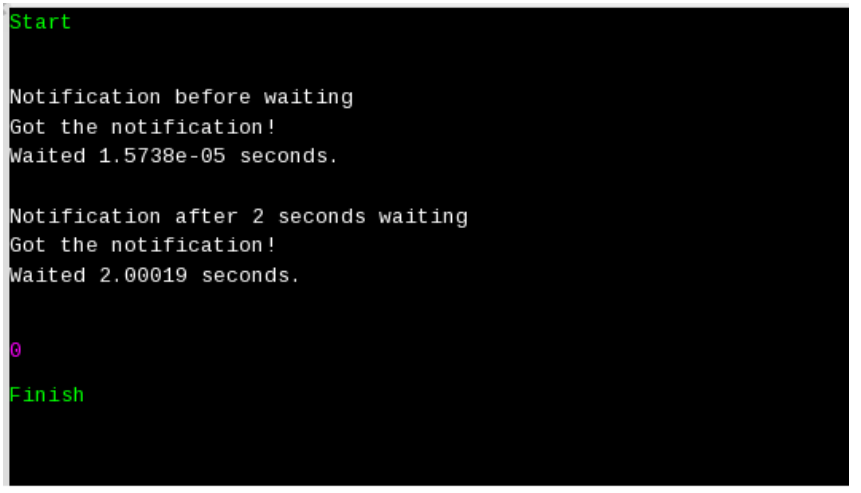
[14] https://en.cppreference.com/w/cpp/thread/condition_variable
[15] https://en.cppreference.com/w/cpp/thread
[16] https://en.cppreference.com/w/cpp/atomic/atomic

```
18      Event(Event&&) = delete;
19      Event& operator=(const Event&) = delete;
20      Event& operator=(Event&&) = delete;
21
22      class Awaiter;
23      Awaiter operator co_await() const noexcept;
24
25      void notify() noexcept;
26
27   private:
28
29      friend class Awaiter;
30
31      mutable std::atomic<void*> suspendedWaiter{nullptr};
32      mutable std::atomic<bool> notified{false};
33
34  };
35
36  class Event::Awaiter {
37   public:
38      Awaiter(const Event& eve): event(eve) {}
39
40      bool await_ready() const;
41      bool await_suspend(std::coroutine_handle<> corHandle) noexcept;
42      void await_resume() noexcept {}
43
44   private:
45      friend class Event;
46
47      const Event& event;
48      std::coroutine_handle<> coroutineHandle;
49  };
50
51  bool Event::Awaiter::await_ready() const {
52
53      // allow at most one waiter
54      if (event.suspendedWaiter.load() != nullptr){
55          throw std::runtime_error("More than one waiter is not valid");
56      }
57
58      // event.notified == false; suspends the coroutine
59      // event.notified == true; the coroutine is executed like a normal function
60      return event.notified;
61  }
62
```

```
63  bool Event::Awaiter::await_suspend(std::coroutine_handle<> corHandle) noexcept {
64          coroutineHandle = corHandle;
65
66          const Event& ev = event;
67          ev.suspendedWaiter.store(this);
68
69          if (ev.notified) {
70                  void* thisPtr = this;
71
72                  if (ev.suspendedWaiter.compare_exchange_strong(thisPtr, nullptr)) {
73                          return false;
74                  }
75          }
76
77          return true;
78  }
79
80  void Event::notify() noexcept {
81          notified = true;
82
83          void* waiter = suspendedWaiter.load();
84
85          if (waiter != nullptr && suspendedWaiter.compare_exchange_strong(waiter, nullptr)) {
86                  static_cast<Awaiter*>(waiter)->coroutineHandle.resume();
87          }
88  }
89
90  Event::Awaiter Event::operator co_await() const noexcept {
91      return Awaiter{ *this };
92  }
93
94  struct Task {
95      struct promise_type {
96          Task get_return_object() { return {}; }
97          std::suspend_never initial_suspend() { return {}; }
98          std::suspend_never final_suspend() noexcept { return {}; }
99          void return_void() {}
100         void unhandled_exception() {}
101     };
102 };
103
104 Task receiver(Event& event) {
105     auto start = std::chrono::high_resolution_clock::now();
106     co_await event;
107     std::cout << "Got the notification! " << '\n';
```

```
108     auto end = std::chrono::high_resolution_clock::now();
109     std::chrono::duration<double> elapsed = end - start;
110     std::cout << "Waited " << elapsed.count() << " seconds." << '\n';
111 }
112
113 using namespace std::chrono_literals;
114
115 int main() {
116
117     std::cout << '\n';
118
119     std::cout << "Notification before waiting" << '\n';
120     Event event1{};
121     auto senderThread1 = std::thread([&event1]{ event1.notify(); });  // Notification
122     auto receiverThread1 = std::thread(receiver, std::ref(event1));
123
124     receiverThread1.join();
125     senderThread1.join();
126
127     std::cout << '\n';
128
129     std::cout << "Notification after 2 seconds waiting" << '\n';
130     Event event2{};
131     auto receiverThread2 = std::thread(receiver, std::ref(event2));
132     auto senderThread2 = std::thread([&event2]{
133         std::this_thread::sleep_for(2s);
134         event2.notify();                                          // Notification
135     });
136
137     receiverThread2.join();
138     senderThread2.join();
139
140     std::cout << '\n';
141
142 }
```

From the user's perspective, thread synchronization with coroutines is straightforward. Let's have a look at the program `senderReceiver.cpp`. The threads `senderThread1` (line 121) and `senderThread2` (line 132) use an event to send its notification in lines 121 and 134. The function `receiver` in lines 104 - 111 is the coroutine, which is executed in threads `receiverThread1` (line 122) and `receiverThread2` (line 132). I measured the time between the beginning and the end of the coroutine and displayed it. This number shows how long the coroutine waits. The following screenshot shows the output of the program.



Thread synchronization

If you compare the class `Generator` in the infinite data stream with the class `Event` in this example, there is a subtle difference. In the first case, the `Generator` is the Awaitable and the Awaiter; in the second case, the `Event` uses the `operator co_await` to return the Awaiter. This separation of concerns into the Awaitable and the Awaiter improves the structure of the code.

The output displays that the execution of the second coroutine takes about two seconds. The reason is that the `event1` sends its notification (line 121) before the coroutine is suspended, but the `event2` sends its notification after a time duration of 2 seconds (line 134).

Now, I put the implementer's hat on. The workflow of the coroutine is quite challenging to grasp. The class `Event` has two interesting members: `suspendedWaiter` and `notified`. Variable `suspendedWaiter` in line 31 holds the waiter for the signal, and `notified` in line 32 has the state of the notification.

In my explanation of both workflows, I assume in the first case (first workflow) that the event notification happens before the coroutine awaits the events. For the second case (second workflow), I assume it is the other way around.

Let's first look at `event1` and the first workflow. Here, `event1` sends its notification before `receiverThread1` is started. The invocation `event1` (line 121) triggers the method `notify` (lines 80 to 88). First the notification flag is set, and then, the call `void* waiter = suspendedWaiter.load()` loads the potential `waiter`. In this case, the `waiter` is a `nullptr` because it was not set before. This means the following `resume` call on the `waiter` in line 86 is not executed. The subsequentially performed function `await_ready` (lines 51 - 61) checks first if there is more than one `waiter`. In this case, I throw a `std::runtime` exception. The crucial part of this method is the return value. `event.notification` was already set to `true` in the `notify` method. `true` means, in this case, that the coroutine is not suspended and executes such as a normal function.

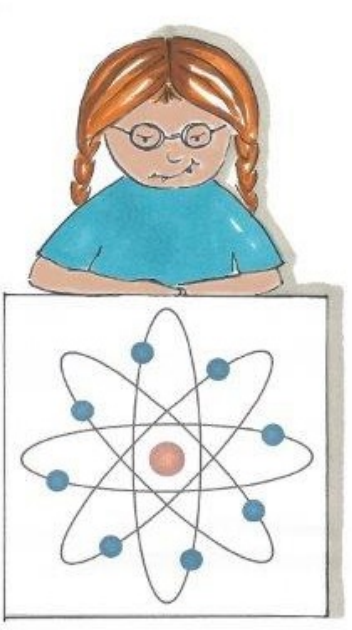
In the second workflow, the `co_await event2` call happens before `event2` sends its notification. `co_await event2` triggers the call `await_ready` (line 51). The big difference with the first workflow is that

`event.notified` is `false`. This `false` value causes the suspension of the coroutine. Technically, method `await_suspend` (lines 63 - 78) is executed. `await_suspend` gets the coroutine handle `corHandle` and stores it for later invocation in the variable `coroutineHandle` (line 64). Of course, later invocation means resumption. Second, the `waiter` is stored in the variable `suspendedWaiter`. When later `event2.notify` triggers its notification, method `notify` (line 80) is executed. The difference with the first workflow is that the condition `waiter != nullptr` evaluates to `true`. The result is that the `waiter` uses the `coroutineHandle` to resume the coroutine.

Distilled Information

- Coroutines are generalized functions that can pause and resume their execution while keeping their state.
- With C++20, we don't get concrete coroutines but a framework for implementing coroutines. This framework consists of more than 20 functions that you partially have to implement and partially could overwrite.
- With the new keywords `co_await` and `co_yield`, C++20 extends the execution of C++ functions with two new concepts.
- Thanks to `co_await expression`, it is possible to suspend and resume the execution of the expression. If you use `co_await expression` in a function `func`, the call `auto getResult = func()` does not block if the function's result is not available. Instead of resource-consuming blocking, you have resource-friendly waiting.
- `co_yield` empowers you to write infinite data streams.

6.2 Atomics



Cippi studies the atomics

Atomics receives a few important extensions in C++20. Probably the most important ones are atomic references and atomic smart pointers.

6.2.1 `std::atomic_ref`

The class template `std::atomic_ref` applies atomic operations to the referenced object.

Concurrent writing and reading of an atomic object ensure that there is no data race. The lifetime of the referenced object must exceed the lifetime of the `atomic_ref`. If any `atomic_ref` accesses an object, all other accesses to the object must use an `atomic_ref`. In addition, no subobject of the `atomic_ref`-accessed object may be accessed by another `atomic_ref`.

6.2.1.1 Motivation

Stop. You may think that using a reference inside an atomic would do the job. Unfortunately not.

In the following program, I have a class `ExpensiveToCopy`, which includes a `counter`. A few threads concurrently increment the `counter`. Consequently, `counter` has to be protected.

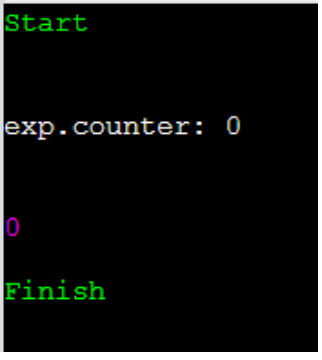
Using an atomic reference

```cpp
// atomicReference.cpp

#include <atomic>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

struct ExpensiveToCopy {
    int counter{};
};

int getRandom(int begin, int end) {

    std::random_device seed;        // initial seed
    std::mt19937 engine(seed());    // generator
    std::uniform_int_distribution<> uniformDist(begin, end);

    return uniformDist(engine);
}

void count(ExpensiveToCopy& exp) {

    std::vector<std::thread> v;
    std::atomic<int> counter{exp.counter};

    for (int n = 0; n < 10; ++n) {
        v.emplace_back([&counter] {
            auto randomNumber = getRandom(100, 200);
            for (int i = 0; i < randomNumber; ++i) { ++counter; }
        });
    }

    for (auto& t : v) t.join();

}

int main() {

    std::cout << '\n';

    ExpensiveToCopy exp;
    count(exp);
    std::cout << "exp.counter: " << exp.counter << '\n';
```

```
45
46      std::cout << '\n';
47
48  }
```

Variable `exp` (line 42) is the expensive-to-copy object. For performance reasons, the function `count` (line 22) takes `exp` by reference. Function `count` initializes the `std::atomic<int>` with `exp.counter` (line 25). The following lines create ten threads (line 27), each performing the lambda expression, which takes `counter` by reference. The lambda expression gets a random number between 100 and 200 (line 29) and increments the counter exactly as often. The function `getRandom` (line 13) starts with an initial seed and creates via the random-number generator Mersenne Twister[17] a uniform distributed number between 100 and 200.

In the end, the `exp.counter` (line 44) should have an approximate value of 1500 because ten threads increment on average 150 times. Executing the program on the Wandbox online compiler[18] gives me a surprising result.



Surprise with an atomic reference

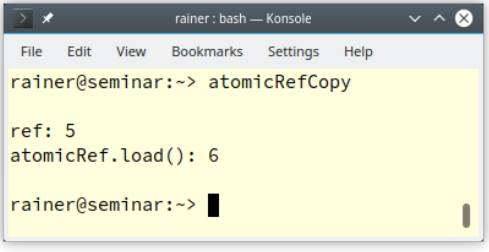
The counter is 0. What is happening? The issue is in line 25. The initialization in the expression `std::atomic<int> counter{exp.counter}` creates a copy. The following small program exemplifies the issue.

[17] https://en.wikipedia.org/wiki/Mersenne_Twister
[18] https://wandbox.org/

Copying the reference

```
1  // atomicRefCopy.cpp
2
3  #include <atomic>
4  #include <iostream>
5
6  int main() {
7
8      std::cout << '\n';
9
10     int val{5};
11     int& ref = val;
12     std::atomic<int> atomicRef(ref);
13     ++atomicRef;
14     std::cout << "ref: " << ref << '\n';
15     std::cout << "atomicRef.load(): " << atomicRef.load() << '\n';
16
17     std::cout << '\n';
18
19 }
```

The increment operation in line 13 does not address the reference `ref` (line 11). The value of `ref` is not changed.



Copying the reference

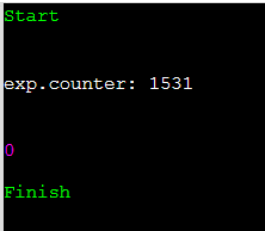
Replacing the `std::atomic<int>` with `std::atomic_ref<int>` solves the issue.

Using a `std::atomic_ref`

```
// atomicRef.cpp

#include <atomic>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

struct ExpensiveToCopy {
    int counter{};
};

int getRandom(int begin, int end) {

    std::random_device seed;        // initial randomness
    std::mt19937 engine(seed());    // generator
    std::uniform_int_distribution<> uniformDist(begin, end);

    return uniformDist(engine);
}

void count(ExpensiveToCopy& exp) {

    std::vector<std::thread> v;
    std::atomic_ref<int> counter{exp.counter};

    for (int n = 0; n < 10; ++n) {
        v.emplace_back([&counter] {
            auto randomNumber = getRandom(100, 200);
            for (int i = 0; i < randomNumber; ++i) { ++counter; }
        });
    }

    for (auto& t : v) t.join();

}

int main() {

    std::cout << '\n';

    ExpensiveToCopy exp;
    count(exp);
    std::cout << "exp.counter: " << exp.counter << '\n';
```

```
    std::cout << '\n';

}
```

Now, the value of `counter` is as expected:



The expected result with `std::atomic_ref`

In keeping with `std::atomic`[19], type `std::atomic_ref` can be specialized and supports specializations for the built-in data types.

6.2.1.2 Specializations of `std::atomic_ref` (C++20)

You can specialize `std::atomic_ref` for user-defined types, use partial specializations for pointer types, or full specializations for arithmetic types such as integral or floating-point types.

6.2.1.2.1 Primary Template

The primary template `std::atomic_ref` can be instantiated with a TriviallyCopyable[20] type T.

```
struct Counters {
    int a;
    int b;
};

Counter counter;
std::atomic_ref<Counters> cnt(counter);
```

6.2.1.2.2 Partial Specializations for Pointer Types

The standard provides partial specializations for a pointer type: `std::atomic_ref<T*>`.

[19] https://en.cppreference.com/w/cpp/atomic/atomic
[20] https://en.cppreference.com/w/cpp/types/is_trivially_copyable

6.2.1.2.3 Specializations for Arithmetic Types

The standard provides specialization for the integral and floating-point types: `std::atomic_ref<arithmetic type>`.

- Character types: `char`, `char8_t` (C++20), `char16_t`, `char32_t`, and `wchar_t`
- Standard signed-integer types: `signed char`, `short`, `int`, `long`, and `long long`
- Standard unsigned-integer types: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`
- Additional integer types, defined in the header `<cstdint>`[21]:
  - `int8_t`, `int16_t`, `int32_t`, and `int64_t` (signed integer with exactly 8, 16, 32, and 64 bits)
  - `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t` (unsigned integer with exactly 8, 16, 32, and 64 bits)
  - `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, and `int_fast64_t` (fastest signed integer with at least 8, 16, 32, and 64 bits)
  - `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, and `uint_fast64_t` (fastest unsigned integer with at least 8, 16, 32, and 64 bits)
  - `int_least8_t`, `int_least16_t`, `int_least32_t`, and `int_least64_t` (smallest signed integer with at least 8, 16, 32, and 64 bits)
  - `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, and `uint_least64_t` (smallest unsigned integer with at least 8, 16, 32, and 64 bits)
  - `intmax_t`, and `uintmax_t` (maximum signed and unsigned integer)
  - `intptr_t`, and `uintptr_t` (signed and unsigned integer for holding a pointer)
- Standard floating-point types: `float`, `double`, and `long double`

6.2.1.2.4 All Atomic Operations

First, here is the list of all operations on `atomic_ref`.

All operations on `atomic_ref`

Function	Description
`is_lock_free`	Checks if the `atomic_ref` object is lock-free.
`atomic_ref<T>::is_always_lock_free`	Checks at compile time if the atomic type is always lock-free.
`load`	Atomically returns the value of the referenced object.
`operator T`	Atomically returns the value of the atomic. Equivalent to `atom.load()`.

[21] http://en.cppreference.com/w/cpp/header/cstdint

All operations on `atomic_ref`

Function	Description
`store`	Atomically replaces the value of the referenced object with a non-atomic.
`exchange`	Atomically replaces the value of the referenced object with the new value.
`compare_exchange_strong` `compare_exchange_weak`	Atomically compares and eventually exchanges the value of the referenced object.
`fetch_add`, `+=` `fetch_sub`, `-=`	Atomically adds (subtracts) the value to (from) the referenced object.

`fetch_or`, `|=`
`fetch_and`, `&=`
`fetch_xor`, `^=`
	Atomically performs bitwise (AND, OR, and XOR) operation on the referenced object.

`++`, `--`	Increments or decrements (either pre- and post-increment) the referenced object.
`notify_one` `notify_all`	Notifies one atomic wait operation. Notifies all atomic wait operations.
`wait`	Blocks until it is notified. Compares itself with the `old` value to protect against spurious wakeups. If the value is different from the `old` value, returns.

The composite assignment operators (`+=`, `-=`, `|=`, `&=`, or `^=`) return the new value; the `fetch` variations return the old value.

Thanks to the `constexpr` function `atomic_ref<type>::is_always_lock_free`, you can check for each atomic type if it's lock-free on all supported hardware that the executable might run on. This check returns only `true` if it is `true` for all supported hardware. The check is performed at compile-time and is available since C++17.

Each function supports an additional memory-ordering argument. The default for the memory-ordering argument is `std::memory_order_seq_cst`, but you can also use `std::memory_order_relaxed`, `std::memory_order_consume`, `std::memory_order_acquire`, `std::memory_order_release`, or `std::memory_order_acq_rel`. The `compare_exchange_strong` and `compare_exchange_weak` member functions can be parameterized with two memory orderings, one for the success case, and the other for the failure case. Both calls perform an atomic `exchange` if equal and an atomic `load` if not. They return `true` in the success case, `false` otherwise. If you only explicitly provide one memory ordering, it is used for both

the success and the failure case. Here are the details for memory ordering[22].

Of course, not all operations are available for all types referenced by `std::atomic_ref`. The table shows the list of all atomic operations, depending on the type referenced by `std::atomic_ref`.

All atomic operations, depending on the type referenced by `std::atomic_ref`

Function	atomic_ref<T>	atomic_ref<floating>	atomic_ref<T*>	atomic_ref<integral>
`is_lock_free`	yes	yes	yes	yes
`load`	yes	yes	yes	yes
`operator T`	yes	yes	yes	yes
`store`	yes	yes	yes	yes
`exchange`	yes	yes	yes	yes
`compare_exchange_strong`	yes	yes	yes	yes
`compare_exchange_weak`	yes	yes	yes	yes
`fetch_add, +=`		yes	yes	yes
`fetch_sub, -=`		yes	yes	yes
`fetch_or, |=`				yes
`fetch_and, &=`				yes
`fetch_xor, ^=`				yes
`++, --`			yes	yes
`notify_one`	yes	yes	yes	yes
`notify_all`	yes	yes	yes	yes
`wait`	yes	yes	yes	yes

6.2.2 Atomic Smart Pointer

A `std::shared_ptr`[23] consists of a control block and its resource. The control block is thread-safe, but access to the resource is not. This means modifying the reference counter is an atomic operation, and you have the guarantee that the resource is deleted exactly once. These are the guarantees `std::shared_ptr` gives you.

[22] https://en.cppreference.com/w/cpp/atomic/memory_order
[23] https://en.cppreference.com/w/cpp/memory/shared_ptr

The Importance of being Thread Safe

I want to take a short detour to emphasize how important it is that the `std::shared_ptr` has well-defined multithreading semantics. At first glance, using a `std::shared_ptr` does not appear to be a sensible choice for multithreaded code. It is by definition shared and mutable and is the ideal candidate for non-synchronized read and write operations and hence for undefined behavior. On the other hand, there is the guideline in modern C++: **Don't use raw pointers**. This means, consequently, that you should use smart pointers in multithreaded programs.

The proposal N4162[24] for atomic smart pointers directly addresses the deficiencies of the current implementation. The shortcomings boil down to these three points: consistency, correctness, and performance.

- **Consistency**: the atomic operations for `std::shared_ptr` are the only atomic operations for a non-atomic data type.
- **Correctness**: using global atomic operations is quite error-prone because the correct usage is based on discipline. It is easy to forget to use an atomic operation - such as using `ptr = localPtr` instead of `std::atomic_store(&ptr, localPtr)`. The result is undefined behavior because of a data race. If we used an atomic smart pointer instead, the type system would not allow it.
- **Performance**: the atomic smart pointers have a significant advantage compared to non-atomic versions. The atomic versions are designed for the particular use case and can internally have a `std::atomic_flag` as cheap spinlock[25]. Designing the non-atomic versions of the pointer functions to be thread-safe would be overkill when they are used in a single-threaded scenario. They would have a performance penalty.

The correctness argument is probably the most important one. Why? The answer lies in the proposal. The proposal presents a thread-safe singly-linked list that supports insertion, deletion, and searching of elements. This singly-linked list is implemented in a lock-free way.

[24] http://wg21.link/n4162
[25] https://en.wikipedia.org/wiki/Spinlock

6.2.2.1 A thread-safe singly-linked list

```
template<typename T> class concurrent_stack {
    struct Node { T t; shared_ptr<Node> next; };
    atomic_shared_ptr<Node> head;
          // in C++11: remove "atomic_" and remember to use the special
          // functions every time you touch the variable
    concurrent_stack( concurrent_stack &) =delete;
    void operator=(concurrent_stack&) =delete;

public:
    concurrent_stack() =default;
    ~concurrent_stack() =default;
    class reference {
        shared_ptr<Node> p;
    public:
       reference(shared_ptr<Node> p_) : p{p_} { }
       T& operator* () { return p->t; }
       T* operator->() { return &p->t; }
    };

    auto find( T t ) const {
        auto p = head.load();  // in C++11: atomic_load(&head)
        while( p && p->t != t )
            p = p->next;
        return reference(move(p));
    }
    auto front() const {
       return reference(head); // in C++11: atomic_load(&head)
    }
    void push_front( T t ) {
      auto p = make_shared<Node>();
      p->t = t;
      p->next = head;          // in C++11: atomic_load(&head)
      while( !head.compare_exchange_weak(p->next, p) ){ }
      // in C++11: atomic_compare_exchange_weak(&head, &p->next, p);
    }
    void pop_front() {
       auto p = head.load();
       while( p && !head.compare_exchange_weak(p, p->next) ){ }
       // in C++11: atomic_compare_exchange_weak(&head, &p, p->next);
    }
};
```

A thread-safe singly-linked list

All changes that are required to compile the program with a C++11 compiler are marked in red. The implementation with atomic smart pointers is much easier and hence less error-prone. C++20's type system does not permit using a non-atomic operation on an atomic smart pointer.

The proposal N4162[26] proposed the new types `std::atomic_shared_ptr` and `std::atomic_weak_ptr` as atomic smart pointers. By merging them into the mainline ISO C++ standard, they became partial template specializations of `std::atomic`, namely `std::atomic<std::shared_ptr<T>>` and `std::atomic<std::weak_ptr<T>>`.

The following program shows five threads modifying a `std::atomic<std::shared_ptr<std::string>>` without synchronization.

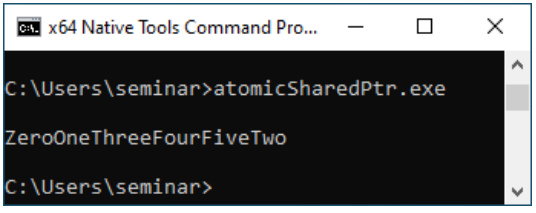
```
// atomicSharedPtr.cpp

#include <iostream>
#include <memory>
#include <atomic>
#include <string>
#include <thread>

int main() {

    std::cout << '\n';

    std::atomic<std::shared_ptr<std::string>> sharString(
        std::make_shared<std::string>("Zero"));

    std::thread t1([&sharString]{
        sharString.store(std::make_shared<std::string>(*sharString.load() + "One"));
    });
    std::thread t2([&sharString]{
        sharString.store(std::make_shared<std::string>(*sharString.load() + "Two"));
    });
    std::thread t3([&sharString]{
        sharString.store(std::make_shared<std::string>(*sharString.load() +"Three"));
    });
    std::thread t4([&sharString]{
        sharString.store(std::make_shared<std::string>(*sharString.load() +"Four"));
    });
    std::thread t5([&sharString]{
        sharString.store(std::make_shared<std::string>(*sharString.load() +"Five"));
    });

    t1.join();
```

[26] http://wg21.link/n4162

```
33      t2.join();
34      t3.join();
35      t4.join();
36      t5.join();
37
38      std::cout << *sharString.load() << '\n';
39
40  }
```

The atomic `std::shared_ptr shaString` (line 13) is initialized with the string "Zero". Each of the five threads `t1` to `t5` (lines 16 - 28) adds a string to `sharString` that is displayed in line 38. Using a `std::shared_ptr` instead of `std::atomic<std::shared_ptr>` would be a data race.

Executing the program shows the interleaving of the threads.



Thread-safe modification of a `std::string`

Consequently, the atomic operations for `std::shared_ptr` are deprecated with C++20.

6.2.3 `std::atomic_flag` Extensions

Before I write about `std::atomic_flag` extension in C++20, I want to give a short reminder of `std::atomic_flag` in C++11. If you want to read more details, read my post about `std::atomic_flag`[27] in C++11.

6.2.3.1 C++11

`std::atomic_flag` is a kind of atomic boolean. It has `clear`- and `set`-state functions. I call the clear state `false` and the set state `true` for simplicity. Its `clear` member function enables you to set its value to `false`. With the `test_and_set` method, you can set the value to `true` and return the previous value in an atomic step. `ATOMIC_FLAG_INIT` enables initializing the `std::atomic_flag` to `false`.

`std::atomic_flag` has two exciting properties, these are

- the only guaranteed lock-free atomic.
- the building block for higher thread abstractions.

With C++11, there is no member function to ask for the current value of a `std::atomic_flag` without modifying it. This changes with C++20.

[27] https://www.modernescpp.com/index.php/the-atomic-flag

6.2.3.2 C++20 Extensions

The following table shows the more powerful interface of `std::atomic_flag` in C++20.

All operations of `std::atomic_flag atomicFlag`

Method	Description
`atomicFlag.clear()`	Clears the atomic flag.
`atomicFlag.test_and_set()`	Sets the atomic flag and returns the old value.
`atomicFlag.test()` (C++20)	Returns the value of the flag.
`atomicFlag.notify_one()` (C++20)	Notifies one thread waiting on the atomic flag.
`atomicFlag.notify_all` (C++20)	Notifies all threads waiting on the atomic flag.
`atomicFlag.wait(bo)` (C++20)	Blocks the thread until notified and the atomic value changes.

The call `atomicFlag.test()` returns the `atomicFlag` value without changing it. Further on, you can use `std::atomic_flag` for thread synchronization: `atomicFlag.wait()`, `atomicFlag.notify_one()`, and `atomicFlag.notify_all()`. The member functions `notify_one` or `notify_all` notify one or all of the waiting atomic flags. `atomicFlag.wait(bo)` needs a boolean `bo`. The call `atomicFlag.wait(bo)` blocks until the next notification or spurious wakeup. It checks when the value of `atomicFlag` is equal to `bo` and unblocks if not. The value `bo` serves as a predicate to protect against spurious wakeups. A spurious wakeup is an erroneous notification.

Compared to C++11, the default construction of a `std::atomic_flag` is initialized to `false` state.

The remaining more powerful atomics can provide their functionality by using a mutex. That is according to the C++ standard. So these atomics have a member function `is_lock_free` to check if the atomic internally uses a mutex. On the popular platforms, I always get the answer false. But you should be aware of that. Thanks to the `constexpr` function `atomic<type>::is_always_lock_free`, you can check any atomic type if it's lock-free on each supported hardware that the executable might run on. This check returns only `true` if it is `true` for all supported hardware. The check is performed at compile-time and is available since C++17.

6.2.3.3 One Time Synchronization of Threads

Sender-receiver workflows are pretty common for threads. In such a workflow, the receiver is waiting for the sender's notification before Future continues to work. There are various ways to implement these workflows. With C++11, you can use condition variables or promise/future pairs; with C++20, you can use `std::atomic_flag`. Each way has its pros and cons. Consequently, I want to compare them. I assume you don't know the details of condition variables or promises and futures. Therefore, I give a short refresher.

6.2.3.3.1 Condition Variables

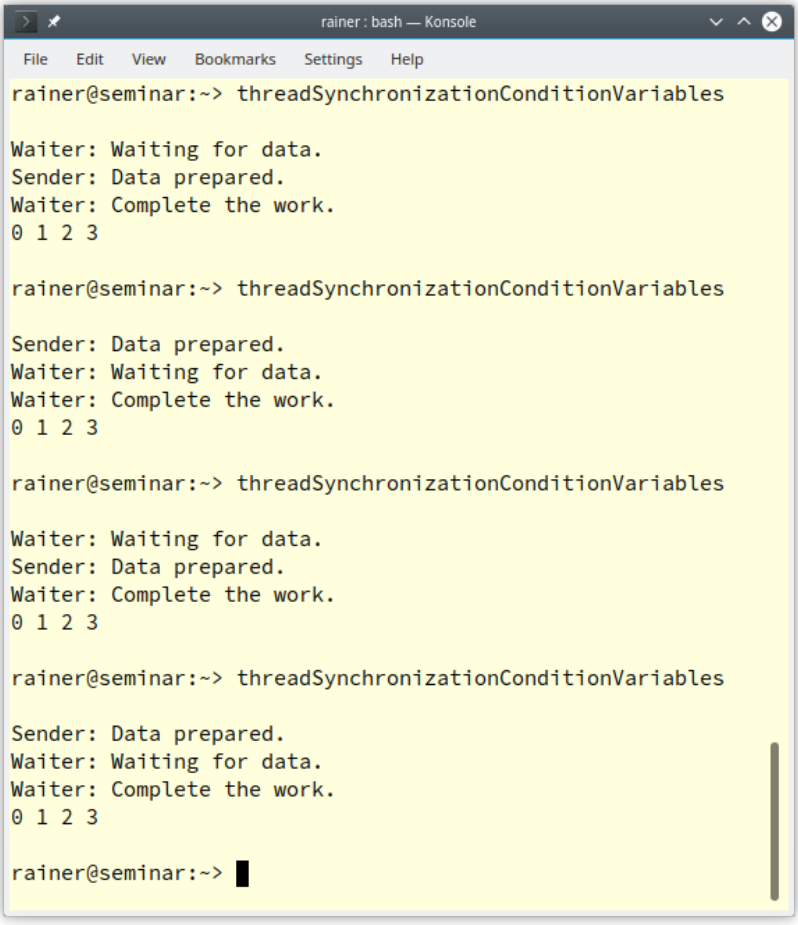
A condition variable can fulfill the role of a sender or a receiver. As a sender, it can notify one or more receivers.

Thread synchronization with condition variables

```
// threadSynchronizationConditionVariable.cpp

#include <iostream>
#include <condition_variable>
#include <mutex>
#include <thread>
#include <vector>

std::mutex mut;
std::condition_variable condVar;

std::vector<int> myVec{};

void prepareWork() {

    {
        std::lock_guard<std::mutex> lck(mut);
        myVec.insert(myVec.end(), {0, 1, 0, 3});
    }
    std::cout << "Sender: Data prepared."  << '\n';
    condVar.notify_one();
}

void completeWork() {

    std::cout << "Waiter: Waiting for data." << '\n';
    std::unique_lock<std::mutex> lck(mut);
    condVar.wait(lck, []{ return not myVec.empty(); });
    myVec[2] = 2;
    std::cout << "Waiter: Complete the work." << '\n';
    for (auto i: myVec) std::cout << i << " ";
    std::cout << '\n';

}

int main() {

    std::cout << '\n';

```

```
40      std::thread t1(prepareWork);
41      std::thread t2(completeWork);
42
43      t1.join();
44      t2.join();
45
46      std::cout << '\n';
47
48  }
```

The program has two child threads: `t1` and `t2`. They get their payload `prepareWork` and `completeWork` in lines 40 and 41. The function `prepareWork` (line 14) notifies that it is done with the preparation of the work: `condVar.notify_one()`. While holding the lock, thread `t2` is waiting for its notification: `condVar.wait(lck, []{ return not myVec.empty(); })`. The waiting thread always performs the same steps. When awoken, it checks the predicate while holding the lock (`[]{ return not myVec.empty();`). If the predicate does not hold, it puts itself back to sleep. If the predicate holds, it continues with its work. In the concrete workflow, the sending thread puts the initial values into the `std::vector` (line 18), which the receiving thread completes (line 29).



Thread synchronization with condition variables

Condition variables have many inherent issues. For example, the receiver could be awakened without notification or could lose the notification. The first issue is known as a spurious wakeup and the second as a lost wakeup. The predicate protects against both flaws. The notification could be lost when the sender sends its notification before the receiver is in the wait state and does not use a predicate. Consequently, the receiver waits for something that never happens. This is a deadlock. When you study the program's output, you see that every second run would cause a deadlock if I did not use a predicate. Of course, it is possible to use condition variables without a predicate.

If you want to know the details of the sender-receiver workflow and the traps of condition variables, read my posts "C++ Core Guidelines: Be Aware of the Traps of Condition Variables"[28].

[28] https://www.modernescpp.com/index.php/c-core-guidelines-be-aware-of-the-traps-of-condition-variables

Let me implement the same workflow using a future/promise pair.

6.2.3.3.2 Futures and Promises

A promise can send a value, an exception, or a notification to its associated future. Here is the corresponding workflow using a promise and a future.

Thread synchronization with a promise/future pair

```
// threadSynchronizationPromiseFuture.cpp

#include <iostream>
#include <future>
#include <thread>
#include <vector>

std::vector<int> myVec{};

void prepareWork(std::promise<void> prom) {

    myVec.insert(myVec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared."  << '\n';
    prom.set_value();

}

void completeWork(std::future<void> fut){

    std::cout << "Waiter: Waiting for data." << '\n';
    fut.wait();
    myVec[2] = 2;
    std::cout << "Waiter: Complete the work." << '\n';
    for (auto i: myVec) std::cout << i << " ";
    std::cout << '\n';

}

int main() {

    std::cout << '\n';

    std::promise<void> sendNotification;
    auto waitForNotification = sendNotification.get_future();

    std::thread t1(prepareWork, std::move(sendNotification));
    std::thread t2(completeWork, std::move(waitForNotification));
```

```
38
39      t1.join();
40      t2.join();
41
42      std::cout << '\n';
43
44  }
```

When you study the workflow, you recognize that the synchronization is reduced to its essential parts: `prom.set_value()` (line 14) and `fut.wait()` (line 21). I skip the screenshot to this run because it is essentially the same as the previous run with condition variables.

Here is more information on promises and futures, often just called tasks[29].

6.2.3.3.3 `std::atomic_flag`

Now, I jump directly from C++11 to C++20.

Thread synchronization with a `std::atomic_flag`

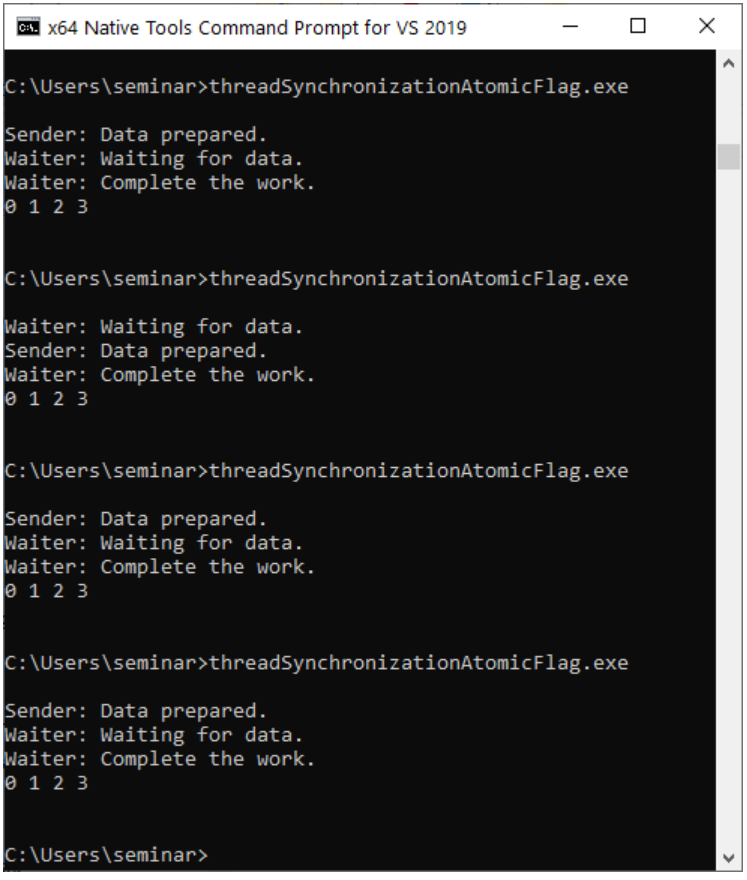
```
1   // threadSynchronizationAtomicFlag.cpp
2
3   #include <atomic>
4   #include <iostream>
5   #include <thread>
6   #include <vector>
7
8   std::vector<int> myVec{};
9
10  std::atomic_flag atomicFlag{};
11
12  void prepareWork() {
13
14      myVec.insert(myVec.end(), {0, 1, 0, 3});
15      std::cout << "Sender: Data prepared."  << '\n';
16      atomicFlag.test_and_set();
17      atomicFlag.notify_one();
18
19  }
20
21  void completeWork() {
22
23      std::cout << "Waiter: Waiting for data." << '\n';
24      atomicFlag.wait(false);
```

[29] https://www.modernescpp.com/index.php/tag/tasks

```
25      myVec[2] = 2;
26      std::cout << "Waiter: Complete the work." << '\n';
27      for (auto i: myVec) std::cout << i << " ";
28      std::cout << '\n';
29
30  }
31
32  int main() {
33
34      std::cout << '\n';
35
36      std::thread t1(prepareWork);
37      std::thread t2(completeWork);
38
39      t1.join();
40      t2.join();
41
42      std::cout << '\n';
43
44  }
```

The thread preparing the work (line 16) sets the `atomicFlag` to `true` and sends the notification. The thread that complets the work is waiting for the notification. It is only unblocked if `atomicFlag` is equal to `true`.

Here are a few runs of the program with the Microsoft Compiler.



Thread synchronization with `std::atomic_flag`

6.2.4 `std::atomic` Extensions

In C++20, `std::atomic`. like `std::atomic_ref`, `std::atomic`[30] can be instantiated with floating-point types such as `float`, `double`, and `long double`. In addition, `std::atomic_flag` and `std::atomic` can be used for thread synchronization via the member functions `notify_one`, `notify_all`, and `wait`. Notifying and waiting are available on all partial and full specializations of `std::atomic` (bools, integrals, floats, and pointers) and `std::atomic_ref`.

Thanks to `atomic<bool>`, the previous program `threadSynchronizationAtomicFlag.cpp` can directly be reimplemented.

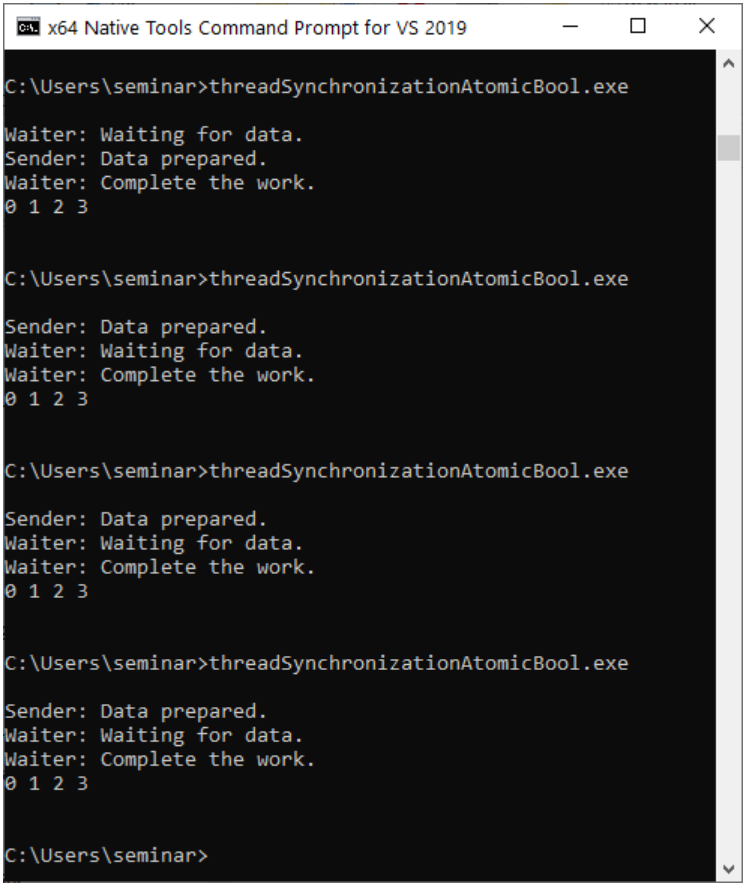
[30] https://en.cppreference.com/w/cpp/atomic/atomic

Thread synchronization with `std::atomic<bool>`

```
// threadSynchronizationAtomicBool.cpp

#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> myVec{};

std::atomic<bool> atomicBool{false};

void prepareWork() {

    myVec.insert(myVec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared."  << '\n';
    atomicBool.store(true);
    atomicBool.notify_one();

}

void completeWork() {

    std::cout << "Waiter: Waiting for data." << '\n';
    atomicBool.wait(false);
    myVec[2] = 2;
    std::cout << "Waiter: Complete the work." << '\n';
    for (auto i: myVec) std::cout << i << " ";
    std::cout << '\n';

}

int main() {

    std::cout << '\n';

    std::thread t1(prepareWork);
    std::thread t2(completeWork);

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

    std::cout << '\n';

}
```

The call `atomicBool.wait(false)` blocks if `atomicBool == false` holds. Consequently, the call `atomicBool.store(true)` (line 16) sets `atomicBool` to `true` and sends its notification.

As before, here are four runs with the Microsoft Compiler.



Thread synchronization with `std::atomic<bool>`

Condition Variables versus Promise/Future Pairs versus `std::atomic_flag`

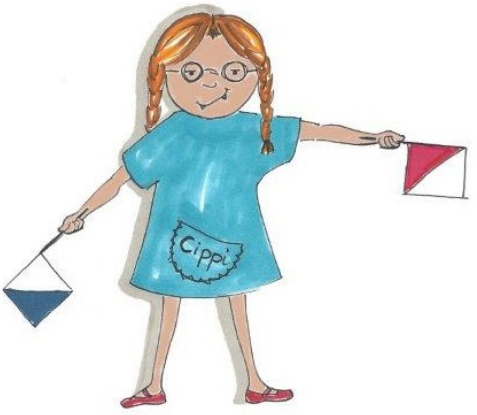
When you only need a one-time notification, such as in the previous program `threadSynchronizationConditionVariable.cpp`, promises, and futures are a better choice than condition variables. Promises and futures cannot be victims of spurious or lost wakeups. Furthermore, there is neither a need to use locks or mutexes nor is there a need to use a predicate to protect against spurious or lost wakeups. There use of promises and futures has only one disadvantege: they can only be used once.

I'm not sure if I would use a future/promise pair or atomics such as `std::atomic_flag` or `std::atomic<bool>` for such a simple thread-synchronization workflow. All are thread-safe by design and require no protection mechanism so far. Promises and futures are easier to use, and atomics are probably faster. I am only sure that if possible I would not use a condition variable.

Distilled Information

- `std::atomic_ref` applies atomic operations to the referenced object. Concurrent writing and reading are atomic for referenced objects, with no data race. The lifetime of the referenced object must exceed the lifetime of the `std::atomic_ref`.
- A `std::shared_ptr` consists of a control block and its resource. The control block is thread-safe, but the access to the resource is not. With C++20, we have an atomic shared pointer: `std::atomic<std::shared_ptr<T>>`, and `std::atomic<std::weak_ptr<T>>`.
- `std::atomic_flag` as a kind of atomic boolean is the only guaranteed lock-free data structure in C++. Its limited interface is extended in C++20. You can return its value, and you can use it for thread synchronization.
- `std::atomic`, introduced in C++11, gets various improvements in C++20. You can specialize a `std::atomic` for a floating-point value and use it for thread synchronization.

6.3 Semaphores



Cippi directs the train

Semaphores are a synchronization mechanism used to control concurrent access to a shared resource. A counting semaphore is a special semaphore that has a counter greater than zero. The counter is initialized in the constructor. Acquiring the semaphore decreases the counter, and releasing the semaphore increases the counter. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

Edsger W. Dijkstra invented Semaphores

The Dutch computer scientist Edsger W. Dijkstra[31] presented in 1965 the concept of a semaphore. A semaphore is a data structure with a queue and a counter. The counter is initialized to a value equal to or greater than zero. It supports the two operations `wait` and `signal`. Operation `wait` acquires the semaphore and decreases the counter. It blocks the thread from acquiring the semaphore if the counter is zero. Operation `signal` releases the semaphore and increases the counter. Blocked threads are added to the queue to avoid starvation[32].

Originally, a semaphore was a railway signal.

[31] https://en.wikipedia.org/wiki/Edsger_W._Dijkstra
[32] https://en.wikipedia.org/wiki/Starvation_(computer_science)

Semaphore

The original uploader was AmosWolfe at English Wikipedia. - Transferred from en.wikipedia to Commons., CC BY 2.0,[33]

C++20 supports a `std::binary_semaphore`, which is an alias for a `std::counting_semaphore<1>`. In this case, the least maximal value is 1. `std::binary_semaphores` can be used to implement locks[34].

```
using binary_semaphore = std::counting_semaphore<1>;
```

In contrast to a `std::mutex`, a `std::counting_semaphore` is not bound to a thread. This means that the acquisition and release of a semaphore call can happen on different threads. The following table presents the interface of a `std::counting_semaphore`.

Member functions of a `std::counting_semaphore sem`

Member function	Description
`std::semaphore sem{num}`	Creates a semaphore with the counter `num`. `cnt` must be a signed integral.
`sem.max()` (static)	Returns the maximum value of the `counter`.
`sem.release(upd = 1)`	Increases `counter` by `upd` and subsequently unblocks threads acquiring the semaphore `sem`.
`sem.acquire()`	Decrements the `counter` by 1 or blocks until the `counter` is greater than 0.

[33] https://commons.wikimedia.org/w/index.php?curid=1972304
[34] https://en.cppreference.com/w/cpp/named_req/BasicLockable

Member functions of a `std::counting_semaphore sem`

Member function	Description
`sem.try_acquire()`	Tries to decrement the `counter` by 1 if it is greater than 0.
`sem.try_acquire_for(relTime)`	Tries to decrement the `counter` by 1 or blocks for at most `relTime` if the counter is 0.
`sem.try_acquire_until(absTime)`	Tries to decrement the `counter` by 1 or blocks at most until `absTime` if the counter is 0.

The constructor call `std::counting_semaphore<10> sem(5)` creates a semaphore `sem` with at least a maximal value of 10 and a counter of 5. The call `sem.max()` returns the maximum possible value of the internal counter. The following relations must hold for `upd` in `sem.release(upd = 1)`: `update >= 0` and `update + counter <= sem.max()`. `sem.try_aquire_for(relTime)` needs a time duration; the member function `sem.try_acquire_until(absTime)` needs a time point. The three calls `sem.try_acquire`, `sem.try_acquire_for`, and `sem.try_acquire_until` return a boolean indicating the success of the calls.

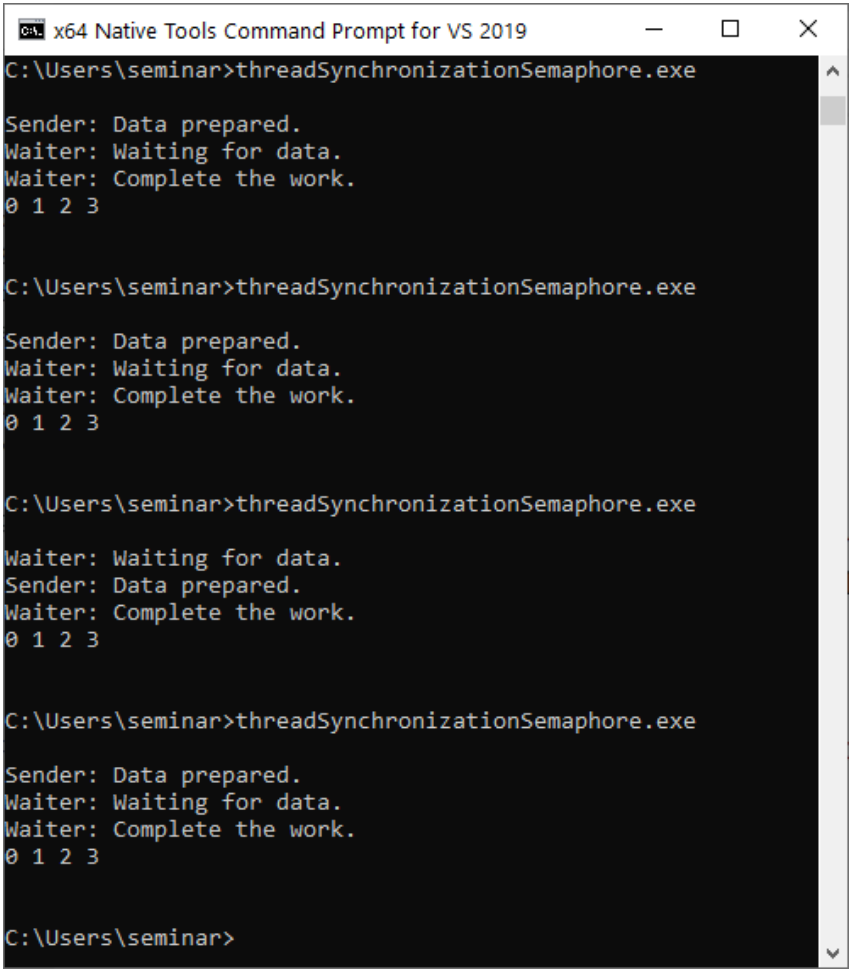
Semaphores are typically used in sender-receiver workflows. For example, initializing the semaphore `sem` with 0 will block the receiver's `sem.acquire()` call until the sender calls `sem.release()`. Consequently, the receiver waits for the notification of the sender. One-time synchronization of threads can easily be implemented using semaphores.

Thread synchronization with a `std::counting_semaphore`

```
1  // threadSynchronizationSemaphore.cpp
2
3  #include <iostream>
4  #include <semaphore>
5  #include <thread>
6  #include <vector>
7
8  std::vector<int> myVec{};
9
10 std::counting_semaphore<1> prepareSignal(0);
11
12 void prepareWork() {
13
14     myVec.insert(myVec.end(), {0, 1, 0, 3});
15     std::cout << "Sender: Data prepared."  << '\n';
16     prepareSignal.release();
17 }
18
19 void completeWork() {
20
```

```
21      std::cout << "Waiter: Waiting for data." << '\n';
22      prepareSignal.acquire();
23      myVec[2] = 2;
24      std::cout << "Waiter: Complete the work." << '\n';
25      for (auto i: myVec) std::cout << i << " ";
26      std::cout << '\n';
27
28  }
29
30  int main() {
31
32      std::cout << '\n';
33
34      std::thread t1(prepareWork);
35      std::thread t2(completeWork);
36
37      t1.join();
38      t2.join();
39
40      std::cout << '\n';
41
42  }
```

The `std::counting_semaphore prepareSignal` (line 10) can have the values 0 and 1. In the concrete example, it's initialized with 0 (line 10). This means, that the call `prepareSignal.release()` sets the value to 1 (line 16) and unblocks the call `prepareSignal.acquire()` (line 22).



Thread synchronization with semaphores

Distilled Information

- Semaphores are a synchronization mechanism used to control concurrent access to a shared resource.
- A counting semaphore in C++20 has a counter. Acquiring the semaphore decreases the counter, and releasing the semaphore increases the counter. If a thread tries to acquire the semaphore when the counter is zero, the thread blocks until another thread increments the counter by releasing the semaphore.

6.4 Latches and Barriers

Cippi waits at the barrier

Latches and barriers are coordination types that enable some threads to block until a counter becomes zero. In C++20 we get latches and barriers in two variations: `std::latch` and `std::barrier`. Concurrent invocations of the member functions of a `std::latch` or a `std::barrier` produce no data race.

First, there are two questions:

1. What are the differences between these two mechanisms to coordinate threads? You can use a `std::latch` only once, but you can use a `std::barrier` more than once. A `std::latch` helps to manage one task by multiple threads. A `std::barrier` helps to manage repeated tasks by multiple threads. Additionally, a `std::barrier` enables you to execute a function in the so-called completion step. The completion step is the state when the counter becomes zero.
2. What use cases do latches and barriers support that cannot be done in C++11 and C++14 with futures, threads, or condition variables combined with locks? Latches and barriers address no new use cases, but they are much easier to use. They are also more performant because they often use a lock-free mechanism internally.

6.4.1 `std::latch`

Now, let us have a closer look at the interface of a `std::latch`.

Member functions of a `std::latch lat`

Member function	Description
`std::latch lat{cnt}`	Creates a `std::latch` with counter `cnt`. `cnt` must be a signed integral.
`lat.count_down(upd = 1)`	Atomically decrements the counter by `upd` without blocking the caller.
`lat.try_wait()`	Returns `true` if `counter == 0`.
`lat.wait()`	Returns immediately if `counter == 0`. If not blocks until `counter == 0`.
`lat.arrive_and_wait(upd = 1)`	Equivalent to `count_down(upd); wait();`.
`std::latch::max`	Returns the maximum value of the counter supported by the implementation

The default value for `upd` is `1`. If `upd` is greater than the counter or negative, the behavior is undefined. The call `lat.try_wait()` never actually waits, as its name suggests.

The following program `bossWorkers.cpp` uses two `std::latch` to build a boss-workers workflow. I synchronized the output to `std::cout` using the function `synchronizedOut` (line 13). This synchronization makes it easier to follow the workflow.

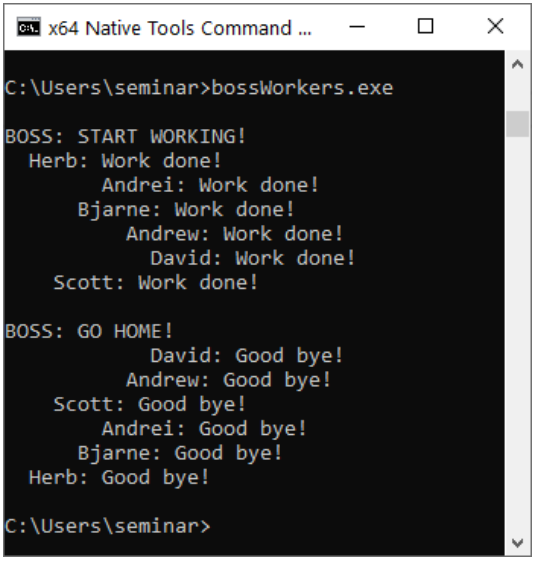
A boss-worker workflow using two `std::latch`

```
1  // bossWorkers.cpp
2
3  #include <iostream>
4  #include <mutex>
5  #include <latch>
6  #include <thread>
7
8  std::latch workDone(6);
9  std::latch goHome(1);
10
11 std::mutex coutMutex;
12
13 void synchronizedOut(const std::string& s) {
14     std::lock_guard<std::mutex> lo(coutMutex);
15     std::cout << s;
16 }
17
18 class Worker {
19  public:
20      Worker(std::string n): name(n) { }
21
```

```
22        void operator() (){
23            // notify the boss when work is done
24            synchronizedOut(name + ": " + "Work done!\n");
25            workDone.count_down();
26
27            // waiting before going home
28            goHome.wait();
29            synchronizedOut(name + ": " + "Good bye!\n");
30        }
31    private:
32        std::string name;
33    };
34
35    int main() {
36
37        std::cout << '\n';
38
39        std::cout << "BOSS: START WORKING! " << '\n';
40
41        Worker herb("  Herb");
42        std::thread herbWork(herb);
43
44        Worker scott("    Scott");
45        std::thread scottWork(scott);
46
47        Worker bjarne("      Bjarne");
48        std::thread bjarneWork(bjarne);
49
50        Worker andrei("        Andrei");
51        std::thread andreiWork(andrei);
52
53        Worker andrew("          Andrew");
54        std::thread andrewWork(andrew);
55
56        Worker david("            David");
57        std::thread davidWork(david);
58
59        workDone.wait();
60
61        std::cout << '\n';
62
63        goHome.count_down();
64
65        std::cout << "BOSS: GO HOME!" << '\n';
66
```

```
67      herbWork.join();
68      scottWork.join();
69      bjarneWork.join();
70      andreiWork.join();
71      andrewWork.join();
72      davidWork.join();
73
74  }
```

The idea of the workflow is straightforward. The six workers `herb`, `scott`, `bjarne`, `andrei`, `andrew`, and `david` (lines 41 - 57) have to do their job. When each has finished his job, it counts down the `std::latch workDone` (line 25). The boss (main thread) is blocked in line 59 until the counter becomes 0. When the counter is 0, the boss uses the second `std::latch goHome` to signal its workers to go home. In this case, the initial counter is 1 (line 9). The call `goHome.wait()` blocks until the counter becomes 0.



A boss-worker workflow using two `std::latch`

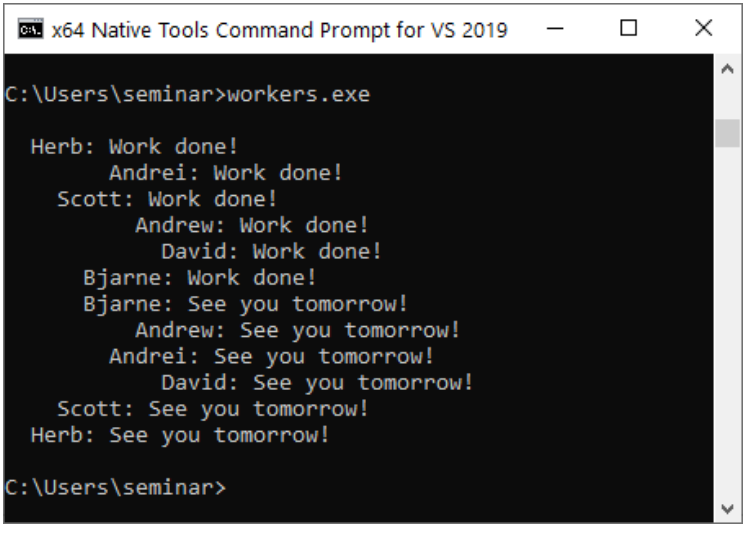
When you think about this workflow, you may notice that it can be done without a boss. Here it is.

A worker's workflow using a `std::latch`

```
// workers.cpp

#include <iostream>
#include <barrier>
#include <mutex>
#include <thread>

std::latch workDone(6);
std::mutex coutMutex;

void synchronizedOut(const std::string& s) {
    std::lock_guard<std::mutex> lo(coutMutex);
    std::cout << s;
}

class Worker {
 public:
    Worker(std::string n): name(n) { }

    void operator() () {
        synchronizedOut(name + ": " + "Work done!\n");
        workDone.arrive_and_wait();  // wait until all work is done
        synchronizedOut(name + ": " + "See you tomorrow!\n");
    }
 private:
    std::string name;
};

int main() {

    std::cout << '\n';

    Worker herb("  Herb");
    std::thread herbWork(herb);

    Worker scott("    Scott");
    std::thread scottWork(scott);

    Worker bjarne("      Bjarne");
    std::thread bjarneWork(bjarne);

    Worker andrei("        Andrei");
    std::thread andreiWork(andrei);

```

```
45      Worker andrew("          Andrew");
46      std::thread andrewWork(andrew);
47
48      Worker david("            David");
49      std::thread davidWork(david);
50
51      herbWork.join();
52      scottWork.join();
53      bjarneWork.join();
54      andreiWork.join();
55      andrewWork.join();
56      davidWork.join();
57
58  }
```

There is not much to add to this simplified workflow. The call `wordDone.arrive_and_wait()` (line 22) is equivalent to the calls `count_down(upd); wait();`. As a result, the workers coordinate themselves, and the boss is no longer necessary, as was the case in the previous program `bossWorkers.cpp`.



A workers workflow using a `std::latch`

A `std::barrier` is similar to a `std::latch`.

6.4.2 `std::barrier`

There are two differences between a `std::latch` and a `std::barrier`. First, you can use a `std::barrier` more than once, and second, you can adjust the counter for the next phase. The counter is set in the

constructor of `std::barrier bar`. Calling `bar.arrive()`, `bar.arrive_and_wait()`, and `bar.arrive_and_drop()` decrements the counter in the current phase. Additionally, `bar.arrive_and_drop()` decrements the counter for the next phase. Immediately after the current phase is finished and the counter becomes zero, the so-called completion step starts. In this completion step, a callable is invoked. The `std::barrier` gets its callable in its constructor. This callable must be declared as `noexcept`.

The completion step performs the following steps:

1. All threads are blocked.
2. An arbitrary thread is unblocked and executes the callable. The callable must be `noexcept`.
3. If the completion step is done, all threads are unblocked.

Member functions of a `std::barrier bar`

Member function	Description	fullfill
`std::barrier bar{cnt}`	Creates a `std::latch` with counter `cnt`. `cnt` must be a signed integral.	
`std::barrier bar{cnt, call}`	Creates a `std::barrier` with counter `cnt` and callable `call`.	
`bar.arrive(upd)`	Atomically decrements counter by `upd`.	
`bar.wait()`	Blocks at the synchronization point until the completion step is done.	
`bar.arrive_and_wait()`	Equivalent to `bar.wait(bar.arrive())`	
`bar.arrive_and_drop()`	Decrements the counter for the current and the subsequent phase by one.	
`std::barrier::max`	Maximum value supported by the implementation	

The call `bar.arrive_and_drop()` means essentially that the counter is decremented by one for the next phase.

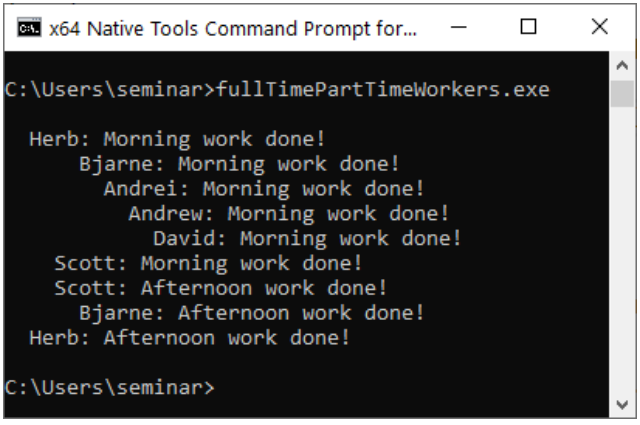
The program `fullTimePartTimeWorkers.cpp` halves the number of workers in the second phase.

Full-time and part-time workers

```cpp
// fullTimePartTimeWorkers.cpp

#include <iostream>
#include <barrier>
#include <mutex>
#include <string>
#include <thread>

std::barrier workDone(6);
std::mutex coutMutex;

void synchronizedOut(const std::string& s) {
    std::lock_guard<std::mutex> lo(coutMutex);
    std::cout << s;
}

class FullTimeWorker {
 public:
    FullTimeWorker(std::string n): name(n) { }

    void operator() () {
        synchronizedOut(name + ": " + "Morning work done!\n");
        workDone.arrive_and_wait();  // Wait until morning work is done
        synchronizedOut(name + ": " + "Afternoon work done!\n");
        workDone.arrive_and_wait();  // Wait until afternoon work is done

    }
 private:
    std::string name;
};

class PartTimeWorker {
 public:
    PartTimeWorker(std::string n): name(n) { }

    void operator() () {
        synchronizedOut(name + ": " + "Morning work done!\n");
        workDone.arrive_and_drop();  // Wait until morning work is done
    }
 private:
    std::string name;
};

int main() {
```

```
45
46      std::cout << '\n';
47
48      FullTimeWorker herb("  Herb");
49      std::thread herbWork(herb);
50
51      FullTimeWorker scott("    Scott");
52      std::thread scottWork(scott);
53
54      FullTimeWorker bjarne("      Bjarne");
55      std::thread bjarneWork(bjarne);
56
57      PartTimeWorker andrei("        Andrei");
58      std::thread andreiWork(andrei);
59
60      PartTimeWorker andrew("          Andrew");
61      std::thread andrewWork(andrew);
62
63      PartTimeWorker david("            David");
64      std::thread davidWork(david);
65
66      herbWork.join();
67      scottWork.join();
68      bjarneWork.join();
69      andreiWork.join();
70      andrewWork.join();
71      davidWork.join();
72
73  }
```

This workflow consists of two kinds of workers: full-time workers (line 17) and part-time workers (line 32). The part-time worker works in the morning, and the full-time worker in the morning and the afternoon. Consequently, the full-time workers call `workDone.arrive_and_wait()` (lines 23 and 25) two times. On the contrary, the part-time workers call `workDone.arrive_and_drop()` (line 38) only once. This `workDone.arrive_and_drop()` call causes the part-time worker to skip the afternoon work. Accordingly, the counter has in the first phase (morning) the value 6, and in the second phase (afternoon) the value 3.



Full-time and part-time workers

Distilled Information

- Latches and barriers are coordination types that enable some threads to block until a counter becomes zero. You can use a `std::latch` only once, but you can use a `std::barrier` more than once.
- A `std::latch` is useful for managing one task by multiple threads; a `std::barrier` helps to manage repeated tasks by multiple threads.

6.5 Cooperative Interruption

Cippi stops in front of the stop sign

The functionality of cooperative interruption is based on the three classes `std::stop_source`, `std::stop_token`, and the `std::stop_callback`. `std::jthread` and `std::condition_variable_any` support an explicit interface for the cooperative interruption.

First, why is it not a good idea to kill a thread?

Killing a Thread is Dangerous

Killing a thread is dangerous because you don't know the state of the thread. Here are two possible malicious outcomes.

- The thread is only half-done with its job. Consequently, you don't know the state of its job and, hence, the state of your program. You end with undefined behavior, and all bets are off.
- The thread may be in a critical section and have locked a mutex. Killing a thread while it locks a mutex ends with a high probability in a deadlock.

The `std::stop_source`, `std::stop_token`, and the `std::stop_callback` classes allows a thread to asynchronously request an execution to stop or ask if an execution got a stop signal. The `std::stop_token` can be passed to an operation and then used to poll actively the token for a stop request or to register a callback via `std::stop_callback`. The `std::stop_source sends` the stop request. This signal

affects all associated `std::stop_token`. The three classes, `std::stop_source`, `std::stop_token`, and the `std::stop_callback` share the ownership of an associated stop state. The stop state is allocated on the heap and automatically released when it is not needed anymore. This cooperative interruption facility is, by design, thread-safe.

In the following subsections, I provide more details about the cooperative interruption.

6.5.1 `std::stop_source`

You can construct a `std::stop_source` in two ways:

Constructors of `std::stop_source`

```
1 std::stop_source();
2 explicit std::stop_source(std::nostopstate_t) noexcept;
```

The default constructor (line 1) creates a `std::stop_source` with an associated stop state. The constructor taking `std::nostopstate_t` (line 2) constructs an empty `std::stop_source` without an associated stop state.

The component `std::stop_source src` provides the following member functions for handling stop requests.

Member functions of `std::stop_source src`

Member function	Description
`std::stop_source src`	Creates a stop source with an associated stop state.
`std::stop_source(std::nostopstate_t)`	Creates a stop source without an associated stop state.
`std::stop_source src{nostopstate}`	Creates a stop_source without associated stop state.
`src.get_token()`	If `src.stop_possible()`, returns a `stop_token` for the associated stop state. Otherwise, returns a default-constructed (empty) `stop_token` without associated stop state.
`src.stop_possible()`	`true` if `src` can be requested to stop.
`src.stop_requested()`	`true` if `stop_possible()` and `request_stop()` was called by one of the owners.
`src.request_stop()`	Calls a stop request if `src.stop_possible()` and `!src.stop_requested()`. Otherwise, the call has no effect.

The call `src.get_token()` returns the stop token `stoken`. Thanks to `stoken` you can check if a stop request has been made or can be made by its associated stop source `src`. The stop token `stoken` observes the stop source `src`.

`src.stop_requested()` returns `true` when `src` has an associated stop state and was not asked to stop earlier.

`src.stop_possible()` return `false` if there is no associated stop state or no stop source anymore and stop has never been requested before.

The calls `src.stop_possible()`, `src.stop_requested()`, and `src.request_stop()` are thread-safe.

`src.request_stop()` of a stop source `src` is visible to all `std::stop_token` and registered callback of the same associated stop state. Also, any `std::condiction_variable_any` waiting on the associated `std::stop_token()` will be awoken. Once a stop is requested, it cannot be withdrawn. `src.request_stop()` is successful and returns `true` if `src` has an associated stop state and was not requested to stop before.

6.5.2 `std::stop_token`

`std::stop_token` is essentially a thread-safe "view" of the associated stop state. It is typically retrieved from a `std::jthread` or a `std::stop_source src` via `src.get_token()`. This causes them to share the same associated stop state as the `std::jthread` or `std::stop_source`.

Thanks to the `std::stop_token`, you can check for the associated `std::stop_source` if a stop request has been made.

The `std::stop_token` can also be passed to the constructor of `std::stop_callback` or the interruptible waiting functions of `std::condition_variable_any`.

Member functions of `std::stop_token stoken`

Member function	Description
`std::stop_token stoken`	Creates a stop token with no associated stop state.
`stoken.stop_possible()`	Returns `true` if `stoken` has an associated stop state or a stop request has already been made, otherwise `false`.
`stoken.stop_requested()`	`true` if `request_stop()` was called on the associated `std::stop_source src`, otherwise `false`.

`stoken.stop_possible()` also returns `false` if there is no longer a stop source.

If the `std::stop_token` should be temporarily disabled, you can replace it with a default-constructed token. A default-constructed token has no associated stop state. The following code snippet shows how to disable and enable a thread's capability to accept stop requests.

Temporarily disable a stop token

```
std::jthread jthr([](std::stop_token stoken) {
    ...
    std::stop_token interruptDisabled;
    std::swap(stoken, interruptDisabled);
    ...
    std::swap(stoken, interruptDisabled);
    ...
}
```

`std::stop_token interruptDisabled` has no associated stop state. This means the thread `jthr` can accept stop requests in all lines except 4 and 5.

6.5.3 `std::stop_callback`

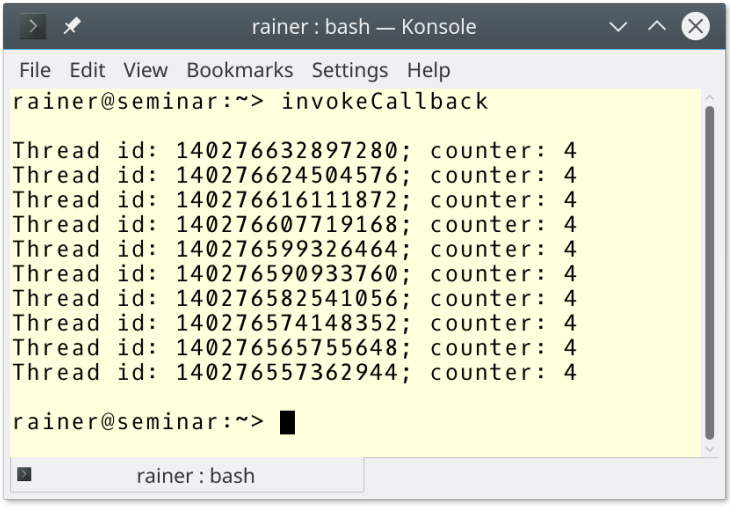
A `std::stop_callback` models RAII. It's constructor registers a callable for a stop token, and it's destructor unregisters it. The following example shows the use of `std::stop_callback`.

Use of callbacks

```
// invokeCallback.cpp

#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>

using namespace std::literals;

auto func = [](std::stop_token stoken) {
        int counter{0};
        auto thread_id = std::this_thread::get_id();
        std::stop_callback callBack(stoken, [&counter, thread_id] {
            std::cout << "Thread id: " << thread_id
                      << "; counter: " << counter << '\n';
        });
        while (counter < 10) {
            std::this_thread::sleep_for(0.2s);
            ++counter;
        }
    };

int main() {
```

```
25
26      std::cout << '\n';
27
28      std::vector<std::jthread> vecThreads(10);
29      for(auto& thr: vecThreads) thr = std::jthread(func);
30
31      std::this_thread::sleep_for(1s);
32
33      for(auto& thr: vecThreads) thr.request_stop();
34
35      std::cout << '\n';
36
37  }
```

Each ten threads invoke the lambda function `func` (lines 11 - 22). The callback in lines 14 - 17 displays the thread id and the local counter. Due to the 1-second sleeping of the main thread and the child threads sleeping, the counter is four when the callbacks are invoked. The call `thr.request_stop()` triggers the callback on each thread.



Use of callbacks

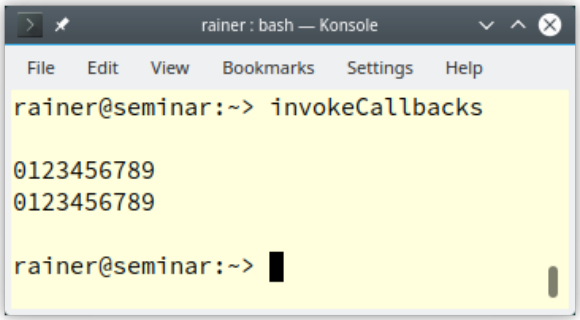
The `std::stop_callback` constructor registers the callback function for the `std::stop_token` given by the associated `std::stop_source`. This callback function is either invoked in the thread invoking `request_stop()` or the thread constructing the `std::stop_callback`. If the request to stop happens prior to the registration of the `std::stop_callback`, the callback is invoked in the thread constructing the `std::stop_callback`. Otherwise, the callback is invoked in the thread invoking `request_stop`. If the call `request_stop()` happens after the execution of the thread constructing the `std::stop_callback`,

the registered callback will never be called.

You can register more than one callback for one or more threads using the same `std::stop_token`. The C++ standard provides no guarantee in which order they are executed.

Use a `std::stop_token` more times on various threads

```
// invokeCallbacks.cpp

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

using namespace std::literals;

void func(std::stop_token stopToken) {
    std::this_thread::sleep_for(100ms);
    for (int i = 0; i <= 9; ++i) {
       std::stop_callback cb(stopToken, [i] { std::cout << i; });
    }
    std::cout << '\n';
}

int main() {

    std::cout << '\n';

    std::jthread thr1 = std::jthread(func);
    std::jthread thr2 = std::jthread(func);
    thr1.request_stop();
    thr2.request_stop();

    std::cout << '\n';

}
```



Use a `std::stop_token` more times on various threads

6.5.4 A General Mechanism to Send Signals

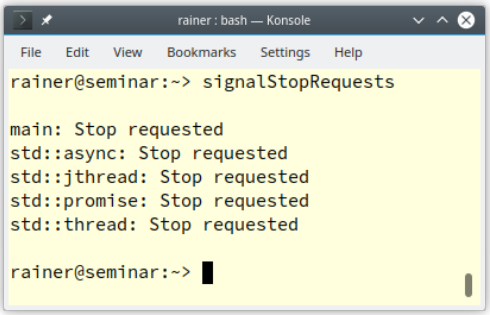
The pair `std::stop_source` and `std::stop_token` can be considered as a general mechanism to send a signal. By copying the `std::stop_token`, you can send the signal to any entity executing something. In the following example, I use `std::async`, `std::promise`, `std::thread`, and `std::jthread` in various combinations.

Sending a signal to various executing entities

```
1  // signalStopRequests.cpp
2
3  #include <iostream>
4  #include <thread>
5  #include <future>
6
7  using namespace std::literals;
8
9  void function1(std::stop_token stopToken, const std::string& str){
10     std::this_thread::sleep_for(1s);
11     if (stopToken.stop_requested()) std::cout << str << ": Stop requested\n";
12 }
13
14 void function2(std::promise<void> prom,
15                std::stop_token stopToken, const std::string& str) {
16     std::this_thread::sleep_for(1s);
17     std::stop_callback callBack(stopToken, [&str] {
18         std::cout << str << ": Stop requested\n";
19     });
20     prom.set_value();
21 }
22
```

```
23  int main() {
24
25      std::cout << '\n';
26
27      std::stop_source stopSource;
28
29      std::stop_token stopToken = std::stop_token(stopSource.get_token());
30
31      std::thread thr1 = std::thread(function1, stopToken, "std::thread");
32
33      std::jthread jthr = std::jthread(function1, stopToken, "std::jthread");
34
35      auto fut1 = std::async([stopToken] {
36          std::this_thread::sleep_for(1s);
37          if (stopToken.stop_requested()) std::cout << "std::async: Stop requested\n";
38      });
39
40      std::promise<void> prom;
41      auto fut2 = prom.get_future();
42      std::thread thr2(function2, std::move(prom), stopToken, "std::promise");
43
44      stopSource.request_stop();
45      if (stopToken.stop_requested()) std::cout << "main: Stop requested\n";
46
47      thr1.join();
48      thr2.join();
49
50      std::cout << '\n';
51
52  }
```

Thanks to the `stopSource` (line 27), I can create the `stopToken` (line 29) for each running entity such as `std::thread` (line 31), `std::jthread` (line 33), `std::async` (line 35), or `std::promise` (line 42). A `std::stop_token` is cheap to copy. Line 44 triggers `stopSource.request_stop`. Also, the `main`-thread (line 45) gets the signal. I use in this example `std::jthread`. `std::jthread` has explicit member functions to deal with cooperative interruption more conveniently. Read more about it in the following section Joining Thread.



Sending a signal to various executing entities

You may wonder why the various executing entities sleep for one second (lines 10, 16, and 36) in the previous program `signalStopRequests.cpp`? I want to be sure that the call `stopSource.request_stop()` in line 44 has an effect. The execution entity as the `std::thread` (line 31), the `std::jthread` (line 33), `std:async` (line 35), or `std::promise` (line 42) can have one of the following states, when the request to stop is signaled.

- Not started: The call `stopToken.stop_requested` returns `true` when executed. The callback is executed when `stopSource.request_stop` is signaled.
- Executing: The execution entity receives the signal. To take effect, the `stopSource.request_stop` must happen before the running entity calls `stopToken.stop_requested`. Accordingly, the `stopSource.request_stop` must happen before the callback is initialized.
- Finished: The call `stopSource.request_stop` has no effect. The callback is not executed.

Let's see what happens when I join the threads `thr1` and `thr2` before the call `stopSource.request_stop` in the previous program `signalStopRequests.cpp`? Here are lines 44 and 45 swapped with lines 47 and 48.

Sending the signal too late

```
44    thr1.join();
45    thr2.join();
46
47    stopSource.request_stop();
48    if (stopToken.stop_requested()) std::cout << "main: Stop requested\n";
```

The swap of the lines affects that only the `main`-thread reacts to the signal.

Ignoring the signal if it is too late

6.5.5 Joining Threads

A `std::jthread` is a `std::thread` with the additional functionality to signal an interrupt and to automatically `join()`. To support this functionality it has a `std::stop_token`.

The member functions of `std::jthread jthr` for stop-token handling

Member Function	Description
`t.get_stop_source()`	Returns a `std::stop_source` object associated with the shared stop state.
`t.get_stop_token()`	Returns a `std::stop_token` object associated with the shared stop state.
`t.request_stop()`	Requests execution stop via the shared stop state.

6.5.6 New `wait` Overloads for the `condition_variable_any`

`std::condition_variable_any` is a generalization of `std::condition_variable`[35]. `std::condition_variable` requires a `std::unique_lock<std::mutex>`, but `std::condition_variable_any` can operate on any lock `lo`, supporting `lo.lock()` and `lo.unlock`.

The three wait variations to `wait`, `wait_for`, and `wait_until` of the `std::condition_variable_any` get new overloads. They take a `std::stop_token`.

[35] https://en.cppreference.com/w/cpp/thread/condition_variable

Three new wait overloads

```
1  template <class Predicate>
2  bool wait(Lock& lock,
3            stop_token stoken,
4            Predicate pred);
5
6  template <class Rep, class Period, class Predicate>
7  bool wait_for(Lock& lock,
8                stop_token stoken,
9                const chrono::duration<Rep, Period>& rel_time,
10               Predicate pred);
11
12 template <class Clock, class Duration, class Predicate>
13 bool wait_until(Lock& lock,
14                 stop_token stoken,
15                 const chrono::time_point<Clock, Duration>& abs_time,
16                 Predicate pred);
```

These new overloads require a predicate. The presented versions ensure that the threads are notified if a stop request for the passed `std::stop_token` `stoken` is signaled. The functions return a boolean that indicates whether the predicate evaluates to `true`. Returning `false` means that the stop was requested or, if applicable, the timeout was triggered. The three overloads are equivalent to the following expressions:

Equivalent expression for the three overloads

```
// wait in lines 1 - 4
while (!stoken.stop_requested()) {
    if (pred()) return true;
    wait(lock);
}
return pred();

// wait_for in lines 6 - 10
return wait_until(lock,
                  std::move(stoken),
                  chrono::steady_clock::now() + rel_time,
                  std::move(pred)
                  );

// wait_until in lines 12 - 16
while (!stoken.stop_requested()) {
    if (pred()) return true;
    if (wait_until(lock, timeout_time) == std::cv_status::timeout) return pred();
```

```
}
return pred();
```

After the wait calls, you can check if a stop request happened.

Handle interrupts with `wait`

```
cv.wait(lock, stoken, predicate);
if (stoken.stop_requested()){
    // interrupt occurred
}
```

The following example shows the use of a condition variable with a stop request.

Use of condition variable with a stop request

```
// conditionVariableAny.cpp

#include <condition_variable>
#include <thread>
#include <iostream>
#include <chrono>
#include <mutex>
#include <thread>

using namespace std::literals;

std::mutex mut;
std::condition_variable_any condVar;

bool dataReady;

void receiver(std::stop_token stopToken) {

    std::cout << "Waiting" << '\n';

    std::unique_lock<std::mutex> lck(mut);
    bool ret = condVar.wait(lck, stopToken, []{return dataReady;});
    if (ret){
        std::cout << "Notification received: " << '\n';
    }
    else{
         std::cout << "Stop request received" << '\n';
    }
}
```

```
30
31  void sender() {
32
33      std::this_thread::sleep_for(5ms);
34      {
35          std::lock_guard<std::mutex> lck(mut);
36          dataReady = true;
37          std::cout << "Send notification"  << '\n';
38      }
39      condVar.notify_one();
40
41  }
42
43  int main(){
44
45      std::cout << '\n';
46
47      std::jthread t1(receiver);
48      std::jthread t2(sender);
49
50      t1.request_stop();
51
52      t1.join();
53      t2.join();
54
55      std::cout << '\n';
56
57  }
```

The `receiver` thread (lines 17 - 29) is waiting for the notification of the `sender` thread (lines 31 - 41). Before the `sender` thread sends its notification in line 39, the `main` thread triggers a stop request in line 50. The program's output shows that the stop request happened before the notification.

```
Waiting
Stop request received
Send notification
```

Sending a stop request to a condition variable

Distilled Information

- Thanks to `std::stop_source`, `std::stop_token`, and `std::stop_callback`, threads and condition variables can be cooperatively interrupted. Cooperative interruption means that the thread gets a stop request that it can accept or ignore.
- The `std::stop_token` can be passed to an operation and than used to actively poll the token for a stop request or register a callback via `std::stop_callback`.
- The pair `std::stop_source` and `std::stop_token` can be considered as a general mechanism to send a signal.
- Additionally to a `std::jthread`, `std::condition_variable_any` can also accept a stop request.

6.6 `std::jthread`

Cippi ties a braid

`std::jthread` stands for joining thread. In addition to `std::thread`[36] from C++11, `std::jthread` automatically joins in its destructor and can cooperatively be interrupted. `std::jthread` models RAII and, therefore, also joins when an exception occurs.

The `std::jthread` constructor creates a `std::stop_source` and stores it as a member of the thread object. It passes the corresponding `std::stop_token` to the called funtion if that functions takes an additional `std::stop_token` as first parameter. `The` std::stop_token‘ can be used by the function to check if a stop request has been made.

The following table gives you a concise overview of the `std::jthread t` functionality. For additional details, please refer to cppreference.com[37].

[36] https://en.cppreference.com/w/cpp/thread/thread
[37] https://en.cppreference.com/w/cpp/thread/jthread

Functions of a `std::jthread t`

Method	Description
`t.~jthread()`	If `joinable()`, calls `request_stop()` and then `join()`.
`t.join()`	Waits until thread `t` has finished its execution.
`t.detach()`	Executes the created thread `t` independently of the creator.
`t.joinable()`	Returns true if thread `t` is still joinable.
`t.get_id()` and `std::this_thread::get_id()`	Returns the id of the thread.
`std::jthread::hardware_concurrency()`	Indicates the number of threads that can run concurrently.
`std::this_thread::sleep_until(absTime)`	Puts thread `t` to sleep until time point `absTime`.
`std::this_thread::sleep_for(relTime)`	Puts thread `t` to sleep for time duration `relTime`.
`std::this_thread::yield()`	Enables the system to run another thread.
`t.swap(t2)`	Swaps the threads. Same as `std::swap(t, t2)`.
`t.get_stop_source()`	Returns a `std::stop_source` object associated with the shared stop state.
`t.get_stop_token()`	Returns a `std::stop_token` object associated with the shared stop state.
`t.request_stop()`	Requests execution stop via the shared stop state. Returns `true` if the stop request was successful.

Detaching a `std::jthread t` with `t.detach()` still allows it to call the functions `t.get_stop_source()` and `t.get_stop_token()`.

6.6.1 Automatically Joining

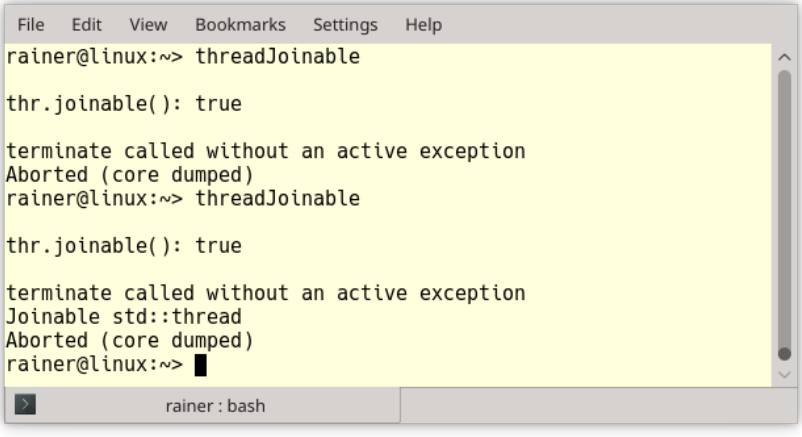
This is the *non-intuitive* behavior of `std::thread`. If a `std::thread` is still joinable, `std::terminate`[38] is called in its destructor, , which calls `std::abort`[39]. A thread `thr` is joinable if neither `thr.join()` nor `thr.detach()` has been called.

[38] https://en.cppreference.com/w/cpp/error/terminate
[39] https://en.cppreference.com/w/cpp/utility/program/abort

Terminating a still joinable `std::thread`

```
// threadJoinable.cpp

#include <iostream>
#include <thread>

int main() {

    std::cout << '\n';
    std::cout << std::boolalpha;

    std::thread thr{[]{ std::cout << "Joinable std::thread" << '\n'; }};

    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';

}
```

When executed, the program terminates.



Terminating a joinable `std::thread`

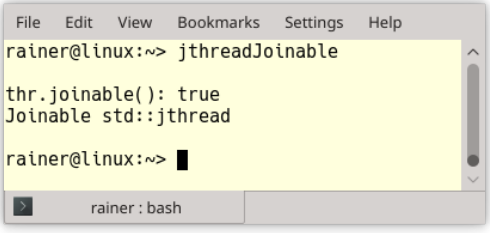
Both executions of `std::thread` terminate. In the second run, the thread `thr` has enough time to display its message: "Joinable std::thread".

In the next example, I use `std::jthread` from the current C++20 standard.

Terminating a still joinable `std::jthread`

```
// jthreadJoinable.cpp

#include <iostream>
#include <thread>

int main() {

    std::cout << '\n';
    std::cout << std::boolalpha;

    std::jthread thr{[]{ std::cout << "Joinable std::thread" << '\n'; }};

    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';

}
```

Now, the thread `thr` automatically joins in its destructor if it's still joinable.



Using a `std::jthread` that joins automatically

Here is a typical implementation of `std::jthreads` destructor.

Typical implemenation of `std::jthreads` destructor

```
1 jthread::~jthread() {
2     if(joinable()) {
3         request_stop();
4         join();
5     }
6 }
```

First, the thread checks if it is still joinable (line 2). A thread is still joinable if neither `join()` or `detach()` was called on it. If the thread is still joinable, it asks for the stopping of the execution (line 3) and calls `join()` afterward (line 4). The join call blocks until the execution of the thread is done.

6.6.2 Cooperative Interruption of a `std::jthread`

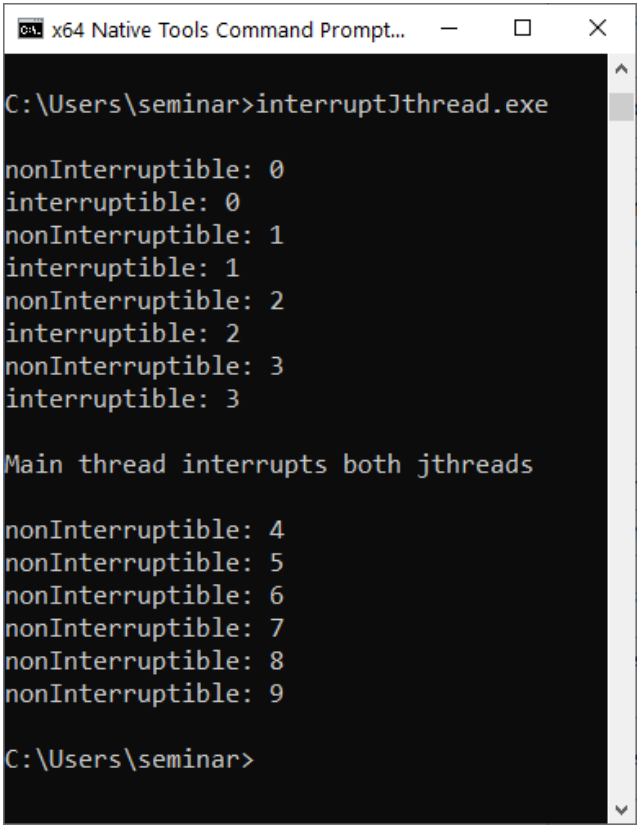
To get the general idea, let me present a simple example.

Interrupt a non-interruptible and interruptible `std::jthread`

```
// interruptJthread.cpp

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

using namespace::std::literals;

int main() {

    std::cout << '\n';

    std::jthread nonInterruptible([]{
        int counter{0};
        while (counter < 10){
            std::this_thread::sleep_for(0.2s);
            std::cerr << "nonInterruptible: " << counter << '\n';
            ++counter;
        }
    });

    std::jthread interruptible([](std::stop_token stoken){
        int counter{0};
        while (counter < 10){
            std::this_thread::sleep_for(0.2s);
            if (stoken.stop_requested()) return;
            std::cerr << "interruptible: " << counter << '\n';
            ++counter;
        }
    });

    std::this_thread::sleep_for(1s);

    std::cerr << '\n';
    std::cerr << "Main thread interrupts both jthreads" << '\n';
    nonInterruptible.request_stop();
    interruptible.request_stop();

    std::cout << '\n';

```

```
41 }
```

In the main program, I start the two threads `nonInterruptible` and `interruptible` (lines 13 and 22). Unlike in the thread `nonInterruptible`, the thread `interruptible` gets a `std::stop_token` and uses it in line 26 to check if it was interrupted: `stoken.stop_requested()`. In case of a stop request, the lambda function returns, and, therefore, the thread ends. The call `interruptible.request_stop()` (line 37) triggers the stop request. This does not hold for the previous call `nonInterruptible.request_stop()`. The call has no effect.



Interrupt a non-interruptible and interruptible `std::jthread`

Distilled Information

- A `std::jthread` stands for joining thread. In addition to `std::thread` from C++11, `std::jthread` automatically joins in its destructor and can cooperatively be interrupted.
- This is the non-intuitive behavior of `std::thread`. If a `std::thread` is still joinable, `std::terminate` is called in its destructor, which calls `std::abort`. In contrast, a `std::jthread` automatically joins in its destructor if necessary.
- A `std::jthread` can cooperatively be interrupted using a `std::stop_token`. Cooperatively means that the `std::jthread` can ignore the stop request.

6.7 Synchronized Output Streams

Cippi sings in the choir

Compiler Support for Synchronized Output Streams

At the end of 2020, only GCC 11 supports synchronized output streams.

What happens when you write without synchronization to `std::cout`?

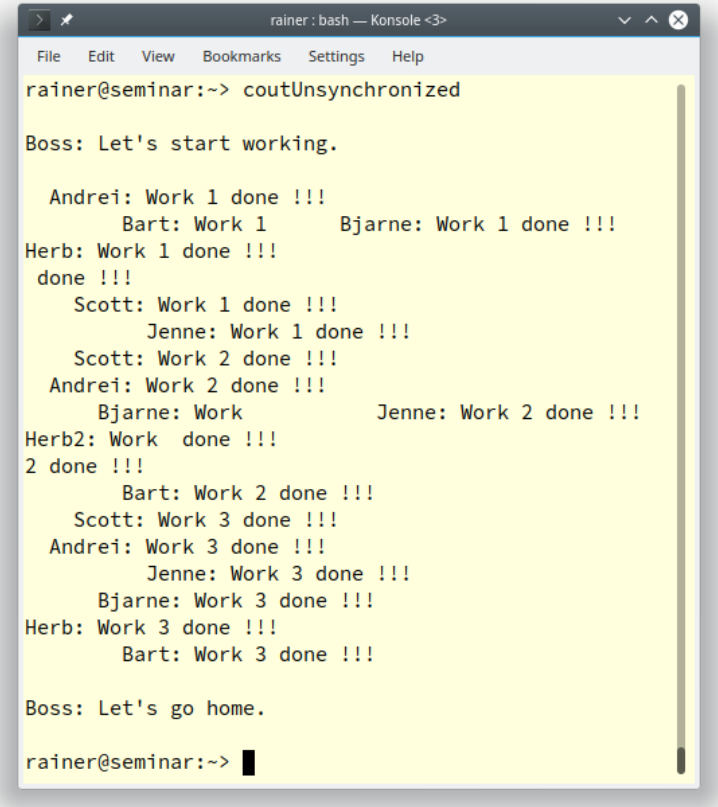
Non-synchronized access to `std::cout`

```
// coutUnsynchronized.cpp

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

class Worker{
public:
  Worker(std::string n):name(n) {};
    void operator() (){
      for (int i = 1; i <= 3; ++i) {
        // begin work
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        // end work
        std::cout << name << ": " << "Work " << i << " done !!!" << '\n';
      }
    }
private:
```

```
19    std::string name;
20  };
21
22
23  int main() {
24
25    std::cout << '\n';
26
27    std::cout << "Boss: Let's start working.\n\n";
28
29    std::thread herb= std::thread(Worker("Herb"));
30    std::thread andrei= std::thread(Worker("  Andrei"));
31    std::thread scott= std::thread(Worker("    Scott"));
32    std::thread bjarne= std::thread(Worker("      Bjarne"));
33    std::thread bart= std::thread(Worker("        Bart"));
34    std::thread jenne= std::thread(Worker("          Jenne"));
35
36
37    herb.join();
38    andrei.join();
39    scott.join();
40    bjarne.join();
41    bart.join();
42    jenne.join();
43
44    std::cout << "\n" << "Boss: Let's go home." << '\n';
45
46    std::cout << '\n';
47
48  }
```

The boss has six workers (lines 29 - 34). Each worker has to take care of three work packages that take 1/5 second each (line 13). After the worker is done with his work package, he screams out loudly to the boss (line 15). Once the boss receives notifications from all workers, he sends them home (line 44).

What a mess for such a simple workflow! Each worker screams out his message ignoring their coworkers!



Non-synchronized writing to `std::cout`

`std::cout` is thread-safe

The C++11 standard guarantees that you need not protect `std::cout`. Each character is written atomically. More output statements like those in the example may interleave. This interleaving is only a visual issue; the program is well-defined. This remark is valid for all global stream objects. Insertion to and extraction from global stream objects (`std::cout`, `std::cin`, `std::cerr`, and `std::clog`) is thread safe. To put it more formally: writing to `std::cout` is not participating in a data race, but does create a race condition. This means that the output depends on the interleaving of threads.

How can we solve this issue? With C++11, the answer is straightforward: use a lock such as `lock_-`

guard[40] to synchronize the access to std::cout.

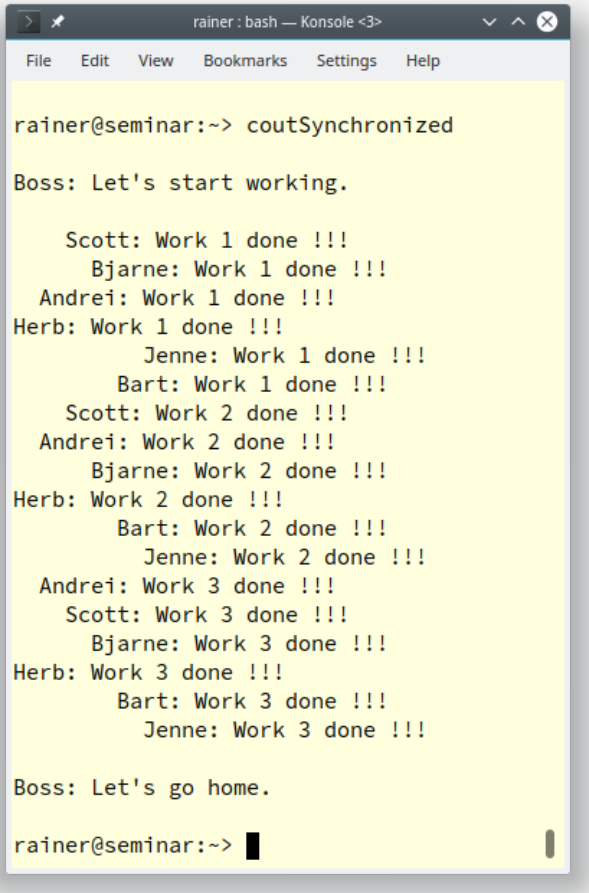
Synchronized access to std::cout

```
// coutSynchronized.cpp

#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex coutMutex;

class Worker{
public:
  Worker(std::string n):name(n) {};

    void operator() () {
      for (int i = 1; i <= 3; ++i) {
        // begin work
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        // end work
        std::lock_guard<std::mutex> coutLock(coutMutex);
        std::cout << name << ": " << "Work " << i << " done !!!\n";
      }
    }
private:
  std::string name;
};


int main() {

  std::cout << '\n';

  std::cout << "Boss: Let's start working." << "\n\n";

  std::thread herb= std::thread(Worker("Herb"));
  std::thread andrei= std::thread(Worker("  Andrei"));
  std::thread scott= std::thread(Worker("    Scott"));
  std::thread bjarne= std::thread(Worker("      Bjarne"));
  std::thread bart= std::thread(Worker("        Bart"));
  std::thread jenne= std::thread(Worker("          Jenne"));

```

[40] https://en.cppreference.com/w/cpp/thread/lock_guard

```
41    herb.join();
42    andrei.join();
43    scott.join();
44    bjarne.join();
45    bart.join();
46    jenne.join();
47
48    std::cout << "\n" << "Boss: Let's go home." << '\n';
49
50    std::cout << '\n';
51
52  }
```

The `coutMutex` in line 8 protects the shared object `std::cout`. Putting the `coutMutex` into a `std::lock_guard` guarantees that the `coutMutex` is locked in the constructor (line 19) and unlocked in the destructor (line 21) of the `std::lock_guard`. Thanks to the `coutMutex`, guarded by the `coutLock`, the chaos becomes a harmony.



Synchronized access of `std::cout`

With C++20, writing synchronized to `std::cout` is a piece of cake. `std::basic_syncbuf` is a wrapper for a `std::basic_streambuf`[41]. It accumulates output in its buffer. The wrapper sets its content to the wrapped buffer when it is destructed. Consequently, the content appears as a contiguous sequence of characters, and no interleaving of characters can happen.

Thanks to `std::basic_osyncstream`, you can directly write synchronously to `std::cout`.

C++20 defines two specializations of `std::basic_osyncstream` for `char` and `wchar_t`.

[41] https://en.cppreference.com/w/cpp/io/basic_streambuf

```
std::osyncstream          std::basic_osyncstream<char>
std::wosyncstream          std::basic_osyncstream<wchar_t>
```

You can create a named-synchronized output stream. Now, the previous program `coutUnsynchronized.cpp` is refactored to write synchronized to `std::cout`.

Synchronized access of `std::cout` with `std::basic_osyncstream`

```
// synchronizedOutput.cpp

#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>

class Worker{
public:
  Worker(std::string n): name(n) {};
    void operator() (){
      for (int i = 1; i <= 3; ++i) {
        // begin work
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        // end work
        std::osyncstream syncStream(std::cout);
        syncStream << name << ": " << "Work " << i << " done !!!" << '\n';
      }
    }
private:
  std::string name;
};


int main() {

  std::cout << '\n';

  std::cout << "Boss: Let's start working.\n\n";

  std::thread herb= std::thread(Worker("Herb"));
  std::thread andrei= std::thread(Worker("  Andrei"));
  std::thread scott= std::thread(Worker("    Scott"));
  std::thread bjarne= std::thread(Worker("      Bjarne"));
  std::thread bart= std::thread(Worker("        Bart"));
  std::thread jenne= std::thread(Worker("          Jenne"));


```

```
39    herb.join();
40    andrei.join();
41    scott.join();
42    bjarne.join();
43    bart.join();
44    jenne.join();
45
46    std::cout << "\n" << "Boss: Let's go home." << '\n';
47
48    std::cout << '\n';
49
50  }
```

The only change to the previous program `coutUnsynchronized.cpp` is that `std::cout` is wrapped in a `std::osyncstream` (line 16). To use the `std::osyncstream`, I add the header `<syncstream>`. When the `std::osyncstream` goes out of scope in line 18, the characters are transferred, and `std::cout` is flushed. It is worth mentioning that the `std::cout` calls in the main program do not introduce a data race and, therefore, need not be synchronized.

Because I use the `syncStream` declared on line 17 only once, a temporary object may be more appropriate. The following code snippet presents the modified call operator.

```
void operator()() {
  for (int i = 1; i <= 3; ++i) {
    // begin work
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // end work
    std::osyncstream(std::cout) << name << ": " << "Work " << i << " done !!!"
                                << '\n';
  }
}
```

`std::basic_osyncstream syncStream` offers two interesting member functions.

- `syncStream.emit()` emits all buffered output and executes all pending flushes.
- `syncStream.get_wrapped()` returns a pointer to the wrapped buffer.

Additionally, the flag `std::flush_emit` allows you to flush the buffer of the synchronized output stream explicitly.

```
1  void operator()() {
2      std::osyncstream syncStream(std::cout);
3      for (int i = 1; i <= 3; ++i) {
4          // begin work
5          std::this_thread::sleep_for(std::chrono::milliseconds(200));
6          // end work
7          syncStream << name << ": " << "Work " << i << " done !!!"
8                     << '\n' << std::flush_emit;
9      }
10 }
```

The modified call operator produces the same output as the previous one. This time, the synchronized output stream is an lvalue and flushes its content explicitly (line 8).

cppreference.com[42] shows how you can sequence the output of different output streams with the `get_wrapped` member function.

Sequence output

```
// sequenceOutput.cpp

#include <syncstream>
#include <iostream>
int main() {

  std::osyncstream bout1(std::cout);
  bout1 << "Hello, ";
  {
      std::osyncstream(bout1.get_wrapped()) << "Goodbye, " << "Planet!" << '\n';
  } // emits the contents of the temporary buffer

  bout1 << "World!" << '\n';

} // emits the contents of bout1
```

```
Goodbye, Planet!
Hello, World!
```

Synchronized access of `std::cout`

[42] https://en.cppreference.com/w/cpp/io/basic_osyncstream/get_wrapped

Distilled Information

- Although `std::cout` is thread safe, you may get an interleaving of output operations when threads concurrently write to `std::cout`. This is only a visual issue but not a data race.
- C++20 supports synchronized output streams. They accumulate output in an internal buffer and write their content in an atomic step. Consequently, no interleaving of output operations happens.

7. Case Studies

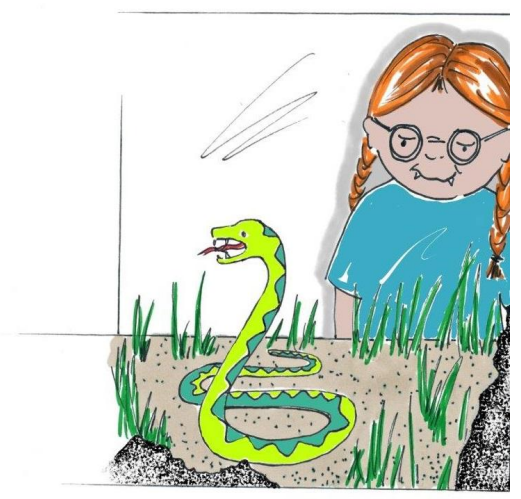
After providing the theory to C++20, I now apply the theory in practice and provide you with a few case studies.

The ranges library are great tool to implement more convenient functions. In the section A Flavor of Python, I start an experiment and implement Python's `2 filter`, `map`, and `list comprehension` functions using the ranges library.

The section on coroutines presented three coroutines, based on `co_return`, `co_yield`, and `co_await`. I use these coroutines as a starting point for further experiments to deepen our understanding of the challenging control-flow of coroutines. In section variations of futures, I implement a lazy future and a future based on the future in section `co_return`. Section modification and generalization of threads improves the generator from section `co_return`, and, finally, section various job workflows discusses the job workflow, started in the section about `co_await`.

When you want to synchronize threads more than once, you can use condition variables, `std::atomic_flag`, `std::atomic<bool>`, or semaphores. In the section fast synchronization of threads, I want to answer which variant is the fastest?

7.1 A Flavor of Python



Cippi starts the workflow

The programming language Python[1] has the convenient functions `filter` and `map`.

- **filter**: applies a predicate to all elements of an iterable and returns those elements for which the predicate returns `true`
- **map**: applies a function to all elements of an iterable and returns a new iterable with the transformed elements

An iterable in C++ would be a type that you could use in a range-based for loop.

Furthermore, Python lets you combine both functions in a list comprehension.

- **list comprehension**: applies a filter and map phase to an iterable and returns a new iterable

My challenge is: I want to implement Python2-like functions `filter`, `map`, and list comprehension in C++20 using the ranges library.

7.1.1 `filter`

Python's `filter` function has a corresponding ranges function.

[1] https://www.python.org/

Python's filter function in C++

```cpp
// filterRanges.cpp

#include <iostream>
#include <numeric>
#include <ranges>
#include <string>
#include <vector>

template <typename Func, typename Seq>
auto filter(Func func, const Seq& seq) {

    typedef typename Seq::value_type value_type;

    std::vector<value_type> result{};
    for (auto i : seq | std::views::filter(func)) result.push_back(i);

    return result;
}


int main() {

    std::cout << '\n';

    std::vector<int> myInts(50);
    std::iota(myInts.begin(), myInts.end(), 1);
    auto res = filter([](int i){ return (i % 3) == 0; }, myInts);
    for (auto v: res) std::cout << v << " ";


    std::vector<std::string> myStrings{"Only", "for", "testing", "purposes"};
    auto res2 = filter([](const std::string& s){ return std::isupper(s[0]); },
                       myStrings);

    std::cout << "\n\n";

    for (auto word: res2) std::cout << word << '\n';

    std::cout << '\n';

}
```

Before I write a few words about the program, let me show you the output.

```
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48

Only
```

The filter function applied

The `filter` function (line 9) should be easy to read. Line 12 detects the type of the underlying element. I simply apply the callable `func` to each element of the sequence and return the elements in the `std::vector`. Line 27 selects all numbers `i` from 1 to 50 for which `(i % 3) == 0` holds. Only the strings that start with an uppercase letter can pass the filter in line 32.

7.1.2 map

`map` applies a callable to each element of the input sequence.

Python's map function in C++

```
1  // mapRanges.cpp
2
3  #include <iostream>
4  #include <list>
5  #include <ranges>
6  #include <string>
7  #include <vector>
8  #include <utility>
9
10
11 template <typename Func, typename Seq>
12 auto map(Func func, const Seq& seq) {
13
14     typedef typename Seq::value_type value_type;
15     using return_type = decltype(func(std::declval<value_type>()));
16
17     std::vector<return_type> result{};
18     for (auto i :seq | std::views::transform(func)) result.push_back(i);
19
20     return result;
21 }
22
23 int main() {
24
25     std::cout << '\n';
26
27     std::list<int> myInts{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
28     auto res = map([](int i){ return i * i; }, myInts);
```

```
29
30      for (auto v: res) std::cout << v << " ";
31
32      std::cout << "\n\n";
33
34      std::vector<std::string> myStrings{"Only", "for", "testing", "purposes"};
35      auto res2 = map([](const std::string& s){ return std::make_pair(s.size(), s); },
36                                                                      myStrings);
37
38      for (auto p: res2) std::cout << "(" <<  p.first << ", " << p.second << ") " ;
39
40      std::cout << "\n\n";
41
42  }
```

Line 15 in the definition of the `map` function is quite interesting. The expression `decltype(func(std::declval<value_type>()))` deduces the `return_type`. The `return_type` is the type to which all input sequence elements are transformed if the function `func` is applied to them. `std::declval<value_type>()` returns an rvalue reference that `decltype` can use to deduce the type. That means the call `map([](int i){ return i * i; }, myInts)` (line 28) maps each element of `myInt` to its square, and the call `map([](const std::string& s){ return std::make_pair(s.size(), s); }, myStrings)` maps each string of `myStrings` to a pair. The first element of each pair is the length of the string.

```
1 4 9 16 25 36 49 64 81 100

(4, Only) (3, for) (7, testing) (8, purposes)
```

The `map` function applied

7.1.3 List Comprehension

The program `listComprehensionRanges.cpp` has a simplified version of Python's list-comprehension algorithm.

`map` applies a callable to each element of the input sequence.

A simplified variant of Python's list comprehension in C++

```cpp
// listComprehensionRanges.cpp

#include <algorithm>
#include <cctype>
#include <functional>
#include <iostream>
#include <ranges>
#include <string>
#include <vector>
#include <utility>

template <typename T>
struct AlwaysTrue {
    constexpr bool operator()(const T&) const {
        return true;
    }
};

template <typename Map, typename Seq, typename Filt = AlwaysTrue<
                                                    typename Seq::value_type>>
auto mapFilter(Map map, Seq seq, Filt filt = Filt()) {

    typedef typename Seq::value_type value_type;
    using return_type = decltype(map(std::declval<value_type>()));

    std::vector<return_type> result{};
    for (auto i :seq | std::views::filter(filt)
                     | std::views::transform(map)) result.push_back(i);
    return result;
}

int main() {

    std::cout << '\n';

    std::vector myInts{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    auto res = mapFilter([](int i){ return i * i; }, myInts);
    for (auto v: res) std::cout << v << " ";

    std::cout << "\n\n";

    res = mapFilter([](int i){ return i * i; }, myInts,
                    [](auto i){ return i % 2 == 1; });
```

```
45      for (auto v: res) std::cout << v << " ";
46
47      std::cout << "\n\n";
48
49      std::vector<std::string> myStrings{"Only", "for", "testing", "purposes"};
50      auto res2 = mapFilter([](const std::string& s){
51                              return std::make_pair(s.size(), s);
52                            }, myStrings);
53      for (auto p: res2) std::cout << "(" <<  p.first << ", " << p.second << ") " ;
54
55      std::cout << "\n\n";
56
57      myStrings = {"Only", "for", "testing", "purposes"};
58      res2 = mapFilter([](const std::string& s){
59                           return std::make_pair(s.size(), s);
60                       }, myStrings,
61                       [](const std::string& word){ return std::isupper(word[0]); });
62
63      for (auto p: res2) std::cout << "(" <<  p.first << ", " << p.second << ") " ;
64
65      std::cout << "\n\n";
66
67  }
```

The default predicate that the filter function applies (line 19) always returns `true` (line 12). Always `true` means that the function `mapFilter` simply behaves by default as a `map` function. Consequently, the `mapFilter` function behaves in lines 37 and 49 as does the previous `map` function. Lines 42 and 55 apply both functions `map` and `filter` in one call.

```
1 4 9 16 25 36 49 64 81 100

1 9 25 49 81

(4, Only) (3, for) (7, testing) (8, purposes)

(4, Only)
```

Both functions `map` and `filter` applied

7.2 Variations of Futures

Cippi starts the workflow

Before I create variations of the future from section co_return, we should understand its control flow. Comments make the control flow transparent. Additionally, I provide a link to the presented programs on online compilers.

Control flow of an eager future

```
// eagerFutureWithComments.cpp

#include <coroutine>
#include <iostream>
#include <memory>

template<typename T>
struct MyFuture {
    std::shared_ptr<T> value;
    MyFuture(std::shared_ptr<T> p): value(p) {
        std::cout << "    MyFuture::MyFuture" << '\n';
    }
    ~MyFuture() {
        std::cout << "    MyFuture::~MyFuture" << '\n';
    }
    T get() {
        std::cout << "    MyFuture::get" << '\n';
        return *value;
    }

    struct promise_type {
        std::shared_ptr<T> ptr = std::make_shared<T>();
        promise_type() {
            std::cout << "        promise_type::promise_type" << '\n';
```

```
25          }
26          ~promise_type() {
27              std::cout << "        promise_type::~promise_type" << '\n';
28          }
29          MyFuture<T> get_return_object() {
30              std::cout << "        promise_type::get_return_object" << '\n';
31              return ptr;
32          }
33          void return_value(T v) {
34              std::cout << "        promise_type::return_value" << '\n';
35              *ptr = v;
36          }
37          std::suspend_never initial_suspend() {
38              std::cout << "        promise_type::initial_suspend" << '\n';
39              return {};
40          }
41          std::suspend_never final_suspend() noexcept {
42              std::cout << "        promise_type::final_suspend" << '\n';
43              return {};
44          }
45          void unhandled_exception() {
46              std::exit(1);
47          }
48      };
49  };
50
51  MyFuture<int> createFuture() {
52      std::cout << "createFuture" << '\n';
53      co_return 2021;
54  }
55
56  int main() {
57
58      std::cout << '\n';
59
60      auto fut = createFuture();
61      auto res = fut.get();
62      std::cout << "res: " << res << '\n';
63
64      std::cout << '\n';
65
66  }
```

The call `createFuture` (line 60) causes the creation of the instance of `MyFuture` (line 59). Before

`MyFuture`'s constructor call (line 10) is completed, the promise `promise_type` is created, executed, and destroyed (lines 20 - 48). The promise uses in each step of its control flow the awaitable `std::suspend_never` (lines 36 and 40) and, hence, never pauses. To save the result of the promise for the later `fut.get()` call (line 60), it has to be allocated. Furthermore, the used `std::shared_ptrs` ensure (lines 9 and 21) that the program does not cause a memory leak. As a local, `fut` goes out of scope in line 65, and the C++ run time calls its destructor.

You can try out the program on the Compiler Explorer[2].

```
        promise_type::promise_type
        promise_type::get_return_object
        promise_type::initial_suspend
createFuture
        promise_type::return_value
        promise_type::final_suspend
        promise_type::~promise_type
    MyFuture::MyFuture
    MyFuture::get
res: 2021

    MyFuture::~MyFuture
```

An eager future

The presented coroutine runs immediately and is, therefore, eager. Furthermore, the coroutine runs in the thread of the caller.

Let's make the coroutine lazy.

7.2.1 A Lazy Future

A lazy future is a future that only runs when asked for the value. Let's see what I have to change in the eager coroutine, presented in `eagerFutureWithComments.cpp`, to make it lazy.

Control flow of a lazy future

```
1  // lazyFuture.cpp
2
3  #include <coroutine>
4  #include <iostream>
5  #include <memory>
6
7  template<typename T>
8  struct MyFuture {
9      struct promise_type;
```

[2] https://godbolt.org/z/Y9naEx

```
    using handle_type = std::coroutine_handle<promise_type>;

    handle_type coro;

    MyFuture(handle_type h): coro(h) {
        std::cout << "    MyFuture::MyFuture" << '\n';
    }
    ~MyFuture() {
        std::cout << "    MyFuture::~MyFuture" << '\n';
        if ( coro ) coro.destroy();
    }

    T get() {
        std::cout << "    MyFuture::get" << '\n';
        coro.resume();
        return coro.promise().result;
    }

    struct promise_type {
        T result;
        promise_type() {
            std::cout << "        promise_type::promise_type" << '\n';
        }
        ~promise_type() {
            std::cout << "        promise_type::~promise_type" << '\n';
        }
        auto get_return_object() {
            std::cout << "        promise_type::get_return_object" << '\n';
            return MyFuture{handle_type::from_promise(*this)};
        }
        void return_value(T v) {
            std::cout << "        promise_type::return_value" << '\n';
            result = v;
        }
        std::suspend_always initial_suspend() {
            std::cout << "        promise_type::initial_suspend" << '\n';
            return {};
        }
        std::suspend_always final_suspend() noexcept {
            std::cout << "        promise_type::final_suspend" << '\n';
            return {};
        }
        void unhandled_exception() {
            std::exit(1);
        }
```

```
55      };
56  };
57
58  MyFuture<int> createFuture() {
59      std::cout << "createFuture" << '\n';
60      co_return 2021;
61  }
62
63  int main() {
64
65      std::cout << '\n';
66
67      auto fut = createFuture();
68      auto res = fut.get();
69      std::cout << "res: " << res << '\n';
70
71      std::cout << '\n';
72
73  }
```

Let's first study the promise. The promise always suspends at the beginning (line 44) and at the end (line 48). Furthermore, the member function `get_return_object` (line 36) creates the return object that is returned to the caller of the coroutine `createFuture` (line 58). The future `MyFuture` is more interesting. It has a handle `coro` (line 12) to the promise. `MyFuture` uses the handle to manage the promise. It resumes the promise (line 24), asks the promise for the result (line 25), and finally destroys it (line 19). The resumption of the coroutine is necessary because it never runs automatically (line 44). When the client invokes `fut.get()` (line 68) to ask for the result of the future, it implicitly resumes the promise (line 24).

You can try out the program on the Compiler Explorer[3].

[3] https://godbolt.org/z/EejWcj

```
        promise_type::promise_type
        promise_type::get_return_object
    MyFuture::MyFuture
        promise_type::initial_suspend
    MyFuture::get
createFuture
        promise_type::return_value
        promise_type::final_suspend
res: 2021

    MyFuture::~MyFuture
        promise_type::~promise_type
```

A lazy future

What happens if the client is not interested in the result of the future? Let's try it out.

The client does not resume the coroutine

```
int main() {

    std::cout << '\n';

    auto fut = createFuture();
    // auto res = fut.get();
    // std::cout << "res: " << res << '\n';

    std::cout << '\n';

}
```

As you may guess, the promise never runs, and the member functions `return_value` and `final_suspend` are not executed.

```
    promise_type::promise_type
    promise_type::get_return_object
MyFuture::MyFuture
    promise_type::initial_suspend

MyFuture::~MyFuture
    promise_type::~promise_type
```

A lazy future that is not started

Lifetime Challenges of Coroutines

One of the challenges of dealing with coroutines is handling the lifetime of the coroutine. In the previous program `eagerFutureWithComments.cpp`, I stored the coroutine result in a `std::shared_ptr`. This is critical because the coroutine is executed eagerly.

In this program `lazyFuture.cpp`, the call `final_suspend` always suspends (line 48): `std::suspend_always final_suspend()`. Consequently, the promise outlives the client, and a `std::shared_ptr` is not necessary anymore. Returning `std::suspend_never` from the function `final_suspend` would cause, in this case, undefined behavior because the client would outlive the promise. Hence, the lifetime of the result ends, before the client asks for it.

Let's vary the coroutine further and run the promise in a separate thread.

7.2.2 Execution on Another Thread

The coroutine is fully suspended before entering the coroutine `createFuture` (line 67) because the member function `initial_suspend` returns `std::suspend_always` (line 52). Consequently, the promise can run on another thread.

Executing the promise on another thread

```
1  // lazyFutureOnOtherThread.cpp
2
3  #include <coroutine>
4  #include <iostream>
5  #include <memory>
6  #include <thread>
7
8  template<typename T>
9  struct MyFuture {
10     struct promise_type;
11     using handle_type = std::coroutine_handle<promise_type>;
12     handle_type coro;
13
14     MyFuture(handle_type h): coro(h) {}
15     ~MyFuture() {
16         if ( coro ) coro.destroy();
17     }
18
19     T get() {
20         std::cout << "    MyFuture::get:  "
21                   << "std::this_thread::get_id(): "
22                   << std::this_thread::get_id() << '\n';
23
```

```
24          std::thread t([this] { coro.resume(); });
25          t.join();
26          return coro.promise().result;
27      }
28
29      struct promise_type {
30          promise_type(){
31              std::cout << "        promise_type::promise_type:  "
32                        << "std::this_thread::get_id(): "
33                        << std::this_thread::get_id() << '\n';
34          }
35          ~promise_type(){
36              std::cout << "        promise_type::~promise_type:  "
37                        << "std::this_thread::get_id(): "
38                        << std::this_thread::get_id() << '\n';
39          }
40
41          T result;
42          auto get_return_object() {
43              return MyFuture{handle_type::from_promise(*this)};
44          }
45          void return_value(T v) {
46              std::cout << "        promise_type::return_value:  "
47                        << "std::this_thread::get_id(): "
48                        << std::this_thread::get_id() << '\n';
49              std::cout << v << std::endl;
50              result = v;
51          }
52          std::suspend_always initial_suspend() {
53              return {};
54          }
55          std::suspend_always final_suspend() noexcept {
56              std::cout << "        promise_type::final_suspend:  "
57                        << "std::this_thread::get_id(): "
58                        << std::this_thread::get_id() << '\n';
59              return {};
60          }
61          void unhandled_exception() {
62              std::exit(1);
63          }
64      };
65  };
66
67  MyFuture<int> createFuture() {
68      co_return 2021;
```

```
69  }
70
71  int main() {
72
73      std::cout << '\n';
74
75      std::cout << "main:  "
76                << "std::this_thread::get_id(): "
77                << std::this_thread::get_id() << '\n';
78
79      auto fut = createFuture();
80      auto res = fut.get();
81      std::cout << "res: " << res << '\n';
82
83      std::cout << '\n';
84
85  }
```

I added a few comments to the program that show the id of the running thread. The program `lazyFutureOnOtherThread.cpp` is quite similar to the previous program `lazyFuture.cpp`. The main difference is the member function `get` (line 19). The call `std::thread t([this] { coro.resume(); });` (line 24) resumes the coroutine on another thread.

You can try out the program on the Wandbox[4] online compiler.

```
main:  std::this_thread::get_id(): 139819561723776
       promise_type::promise_type:  std::this_thread::get_id(): 139819561723776
   MyFuture::get:  std::this_thread::get_id(): 139819561723776
       promise_type::return_value:  std::this_thread::get_id(): 139819456755456
       promise_type::final_suspend:  std::this_thread::get_id(): 139819456755456
res: 2021

       promise_type::~promise_type:  std::this_thread::get_id(): 139819561723776
```

Execution on another thread

I want to add a few additional remarks about the member function `get`. It is crucial that the promise, resumed in a separate thread and finishes before it returns `coro.promise().result`.

[4] https://wandbox.org/permlink/jFVVj80Gxu6bnNkc

The member function `get` using `std::thread`

```
T get() {
    std::thread t([this] { coro.resume(); });
    t.join();
    return coro.promise().result;
}
```

Where I join the thread `t` after the call `return coro.promise().result`, the program would have undefined behavior. In the following implementation of the function `get`, I use a `std::jthread`. Since `std::jthread` automatically joins when it goes out of scope. This is too late.

The member function `get` using `std::jthread`

```
T get() {
    std::jthread t([this] { coro.resume(); });
    return coro.promise().result;
}
```

In this case, the client likely gets its result before the promise prepares it using the member function `return_value`. Now, `result` has an arbitrary value, and therefore so does `res`.

```
main:  std::this_thread::get_id(): 139913381070720
       promise_type::promise_type:  std::this_thread::get_id(): 139913381070720
   MyFuture::get:  std::this_thread::get_id(): 139913381070720
       promise_type::return_value:  std::this_thread::get_id(): 139913276102400
       promise_type::final_suspend:  std::this_thread::get_id(): 139913276102400
res: -1

       promise_type::~promise_type:  std::this_thread::get_id(): 139913381070720
```

Execution on another thread

There are other possibilities to ensure that the thread is done before the return call.

- Create a `std::jthread` in its scope.

`std::jthread` has its own scope

```
T get() {
    {
        std::jthread t([this] { coro.resume(); });
    }
    return coro.promise().result;
}
```

- Make `std::jthread` a temporary object

`std::jthread` as a temporary

```
T get() {
    std::jthread([this] { coro.resume(); });
    return coro.promise().result;
}
```

In particular, I don't like the last solution because it may take you a few seconds to recognize that I just called the constructor of `std::jthread`.

7.3 Modification and Generalization of a Generator

Cippi handles a data stream

Before I modify and generalize the generator for an infinite data stream, I want to present it as a starting point of our journey. I intentionally put many output operations in the source code and only ask for three values. This simplification and visualization should help to understand the control flow.

Generator generating an infinite data stream

```
1  // infiniteDataStreamComments.cpp
2
3  #include <coroutine>
4  #include <memory>
5  #include <iostream>
6
7  template<typename T>
8  struct Generator {
9
10     struct promise_type;
11     using handle_type = std::coroutine_handle<promise_type>;
12
13     Generator(handle_type h): coro(h) {
14         std::cout << "        Generator::Generator" << '\n';
15     }
16     handle_type coro;
17
```

```
18      ~Generator() {
19          std::cout << "        Generator::~Generator" << '\n';
20          if ( coro ) coro.destroy();
21      }
22      Generator(const Generator&) = delete;
23      Generator& operator = (const Generator&) = delete;
24      Generator(Generator&& oth): coro(oth.coro) {
25          oth.coro = nullptr;
26      }
27      Generator& operator = (Generator&& oth) {
28          coro = oth.coro;
29          oth.coro = nullptr;
30          return *this;
31      }
32      int getNextValue() {
33          std::cout << "        Generator::getNextValue" << '\n';
34          coro.resume();
35          return coro.promise().current_value;
36      }
37      struct promise_type {
38          promise_type() {
39              std::cout << "            promise_type::promise_type" << '\n';
40          }
41
42          ~promise_type() {
43              std::cout << "            promise_type::~promise_type" << '\n';
44          }
45
46          std::suspend_always initial_suspend() {
47              std::cout << "            promise_type::initial_suspend" << '\n';
48              return {};
49          }
50          std::suspend_always final_suspend() noexcept {
51              std::cout << "            promise_type::final_suspend" << '\n';
52              return {};
53          }
54          auto get_return_object() {
55              std::cout << "            promise_type::get_return_object" << '\n';
56              return Generator{handle_type::from_promise(*this)};
57          }
58
59          std::suspend_always yield_value(int value) {
60              std::cout << "            promise_type::yield_value" << '\n';
61              current_value = value;
62              return {};
```

```
        }
        void return_void() {}
        void unhandled_exception() {
            std::exit(1);
        }

        T current_value;
    };

};

Generator<int> getNext(int start = 10, int step = 10) {
    std::cout << "    getNext: start" << '\n';
    auto value = start;
    while (true) {
        std::cout << "    getNext: before co_yield" << '\n';
        co_yield value;
        std::cout << "    getNext: after co_yield" << '\n';
        value += step;
    }
}

int main() {

    auto gen = getNext();
    for (int i = 0; i <= 2; ++i) {
        auto val = gen.getNextValue();
        std::cout << "main: " << val << '\n';
    }

}
```

Executing the program on the Compiler Explorer[5] makes the control flow transparent.

[5] https://godbolt.org/z/cTW9Gq

```
            promise_type::promise_type
            promise_type::get_return_object
        Generator::Generator
            promise_type::initial_suspend
        Generator::getNextValue
    getNext: start
    getNext: before co_yield
            promise_type::yield_value
main: 10
        Generator::getNextValue
    getNext: after co_yield
    getNext: before co_yield
            promise_type::yield_value
main: 20
        Generator::getNextValue
    getNext: after co_yield
    getNext: before co_yield
            promise_type::yield_value
main: 30
        Generator::~Generator
            promise_type::~promise_type
```

Generator generating an infinite data stream

Let's analyze the control flow.

The call `getNext()` (line 87) triggers the creation of the `Generator<int>`. First, the `promise_type` (line 38) is created, and the following `get_return_object` call (line 54) creates the generator (line 56) and stores it in a local variable. The result of this call is returned to the caller when the coroutine is suspended the first time. The initial suspension happens immediately (line 48). Because the member function call `initial_suspend` returns an awaitable `std::suspend_always` (line 48), the control flow continues with the coroutine `getNext` until the instruction `co_yield value` (line 79). This call is mapped to the call `yield_value(int value)` (line 59), and the current value is prepared `current_value = value` (line 61). The member function `yield_value(int value)` returns the awaitable `std::suspend_always` (line 59). Consequently, the execution of the coroutine pauses and the control flow goes back to the `main` function, and the for loop starts (line 89). The call `gen.getNextValue()` (line 89) starts the execution of the coroutine by resuming the coroutine, using `coro.resume()` (line 34). Further, the function `getNextValue()` returns the current value that was prepared using the previously invoked member function `yield_value(int value)` (line 59). Finally, the generated number is displayed in line 90, and the for loop continues. In the end, the generator and the promise are destructed.

After this detailed analysis, I would like to make a first modification of the control flow.

7.3.1 Modifications

The snippets and line numbers are based on the previous program `infiniteDataStreamComments.cpp`. I only show the modifications.

7.3.1.1 The Coroutine is Not Resumed

When I disable the resumption of the coroutine (`gen.getNextValue()` in line 89) and the display of its value (line 90), the coroutine pauses immediately.

Not resuming the coroutine

```
int main() {

    auto gen = getNext();
    for (int i = 0; i <= 2; ++i) {
        // auto val = gen.getNextValue();
        // std::cout << "main: " << val << '\n';
    }

}
```

The coroutine never runs. Consequently, the generator and its promise are created and destroyed.

```
    promise_type::promise_type
    promise_type::get_return_object
Generator::Generator
    promise_type::initial_suspend
Generator::~Generator
    promise_type::~promise_type
```

Not resuming the coroutine

7.3.1.2 `initial_suspend` Never Suspends

In the program, the member function `initial_suspend` returns the awaitable `std::suspend_always` (line 46). As its name suggests, the awaitable `std::suspends_always` causes the coroutine to pause immediately. Let me return `std::suspend_never` instead of `std::suspend_always`.

`initial_suspend` suspends never

```
std::suspend_never initial_suspend() {
    std::cout << "            promise_type::initial_suspend" << '\n';
    return {};
}
```

In this case, the coroutine runs immediately and pauses when the function `yield_value` (line 59) is invoked. A subsequent call `gen.getNextValue()` (line 89), resumes the coroutine and triggers the execution of the member function `yield_value` once more. The result is that the starting value `10` is ignored, and the coroutine returns the values `20`, `30`, and `40`.

```
            promise_type::promise_type
            promise_type::get_return_object
        Generator::Generator
            promise_type::initial_suspend
    getNext: start
    getNext: before co_yield
            promise_type::yield_value
        Generator::getNextValue
    getNext: after co_yield
    getNext: before co_yield
            promise_type::yield_value
main: 20
        Generator::getNextValue
    getNext: after co_yield
    getNext: before co_yield
            promise_type::yield_value
main: 30
        Generator::getNextValue
    getNext: after co_yield
    getNext: before co_yield
            promise_type::yield_value
main: 40
        Generator::~Generator
            promise_type::~promise_type
```

Don't Resuming the Coroutine

7.3.1.3 `yield_value` Never Suspends

The member function `yield_value` (line 59) is triggered by the call `co_yield value` and prepares the `current_value` (line 61). The function returns the awaitable `std::suspend_always` (line 62) and,

therefore, pauses the coroutine. Consequently, a subsequent call `gen.getNextValue` (line 89) has to resume the coroutine. When I change the return value of the member function `yield_value` to `std::suspend_never`, let me see what happens.

`yield_value` never suspends

```
std::suspend_never yield_value(int value) {
    std::cout << "           promise_type::yield_value" << '\n';
    current_value = value;
    return {};
}
```

As you may guess, the while loop (lines 77 - 82) runs forever, and the coroutine does not return anything.

```
           promise_type::promise_type
           promise_type::get_return_object
       Generator::Generator
           promise_type::initial_suspend
       Generator::getNextValue
   getNext: start
   getNext: before co_yield
           promise_type::yield_value
   getNext: after co_yield
   getNext: before co_yield
           promise_type::yield_value
   getNext: after co_yield
   getNext: before co_yield
           promise_type::yield_value
   getNext: after co_yield
   getNext: before co_yield
           promise_type::yield_value
   getNext: after co_yield
   getNext: before co_yield
           promise_type::yield_value
   getNext: after co_yield
```

`yield_value` Never Suspends

It is straightforward to restructure the generator `infiniteDataStreamComments.cpp` so that it produces a finite number of values.

7.3.2 Generalization

You may wonder why I never used the full generic potential of `Generator`. Let me adjust its implementation to produce the successive elements of an arbitrary container of the Standard Template Library.

Generator successively returning each element

```
// coroutineGetElements.cpp

#include <coroutine>
#include <memory>
#include <iostream>
#include <string>
#include <vector>

template<typename T>
struct Generator {

    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;

    Generator(handle_type h): coro(h) {}

    handle_type coro;

    ~Generator() {
        if ( coro ) coro.destroy();
    }
    Generator(const Generator&) = delete;
    Generator& operator = (const Generator&) = delete;
    Generator(Generator&& oth): coro(oth.coro) {
        oth.coro = nullptr;
    }
    Generator& operator = (Generator&& oth) {
        coro = oth.coro;
        oth.coro = nullptr;
        return *this;
    }
    T getNextValue() {
        coro.resume();
        return coro.promise().current_value;
    }
    struct promise_type {
        promise_type() {}

```

```
39          ~promise_type() {}
40
41          std::suspend_always initial_suspend() {
42              return {};
43          }
44          std::suspend_always final_suspend() noexcept {
45              return {};
46          }
47          auto get_return_object() {
48              return Generator{handle_type::from_promise(*this)};
49          }
50
51          std::suspend_always yield_value(const T value) {
52              current_value = value;
53              return {};
54          }
55           void return_void() {}
56          void unhandled_exception() {
57              std::exit(1);
58          }
59
60          T current_value;
61      };
62
63  };
64
65  template <typename Cont>
66  Generator<typename Cont::value_type> getNext(Cont cont) {
67      for (auto c: cont) co_yield c;
68  }
69
70  int main() {
71
72      std::cout << '\n';
73
74      std::string helloWorld = "Hello world";
75      auto gen = getNext(helloWorld);
76      for (int i = 0; i < helloWorld.size(); ++i) {
77          std::cout << gen.getNextValue() << " ";
78      }
79
80      std::cout << "\n\n";
81
82      auto gen2 = getNext(helloWorld);
83      for (int i = 0; i < 5 ; ++i) {
```

```
84          std::cout << gen2.getNextValue() << " ";
85      }
86
87      std::cout << "\n\n";
88
89      std::vector myVec{1, 2, 3, 4 ,5};
90      auto gen3 = getNext(myVec);
91      for (int i = 0; i < myVec.size() ; ++i) {
92          std::cout << gen3.getNextValue() << " ";
93      }
94
95      std::cout << '\n';
96
97  }
```

In this example, the generator is instantiated and used three times. In the first two cases, `gen` (line 76) and `gen2` (line 83) are initialized with `std::string helloWorld`, while `gen3` uses a `std::vector<int>` (line 91). The output of the program should not be surprising. Line 78 returns all characters of the string `helloWorld` successively, line 85 only the first five characters, and line 93 the elements of the `std::vector<int>`.

You can try out the program on the Compiler Explorer[6].

```
H e l l o   w o r l d

H e l l o

1 2 3 4 5
```

A generator successively returning each element

To make it short. The implementation of the `Generator<T>` is almost identical to the previous one. The crucial difference with the previous program is the coroutine `getNext`.

getNext

```
template <typename Cont>
Generator<typename Cont::value_type> getNext(Cont cont) {
    for (auto c: cont) co_yield c;
}
```

`getNext` is a function template that takes a container as an argument and iterates in a range-based for loop through all container elements. After each iteration, the function template pauses. The return type `Generator<typename Cont::value_type>` may look surprising to you. `Cont::value_type` is

[6] https://godbolt.org/z/j9znva

a dependent template parameter for which the parser needs a hint. By default, the compiler assumes a non-type if it could be interpreted as a type or a non-type. For this reason, I have to put `typename` in front of `Cont::value_type`.

The generalized coroutine still has a few flaws. In the final iteration, I fix these flaws and extend its interface so that it can be used in a range-based for-loop.

7.3.3 Iterator Protocol

A generator supporting the iterator protocol

```
// coroutineRange.cpp

#include <concepts>
#include <coroutine>
#include <exception>
#include <iostream>
#include <string>
#include <vector>


template<typename T>
struct Generator {

    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;


    Generator(handle_type h) : coro{h} { }
    handle_type coro;
    ~Generator() { if (coro) coro.destroy(); }

    Generator(const Generator&) = delete;
    Generator& operator=(const Generator&) = delete;
    Generator(Generator&& oth): coro(oth.coro) {
        oth.coro = nullptr;
    }
    Generator& operator = (Generator&& oth) {
        coro = oth.coro;
        oth.coro = nullptr;
        return *this;
    }

    struct promise_type {
        T coroValue{};
```

```
35
36          std::suspend_always yield_value(const T val) {
37              coroValue = val;
38              return {};
39          }
40
41          auto get_return_object() {
42              return std::coroutine_handle<promise_type>::from_promise(*this);
43          }
44          std::suspend_always initial_suspend() { return {}; }
45          void return_void() { }
46          void unhandled_exception() { std::terminate(); }
47          std::suspend_always final_suspend() noexcept { return {}; }
48      };
49
50      struct Iterator {
51          handle_type coro;
52          Iterator(auto p) : coro{p} { }
53          void getNext() {
54              if (coro) {
55                  coro.resume();
56                  if (coro.done()) {
57                      coro = nullptr;
58                  }
59              }
60          }
61          T operator*() const {
62              return coro.promise().coroValue;
63          }
64          Iterator operator++() {
65              getNext();
66              return *this;
67          }
68          bool operator== (const Iterator& i) const = default;
69      };
70
71
72      Iterator begin() const {
73          if (!coro || coro.done()) {
74              return Iterator{nullptr};
75          }
76          Iterator itor{coro};
77          itor.getNext();
78          return itor;
79      }
```

```
80
81      Iterator end() const {
82          return Iterator{nullptr};
83      }
84  };
85
86  template <std::ranges::forward_range Cont>
87  Generator<typename Cont::value_type> getCoroutine(Cont cont) {
88
89      for (const auto& c: cont){
90          co_yield c;
91      }
92
93  }
94
95  int main() {
96
97      std::string helloWorld = "Hello world";
98      auto genChar = getCoroutine(helloWorld);
99      for (const auto& val : genChar) {
100         std::cout << val << " ";
101     }
102
103     std::cout << "\n\n";
104
105     std::vector myVec{1, 2, 3, 4, 5};
106     auto genNumbers = getCoroutine(myVec);
107     for (const auto& val : genNumbers) {
108         std::cout << val << " ";
109     }
110     std::cout << "\n\n";
111
112 }
```

Thanks to the member functions `begin` (line 72) und `end` (line 81), `Generator` supports the iterator protocol. Both member functions return iterator objects. `begin` return the first element of the container and `end` the end iterator `Iterator{nullptr}`. To get the first value, `begin` resumes one time the coroutine by calling `getNext()` (line 77),`Iterator` is a minimal iterator and support the three essential member functions `operator*`(line 61), `operator++` (line 64), and `operator==` (line 68).

- `operator*` returns the current value
- `operator++` increments the current value
- `operator!=` compares the current value with the end iterator. The compiler generates `operator!=` from `operator==`.

The `Generator` in the program `coroutineRange.cpp` is clearly more robust than the previous one in the program `coroutineGetElements.cpp`. This robustness is mainly due to the `Iterator`. The member function `getNext` (line 53) checks if the handle `coro` represents a coroutine `line 54`. Additionally, the member function `begin` (line 72) guarantees that the coroutine is only resumed if not completed: `coro.done()` (line 73). To resume an already completed coroutine is undefined behavior.

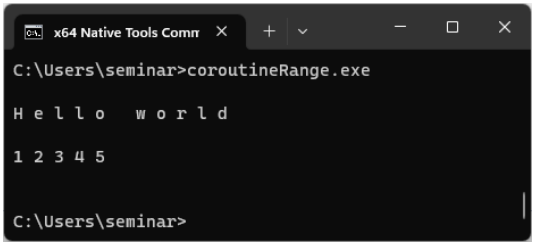
Resuming a completed coroutine

```
1 std::string helloWorld = "Hello world";
2 auto genChar = getCoroutine(helloWorld);
3 for (const auto& val : genChar) {
4     std::cout << val << " ";
5 }
6
7 for (const auto& val : genChar) {
8     std::cout << val << " ";
9 }
```

Without the check `coro.done()` (line 73) in the program `coroutineRange.cpp`. The second range-based for-loop (line 7) would trigger are resumption of the coroutine, and, therefore, undefined behavior.

The generic function `getCoroutine` (line 86) uses the concept `std::ranges::forward_range`.

Finally, here is the output of the program:



A generator supporting the iterator protocol

7.4 Various Job Workflows

Cippi digs the garden

Before I modify the workflow from section co_await, I want to make the awaiter workflow more transparent.

7.4.1 The Transparent Awaiter Workflow

I added a few output messages to the program startJob.cpp.

Starting a job on request (including comments)

```
1  // startJobWithComments.cpp
2
3  #include <coroutine>
4  #include <iostream>
5
6  struct MySuspendAlways {
7      bool await_ready() const noexcept {
8          std::cout << "        MySuspendAlways::await_ready"  << '\n';
9          return false;
10     }
11     void await_suspend(std::coroutine_handle<>) const noexcept {
12         std::cout << "        MySuspendAlways::await_suspend"  << '\n';
13
```

```
    }
    void await_resume() const noexcept {
        std::cout << "        MySuspendAlways::await_resume"  << '\n';
    }
};

struct MySuspendNever {
    bool await_ready() const noexcept {
        std::cout << "        MySuspendNever::await_ready"  << '\n';
        return true;
    }
    void await_suspend(std::coroutine_handle<>) const noexcept {
        std::cout << "        MySuspendNever::await_suspend"  << '\n';

    }
    void await_resume() const noexcept {
        std::cout << "        MySuspendNever::await_resume"  << '\n';
    }
};

struct Job {
    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;
    handle_type coro;
    Job(handle_type h): coro(h){}
    ~Job() {
        if ( coro ) coro.destroy();
    }
    void start() {
        coro.resume();
    }


    struct promise_type {
        auto get_return_object() {
            return Job{handle_type::from_promise(*this)};
        }
        MySuspendAlways initial_suspend() {
            std::cout << "    Job prepared" << '\n';
            return {};
        }
        MySuspendAlways final_suspend() noexcept {
            std::cout << "    Job finished" << '\n';
            return {};
        }
```

```
59          void return_void() {}
60          void unhandled_exception() {}
61
62      };
63  };
64
65  Job prepareJob() {
66      co_await MySuspendNever();
67  }
68
69  int main() {
70
71      std::cout <<  "Before job" << '\n';
72
73      auto job = prepareJob();
74      job.start();
75
76      std::cout <<  "After job" <<  '\n';
77
78  }
```

First of all, I replaced the predefined awaitables `std::suspend_always` and `std::suspend_never` with awaitables `MySuspendAlways` (line 6) and `MySuspendNever` (line 20). I use them in lines 51, 55, and 66. The awaitables mimic the behavior of the predefined awaitables but additionally write a comment. Due to the use of `std::cout`, the member functions `await_ready`, `await_suspend`, and `await_resume` cannot be declared as `constexpr`.

The screenshot of the program execution shows the control flow nicely, which you can directly observe on the Compiler Explorer[7].

[7] https://godbolt.org/z/T5rcE4

```
Before job
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
        MySuspendAlways::await_resume
        MySuspendNever::await_ready
        MySuspendNever::await_resume
    Job finished
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
After job
```

Starting a job on request (including comments)

The function `initial_suspend` (line 51) is executed at the beginning of the coroutine, and the function `final_suspend` at its end (line 55). The call `prepareJob()` (line 73) triggers the creation of the coroutine object, and the function call `job.start()` its resumption and, hence, completion (line 74). Consequently, the members `await_ready`, `await_suspend`, and `await_resume` of `MySuspendAlways` are executed. When you don't resume the awaitable such as the coroutine object returned by the member function `final_suspend`, the function `await_resume` is not processed. In contrast, the awaitable's `MySuspendNever` function is immediately ready because `await_ready` returns `true` and, hence, does not suspend.

Thanks to the comments, you should have an elementary understanding of the awaiter workflow. Now, it's time to vary it.

7.4.2 Automatically Resuming the Awaiter

In the previous workflow, I explicitly started the job.

Explicitly starting the job

```
int main() {

    std::cout <<  "Before job" << '\n';

    auto job = prepareJob();
    job.start();

    std::cout <<  "After job" <<  '\n';

}
```

This explicit invoking of `job.start()` was necessary because `await_ready` in the awaitable `MySuspendAlways` always returned `false`. Now let's assume that `await_ready` can return `true` or `false` and the job is not

explicitly started. A short reminder: When `await_ready` returns `true`, the function `await_resume` is directly invoked but not `await_suspend`.

Automatically Resuming the Awaiter

```
// startJobWithAutomaticResumption.cpp

#include <coroutine>
#include <functional>
#include <iostream>
#include <random>

std::random_device seed;
auto gen = std::bind_front(std::uniform_int_distribution<>(0,1),
                           std::default_random_engine(seed()));

struct MySuspendAlways {
    bool await_ready() const noexcept {
        std::cout << "        MySuspendAlways::await_ready"  << '\n';
        return gen();
    }
    bool await_suspend(std::coroutine_handle<> handle) const noexcept {
        std::cout << "        MySuspendAlways::await_suspend"  << '\n';
        handle.resume();
        return true;

    }
    void await_resume() const noexcept {
        std::cout << "        MySuspendAlways::await_resume"  << '\n';
    }
};

struct Job {
    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;
    handle_type coro;
    Job(handle_type h): coro(h){}
    ~Job() {
        if ( coro ) coro.destroy();
    }

    struct promise_type {
        auto get_return_object() {
            return Job{handle_type::from_promise(*this)};
        }
        MySuspendAlways initial_suspend() {
```

```
42              std::cout << "    Job prepared" << '\n';
43              return {};
44          }
45          std::suspend_always final_suspend() noexcept {
46              std::cout << "    Job finished" << '\n';
47              return {};
48          }
49          void return_void() {}
50          void unhandled_exception() {}
51
52      };
53  };
54
55  Job performJob() {
56      co_await std::suspend_never();
57  }
58
59  int main() {
60
61      std::cout <<  "Before jobs" << '\n';
62
63      performJob();
64      performJob();
65      performJob();
66      performJob();
67
68      std::cout <<  "After jobs" <<  '\n';
69
70  }
```

First of all, the coroutine is now called `performJob` and runs automatically. `gen` (line 9) is a random number generator for the numbers 0 or 1. It uses for its job the default random engine, initialized with the seed. Thanks to `std::bind_front`, I can bind it together with the `std::uniform_int_distribution` to get a callable which, when used, gives me a random number 0 or 1.

I removed in this example the awaitables with predefined awaitables from the C++ standard, except the awaitable `MySuspendAlways` as the return type of the member function `initial_suspend` (line 41). `await_ready` (line 13) returns a boolean. If the boolean is `true`, the control flow jumps directly to the `await_resume` member function (line 23), if `false`, the coroutine is immediately suspended and the function `await_suspend` is executed (line 17). The function `await_suspend` gets the handle to the coroutine and uses it to resume the coroutine (line 19). Instead of returning the value `true`, `await_suspend` can also return `void`.

The following screenshot shows: When `await_ready` returns `true`, the function `await_resume` is called, when `await_ready` returns `false`, the function `await_suspend` is also called.

You can try out the program on the Compiler Explorer[8].

```
Before jobs
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
After jobs
```

Automatically Resuming the Awaiter

Let me improve the presented program more and resume the awaiter on a separate thread.

7.4.3 Automatically Resuming the Awaiter on a Separate Thread

The following program is based on the previous one.

[8] https://godbolt.org/z/8b1Y14

Automatically Resuming the Awaiter on a Seperate Thread

```
// startJobWithAutomaticResumptionOnThread.cpp

#include <coroutine>
#include <functional>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

std::random_device seed;
auto gen = std::bind_front(std::uniform_int_distribution<>(0,1),
                           std::default_random_engine(seed()));

struct MyAwaitable {
    std::jthread& outerThread;
    bool await_ready() const noexcept {
        auto res = gen();
        if (res) std::cout << " (executed)" << '\n';
        else std::cout << " (suspended)" << '\n';
        return res;
    }
    void await_suspend(std::coroutine_handle<> h) {
        outerThread = std::jthread([h] { h.resume(); });
    }
    void await_resume() {}
};


struct Job{
    static inline int JobCounter{1};
    Job() {
        ++JobCounter;
    }

    struct promise_type {
        int JobNumber{JobCounter};
        Job get_return_object() { return {}; }
        std::suspend_never initial_suspend() {
            std::cout << "    Job " << JobNumber << " prepared on thread "
                      << std::this_thread::get_id();
            return {};
        }
        std::suspend_never final_suspend() noexcept {
            std::cout << "    Job " << JobNumber << " finished on thread "
```

```
45                          << std::this_thread::get_id() << '\n';
46              return {};
47          }
48          void return_void() {}
49          void unhandled_exception() { }
50      };
51  };
52
53  Job performJob(std::jthread& out) {
54      co_await MyAwaitable{out};
55  }
56
57  int main() {
58
59      std::vector<std::jthread> threads(8);
60      for (auto& thr: threads) performJob(thr);
61
62  }
```

The main difference with the previous program is the new awaitable `MyAwaitable`, used in the coroutine `performJob` (line 54). On the contrary, the coroutine object returned from the coroutine `performJob` is straightforward. Essentially, its member functions `initial_suspend` (line 38) and `final_suspend` (line 43) returns the predefined awaitable `std::suspend_never`. Additionally, both functions show the `JobNumber` of the executed job and the thread ID on which it runs. The screenshot shows which coroutine runs immediately and which one is suspended. Thanks to the thread id, you can observe that suspended coroutines are resumed on a different thread.

You can try out the program on the Wandbox[9].

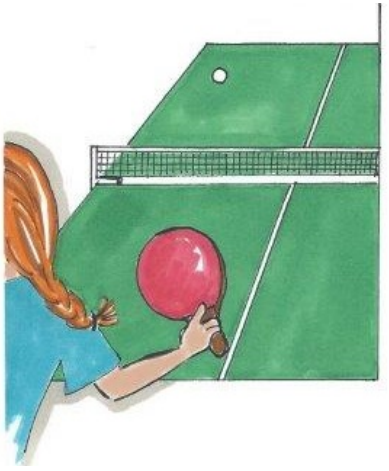
[9] https://wandbox.org/permlink/skHgWKF0SYAwp8Dm

```
Job 1 prepared on thread 140434982274944 (executed)
Job 1 finished on thread 140434982274944
Job 2 prepared on thread 140434982274944 (suspended)
Job 3 prepared on thread 140434982274944 (suspended)
Job 4 prepared on thread 140434982274944 (suspended)
Job 2 finished on thread 140434877310720
Job 5 prepared on thread 140434982274944 (executed)
Job 5 finished on thread 140434982274944
Job 6 prepared on thread 140434982274944 (suspended)
Job 7 prepared on thread 140434982274944 (suspended)
Job 3 finished on thread 140434868918016
Job 8 prepared on thread 140434982274944 (executed)
Job 8 finished on thread 140434982274944
Job 4 finished on thread 140434860525312
Job 6 finished on thread 140434852132608
Job 7 finished on thread 140434843739904
```

Automatically Resuming the Awaiter on a Separate Thread

Let me discuss the interesting control flow of the program. Line 59 creates eight default-constructed threads, which the coroutine `performJob` (line 53) takes by reference. Further, the reference becomes the argument for creating `MyAwaitable{out}` (line 54). Depending on the value of `res` (line 17), and, therefore, the return value of the function `await_ready`, the awaitable continues (`res` is `true`) to run or is suspended (`res` is `false`). In case `MyAwaitable` is suspended, the function `await_suspend` (line 22) is executed. Thanks to the assignment of `outerThread` (line 23), it becomes a running thread. The running threads must outlive the lifetime of the coroutine. For this reason, the threads have the scope of the `main` function.

7.5 Fast Synchronization of Threads



Cippi plays ping-pong

The Reference PCs

You should take the performance numbers with a **grain of salt**. I'm not interested in the exact number for each variation of the algorithms on Linux and Windows. I'm more interested in getting a gut feeling about which algorithms may work and which may not. I'm not comparing the absolute numbers of my Linux desktop with the numbers on my Windows laptop, but I'm interested to know if some algorithms work better on Linux or Windows.

When you want to synchronize threads more than once, you can use condition variables, `std::atomic_flag`, `std::atomic<bool>`, or semaphores. In this section, I would like to answer the question: which variant is the fastest.

To get comparable numbers, I implement a ping-pong game. One thread executes a `ping` function (or ping thread for short), and the other thread a `pong` function (or pong thread for short). The ping thread waits for the pong-thread notification and sends the notification back to the pong thread. The game stops after 1,000,000 ball changes. I perform each game five times to get comparable performance numbers.

About the Numbers

I made my performance test at the end of 2020 with the brand new Visual Studio compiler 19.28 because it already supported synchronization with atomics (`std::atomic_flag` and `std::atomic`) and semaphores. Additionally, I compiled the examples with maximum optimization (`/Ox`). The performance number should only give a rough idea of the relative performance of the various ways to synchronize threads. If you want to have the exact number on your platform, you have to repeat the tests.

Let me start the comparison with C++11.

7.5.1 Condition Variables

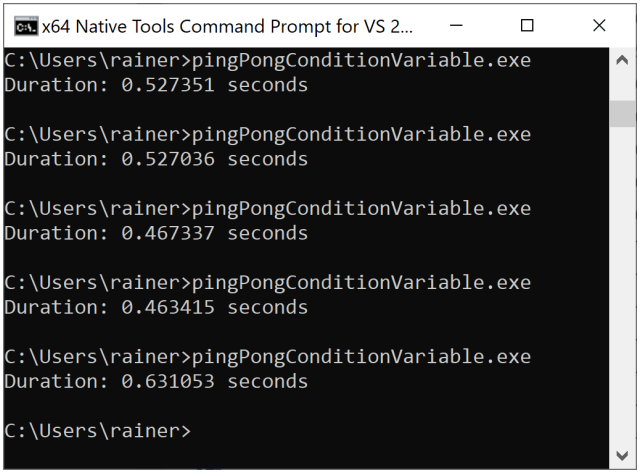
Multiple time synchronization with a condition variable

```
1  // pingPongConditionVariable.cpp
2
3  #include <condition_variable>
4  #include <iostream>
5  #include <atomic>
6  #include <thread>
7
8  bool dataReady{false};
9
10 std::mutex mutex_;
11 std::condition_variable condVar1;
12 std::condition_variable condVar2;
13
14 std::atomic<int> counter{};
15 constexpr int countlimit = 1'000'000;
16
17 void ping() {
18
19     while(counter <= countlimit) {
20         {
21             std::unique_lock<std::mutex> lck(mutex_);
22             condVar1.wait(lck, []{return dataReady == false;});
23             dataReady = true;
24         }
25         ++counter;
26         condVar2.notify_one();
27   }
28 }
29
30 void pong() {
```

```
31
32      while(counter <= countlimit) {
33          {
34              std::unique_lock<std::mutex> lck(mutex_);
35              condVar2.wait(lck, []{return dataReady == true;});
36              dataReady = false;
37          }
38          condVar1.notify_one();
39    }
40
41  }
42
43  int main(){
44
45      auto start = std::chrono::system_clock::now();
46
47      std::thread t1(ping);
48      std::thread t2(pong);
49
50      t1.join();
51      t2.join();
52
53      std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
54      std::cout << "Duration: " << dur.count() << " seconds" << '\n';
55  }
```

I use two condition variables in the program: `condVar1` and `condVar2`. The ping thread waits for the notification of `condVar1` and sends its notification with `condVar2`. Variable `dataReady` protects against spurious and lost wakeups. The ping-pong game ends when `counter` reaches the `countlimit`. The `notify_one` calls (lines 26 and 38) and the `counter` are thread-safe and are, therefore, outside the critical region.

Here are the numbers.



Multiple time synchronizations with condition variables

The average execution time is 0.52 seconds.

Porting this workflow to `std::atomic_flag` in C++20 is straightforward.

7.5.2 `std::atomic_flag`

Here is the same workflow using two atomic flags and then one.

7.5.2.1 Two Atomic Flags

In the following program, I replace the waiting on the condition variable with the waiting on the atomic flag and the condition variable's notification with the atomic-flag setting followed by the notification.

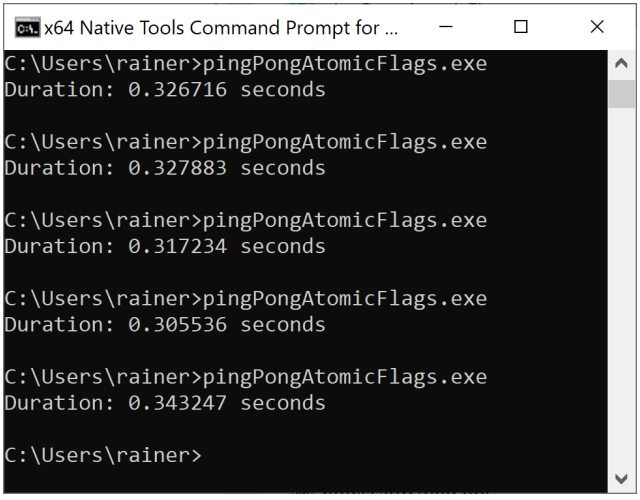
Multiple time synchronization with two atomic flags

```
1  // pingPongAtomicFlags.cpp
2
3  #include <iostream>
4  #include <atomic>
5  #include <thread>
6
7  std::atomic_flag condAtomicFlag1{};
8  std::atomic_flag condAtomicFlag2{};
9
10 std::atomic<int> counter{};
11 constexpr int countlimit = 1'000'000;
12
```

```
13  void ping() {
14      while(counter <= countlimit) {
15          condAtomicFlag1.wait(false);
16          condAtomicFlag1.clear();
17
18          ++counter;
19
20          condAtomicFlag2.test_and_set();
21          condAtomicFlag2.notify_one();
22      }
23  }
24
25  void pong() {
26      while(counter <= countlimit) {
27          condAtomicFlag2.wait(false);
28          condAtomicFlag2.clear();
29
30          condAtomicFlag1.test_and_set();
31          condAtomicFlag1.notify_one();
32      }
33  }
34
35  int main() {
36
37      auto start = std::chrono::system_clock::now();
38
39      condAtomicFlag1.test_and_set();
40      std::thread t1(ping);
41      std::thread t2(pong);
42
43      t1.join();
44      t2.join();
45
46      std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
47      std::cout << "Duration: " << dur.count() << " seconds" << '\n';
48
49  }
```

A call `condAtomicFlag1.wait(false)` (line 15) blocks if the atomic flag's value is `false`, and returns if `condAtomicFlag1` has the value `true`. The boolean value serves as a kind of predicate and must, therefore, be set back to `false` (line 15). Before the notification (line 21) is sent to the pong thread, `condAtomicFlag1` is set to true (line 20). The initial setting of `condAtomicFlag1` (line 39) to `true` starts the game.

Thanks to `std::atomic_flag`, the game ends faster.



Multiple time synchronization with two atomic flags

On average, a game takes 0.32 seconds.

When you analyze the program, you may recognize that one atomic flag is sufficient for the workflow.

7.5.2.2 One Atomic Flag

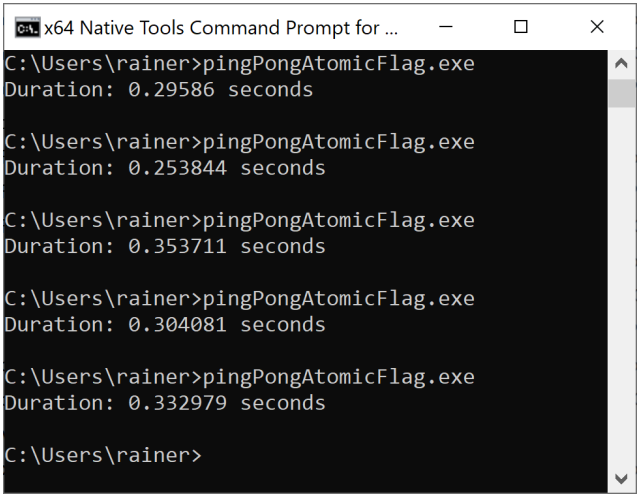
Using one atomic flag makes the workflow easier to understand.

Multiple time synchronization with one atomic flag

```
// pingPongAtomicFlag.cpp

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

std::atomic_flag condAtomicFlag{};

std::atomic<int> counter{};
constexpr int countlimit = 1'000'000;

void ping() {
    while(counter <= countlimit) {
        condAtomicFlag.wait(true);
        condAtomicFlag.test_and_set();

        ++counter;

```

```
        condAtomicFlag.notify_one();
    }
}

void pong() {
    while(counter <= countlimit) {
        condAtomicFlag.wait(false);
        condAtomicFlag.clear();
        condAtomicFlag.notify_one();
    }
}

int main() {

    auto start = std::chrono::system_clock::now();

    condAtomicFlag.test_and_set();
    std::thread t1(ping);
    std::thread t2(pong);

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

    std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
    std::cout << "Duration: " << dur.count() << " seconds" << '\n';

}
```

In this case, the ping thread blocks on `true` but the pong thread blocks on `false`. From the performance perspective, using one or two atomic flags makes no difference.



Multiple time synchronization with one atomic flag

The average execution time is 0.31 seconds.

I used `std::atomic_flag` like an atomic boolean in this example. Let's give it another try with `std::atomic<bool>`.

7.5.3 `std::atomic<bool>`

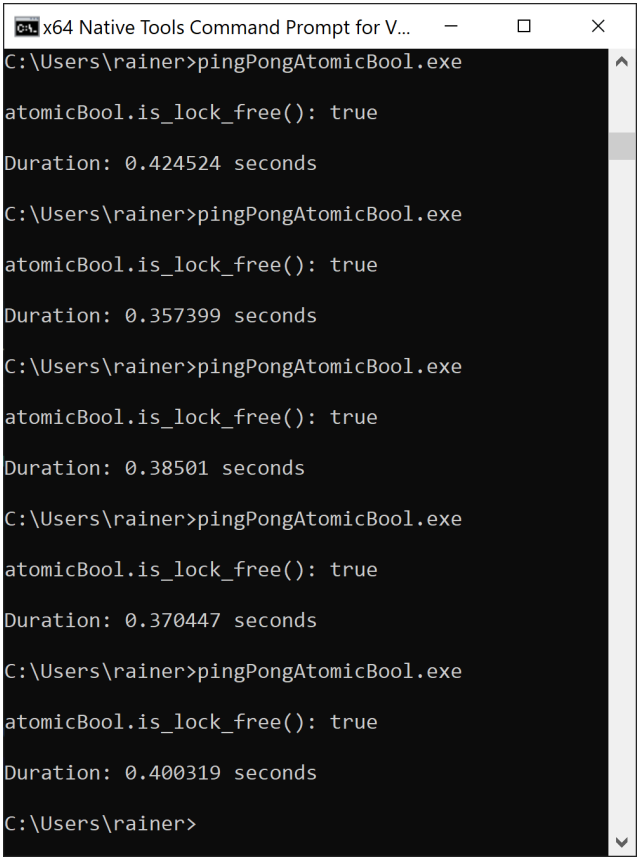
The following C++20 implementation is based on `std::atomic`.

Multiple time synchronization with an atomic bool

```
1  // pingPongAtomicBool.cpp
2
3  #include <iostream>
4  #include <atomic>
5  #include <thread>
6
7  std::atomic<bool> atomicBool{};
8
9  std::atomic<int> counter{};
10 constexpr int countlimit = 1'000'000;
11
12 void ping() {
13     while(counter <= countlimit) {
14         atomicBool.wait(true);
15         atomicBool.store(true);
16
17         ++counter;
```

```
18
19          atomicBool.notify_one();
20      }
21  }
22
23  void pong() {
24      while(counter <= countlimit) {
25          atomicBool.wait(false);
26          atomicBool.store(false);
27          atomicBool.notify_one();
28      }
29  }
30
31  int main() {
32
33      std::cout << std::boolalpha << '\n';
34
35      std::cout << "atomicBool.is_lock_free(): "
36                << atomicBool.is_lock_free()  << '\n';
37
38      std::cout << '\n';
39
40      auto start = std::chrono::system_clock::now();
41
42      atomicBool.store(true);
43      std::thread t1(ping);
44      std::thread t2(pong);
45
46      t1.join();
47      t2.join();
48
49      std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
50      std::cout << "Duration: " << dur.count() << " seconds" << '\n';
51
52  }
```

`std::atomic<bool>` can internally use a locking mechanism like a mutex. My Windows run time is lock-free.



Multiple time synchronization with an atomic bool

On average, the execution time is 0.38 seconds.

From the readability perspective, this implementation based on `std::atomic` is straightforward to understand. This observation also holds for the next implementation of the ping-pong game based on semaphores.

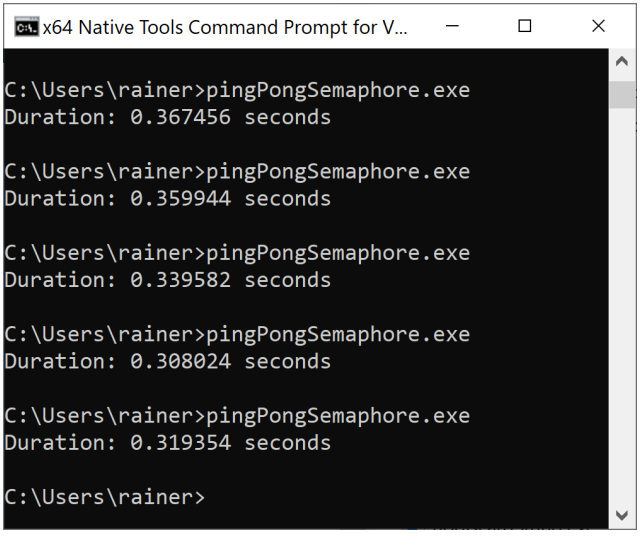
7.5.4 Semaphores

Semaphores promise to be faster than condition variables. Let's see if this is true.

Multiple time synchronization with semaphores

```cpp
// pingPongSemaphore.cpp

#include <iostream>
#include <semaphore>
#include <thread>

std::counting_semaphore<1> signal2Ping(0);
std::counting_semaphore<1> signal2Pong(0);

std::atomic<int> counter{};
constexpr int countlimit = 1'000'000;

void ping() {
    while(counter <= countlimit) {
        signal2Ping.acquire();
        ++counter;
        signal2Pong.release();
    }
}

void pong() {
    while(counter <= countlimit) {
        signal2Pong.acquire();
        signal2Ping.release();
    }
}

int main() {

    auto start = std::chrono::system_clock::now();

    signal2Ping.release();
    std::thread t1(ping);
    std::thread t2(pong);

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

    std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
    std::cout << "Duration: " << dur.count() << " seconds" << '\n';

}
```

The program `pingPongsemaphore.cpp` uses two semaphores: `signal2Ping` and `signal2Pong` (lines 7 and 8). Both can have the two values 0 or 1, and are initialized with 0. This means when the value is 0 for the semaphore `signal2Ping`, a call `signal2Ping.release()` (lines 24 and 32) sets the value to 1 and is, therefore, a notification. A `signal2Ping.acquire()` (line 15) call blocks until the value becomes 1. The same argumentation holds for the second semaphore `signal2Pong`.



Multiple time synchronization with semaphores

On average, the execution time is 0.33 seconds.

7.5.5 All Numbers

As expected, condition variables are the slowest way, and atomic flag the fastest way to synchronize threads. In between is the performance of a `std::atomic<bool>`. There is one downside with `std::atomic<bool>`. `std::atomic_flag` is the only atomic data type that is always lock-free. Semaphores impressed me most because they are nearly as fast as atomic flags.

Execution Time

	Condition Variables	Two Atomic Flags	One Atomic Flag	Atomic Boolean	Semaphores
Execution Time	0.52	0.32	0.31	0.38	0.33

Distilled Information

- The section coroutines introduced an eager future, using co_return. This future is an ideal starting point to make it lazy and finally let it run on its thread.
- Thanks to the Ranges Library, I can implement Python's 2 filter, map, and list comprehension functions in C++20.
- Modifications of the generator for an infinite data stream reveal its nature. When the member function initial_suspend returns std::suspend_never, the coroutine starts immediately and ignores the first value. In contrast, returning std::suspend_never from the function yield_value ends in an infinite loop. When you forget to resume the coroutine, it will never run. The generator Generator<T> is generally applicable. Instead of an infinite data stream, it can successively return the elements of an arbitrary container of the Standard Template Library.
- Implementing your own awaitable MySuspendNever and MySuspendAlways makes the awaiter workflow transparent. Adapting the awaitable MySuspendAlways enables it to create an Awaiter that resumes itself if necessary. Changing awaitable allows you to automatically resume the coroutine on a separate thread.
- If you want to synch threads more than once, you have many options. You can use condition variables, std::atomic_flag, std::atomic<bool>, or semaphores. This case study answers the question: Which variant is the fastest one? The numbers show that condition variables are the slowest way and atomic flags are the fastest way to synchronize threads. The performance of std::atomic<bool> is in between. Semaphores are nearly as fast as atomic flags.

Epilogue

Congratulations! If you are reading these lines, you have mastered the challenging and exciting C++20 standard. C++20 is a C++ standard that likely has the same influence on C++, such as the other two significant C++ standards: C++98 and C++11. Due to C++11, the following names for the C++ standards are used by the C++ community.

- **Legacy C++**: C++98, and C++03
- **Modern C++** : C++11, C++14, and C++17
- **<Placeholder>**: C++20

I'm not sure what name will be used for C++20 in the future. I'm only sure that C++20 starts a new C++ area. Let me remind you why, in particular, the *Big Four* change the way we program in C++.

- **Concepts**: Concepts are revolutionizing the way we think about and write generic code. Thanks to them, we can reason about our program for the first time in semantic categories such as `Number` or `Ordering`.
- **Modules**: Modules are the starting point of software components. Modules help overcome the deficiencies of legacy headers and macros.
- **Ranges**: The ranges library extends the Standard Template Library with functional ideas. Algorithms can operate directly on the containers, be evaluated lazily, and be composed.
- **Coroutines**: Thanks to coroutines, asynchronous programming becomes a first-class citizen in C++. Coroutines transform blocking function calls in waiting and are highly valuable in event-driven systems such as simulations, servers, or user interfaces.

C++20 is just the starting point. In the chapter about C++23 and Beyond, I give more details on the near future of C++.

To make it short: C++ has a bright, shining future.

Further Information

8. C++23 and Beyond

Anyone who thinks a significant C++ standard is followed by a small C++ standard is wrong. C++23 provides powerful extensions to C++20. These extensions include the core language but, in particular, the standard library. Thanks to contracts, reflection, and pattern matching, the C++ future beyond C++23 shines pretty bright.

8.1 C++23

This C++23 chapter should give you a first impression but not a detailed presentation of C++23.

Let me start with the core language.

8.1.1 Core Language

8.1.1.1 Deducing This

Deducing this, sometimes also called explicit object parameter, allows it to make the implicit this pointer of a member function explicit. Similar to Python[1], the explicit object parameter must be the first function parameter and is called in C++, by convention, `Self` and `self`.

```
struct Test {
    void implicitParameter();              // implicit this pointer
    void explictParameter(this Self& self); // explicit this pointer
};
```

Implicit this enables new programming techniques in C++23: deduplication of function overloading based on the object's lvalue/rvalue value category and its constness Additionally, you can reference a lambda and invoke it recursively. Furthermore, deducing this simplifies the implementation of CRTP[2].

8.1.1.1.1 Deduplicating Function Overloading

Assume you want to overload a member function based on the lvalue/rvalue value category and constness of the calling object. This means you have to overload your member function four times.

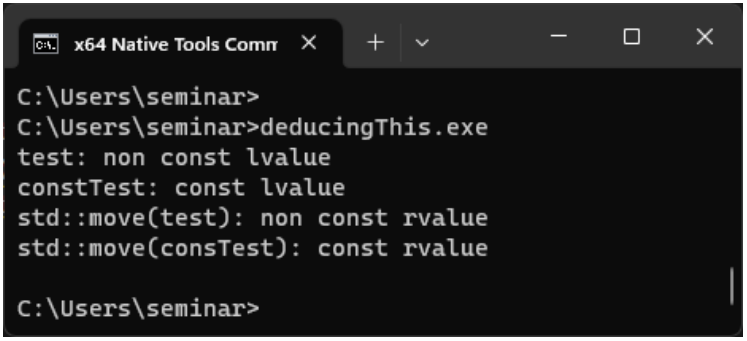
Deduplicating Function Overloading

```
1  // deducingThis.cpp
2
3  #include <iostream>
4
5  struct Test {
6      template <typename Self>
7      void explicitCall(this Self&& self, const std::string& text) {
8          std::cout << text << ": ";
9          std::forward<Self>(self).implicitCall();
10         std::cout << '\n';
11     }
12
```

[1] https://www.python.org/
[2] https://www.modernescpp.com/index.php/c-is-still-lazy

```
13      void implicitCall() & {
14          std::cout << "non const lvalue";
15      }
16
17      void implicitCall() const& {
18          std::cout << "const lvalue";
19      }
20
21      void implicitCall() && {
22          std::cout << "non const rvalue";
23      }
24
25      void implicitCall() const&& {
26          std::cout << "const rvalue";
27      }
28
29
30  };
31
32  int main() {
33
34      std::cout << '\n';
35
36      Test test;
37      const Test constTest;
38
39      test.explicitCall("test");
40      constTest.explicitCall("constTest");
41      std::move(test).explicitCall("std::move(test)");
42      std::move(constTest).explicitCall("std::move(constTest)");
43
44      std::cout << '\n';
45
46  }
```

Lines 13, 17, 21, and 25 are the required function overloads. Lines 13 and 17 take a non const and const lvalue object, lines 21 and 25 a non const and const rvalue object. Deducing this in line 7 enables it to deduplicate the four overloads in one member function, that perfectly forwards `self` (line 9) and calls `implicitCall`



Deduplication of member function overloading

8.1.1.1.2 Reference a Lambda

The crucial idea of the Visitor Pattern[3] is performing operations on an object hierarchy. The object hierarchy is stable, but the operations may change frequently. The Overload Pattern[4] represents the modern C++ version of the Visitor Pattern. It combines variadic templates with `std:variant`[5] and its function `std::visit`[6]. Thanks to explicit object parameters in C++23, a lambda expression can explicitly reference its implicit lambda object.

The Overload Pattern

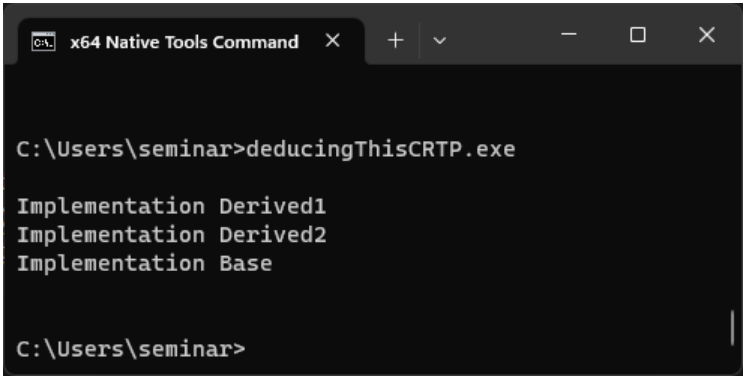
```
1  // deducingThisVisitor.cpp
2
3  #include <iostream>
4  #include <string>
5  #include <vector>
6  #include <variant>
7
8  template<class... Ts> struct overloaded : Ts... {
9      using Ts::operator()...;
10 };
11
12 class Wheel {
13  public:
14     Wheel(const std::string& n): name(n) { }
15     std::string getName() const {
16         return name;
17     }
18  private:
```

[3] https://www.modernescpp.com/index.php/the-visitor-pattern
[4] https://www.modernescpp.com/index.php/visiting-a-std-variant-with-the-overload-pattern
[5] https://en.cppreference.com/w/cpp/utility/variant
[6] https://en.cppreference.com/w/cpp/utility/variant/visit

```
19      std::string name;
20  };
21
22  class Body {};
23
24  class Engine {};
25
26  class Car;
27
28  using CarElement = std::variant<Wheel, Body, Engine, Car>;
29
30  class Car {
31   public:
32      Car(std::initializer_list<CarElement*> carElements ):
33        elements{carElements} {}
34
35     template<typename T>
36     void visitCarElements(T&& visitor) const {
37         for (auto elem : elements) {
38             std::visit(visitor, *elem);
39         }
40     }
41   private:
42      std::vector<CarElement*> elements;
43  };
44
45  overloaded carElementPrintVisitor {
46      [](const Body& body)     {  std::cout << "Visiting body" << '\n'; },
47      [](this auto const& self, const Car& car)  {  car.visitCarElements(self);
48                                                    std::cout << "Visiting car" << '\n'; },
49      [](const Wheel& wheel)   {  std::cout << "Visiting "
50                                             << wheel.getName() << " wheel" << '\n'; },
51      [](const Engine& engine) {  std::cout << "Visiting engine" << '\n';}
52  };
53
54  overloaded carElementDoVisitor {
55      [](const Body& body)     {  std::cout << "Moving my body" << '\n'; },
56      [](this auto const& self, const Car& car) {  car.visitCarElements(self);
57                                                   std::cout << "Starting my car" << '\n'; },
58      [](const Wheel& wheel)   {  std::cout << "Kicking my "
59                                             << wheel.getName()  << " wheel" << '\n'; },
60      [](const Engine& engine) {  std::cout << "Starting my engine" << '\n';}
61  };
62
63
```

```
64  int main() {
65
66      std::cout << '\n';
67
68      CarElement wheelFrontLeft  = Wheel("front left");
69      CarElement wheelFrontRight = Wheel("front right");
70      CarElement wheelBackLeft   = Wheel("back left");
71      CarElement wheelBackRight  = Wheel("back right");
72      CarElement body            = Body{};
73      CarElement engine          = Engine{};
74
75      CarElement car  = Car{&wheelFrontLeft, &wheelFrontRight,
76              &wheelBackLeft, &wheelBackRight,
77              &body, &engine};
78
79      std::visit(carElementPrintVisitor, engine);
80      std::visit(carElementPrintVisitor, car);
81      std::cout << '\n';
82
83      std::visit(carElementDoVisitor, engine);
84      std::visit(carElementDoVisitor, car);
85      std::cout << '\n';
86
87  }
```

The `car` (line 75) stands for the object hierarchy, and the two operations visit that is `carElementPrintVisitor` (line 45) and `carElementDoVistor` (line 54). Deducing this enables the car-visiting lambda expressions in lines 47 and 58 to reference the implicit lambda object to visit the components of the car: `car.visitCarElement(self)`.



The Overload Pattern

8.1.1.1.3 Simplifying CRTP

But what does CRTP mean? The acronym CRTP stands for the C++ idiom **C**uriously **R**ecurring **T**emplate **P**attern and means a technique in C++ in which a class `Derived` derives from a class template `Base`. The crucial point is that `Base` has `Derived` as a template argument.

```
template <typename T>
class Base{
    ...
};

class Derived : public Base<Derived>{
    ...
};
```

CRTP is typically used to implement polymorphism at compile time. For further information, please read my post More about Dynamic and Static Polymorphism[7].

Thanks to the explicit object parameter, I can remove the C and the R from the acronym CRTP.

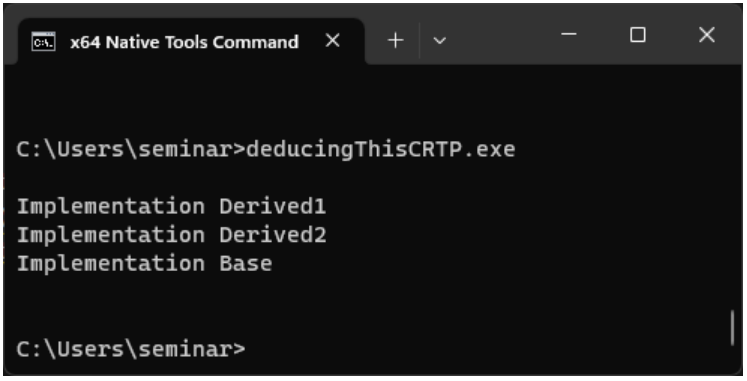
Simplifying CRTP

```
1  // deducingThisCRTP.cpp
2
3  #include <iostream>
4
5  struct Base{
6      template <typename Self>
7      void interface(this Self&& self){
8          std::forward<Self>(self).implementation();
9      }
10     void implementation(){
11         std::cout << "Implementation Base\n";
12   }
13 };
14
15 struct Derived1: Base {
16     void implementation(){
17         std::cout << "Implementation Derived1\n";
18     }
19 };
20
21 struct Derived2: Base {
22     void implementation(){
```

[7] https://www.modernescpp.com/index.php/more-about-dynamic-and-static-polymorhism

```
23          std::cout << "Implementation Derived2\n";
24      }
25  };
26
27  struct Derived3: Base {};
28
29  template <typename T>
30  void execute(T& base){
31      base.interface();
32  }
33
34
35  int main(){
36
37      std::cout << '\n';
38
39      Derived1 d1;
40      execute(d1);
41
42      Derived2 d2;
43      execute(d2);
44
45      Derived3 d3;
46      execute(d3);
47
48      std::cout << '\n';
49
50  }
```

Excplit object parameters enable deducing the derived type and perfectly forwarding it (line 7). The concrete type in line 32, `Derived1` (line 39), `Derived2` (line 42), and `Derived3` (line 45) is used. Consequentially, the corresponding *virtual* function `implementation` is called: `std::forward<Self>(self).implementation(`



simplifying CRTP

8.1.2 The Standard Library

8.1.2.1 Ranges Extensions

The ranges library got powerful extensions in C++23. This extension includes a convenient way to construct a container, various new views, and with `std::generator` the first concrete coroutine.

8.1.2.1.1 `ranges::to`

`std::ranges::to` is a convenient way to construct a container from a range:

```
std::vector<int> range(int begin, int end, int stepsize = 1) {
    auto boundary = [end](int i){ return i < end; };
    std::vector<int> result = std::ranges::views::iota(begin)
                            | std::views::stride(stepsize)
                            | std::views::take_while(boundary)
                            | std::ranges::to<std::vector>();
    return result;
}
```

The function `range` creates a `std::vector<int>` consisting of all elements from `begin` to `end` with the stepsize `stepsize`. `begin` must be smaller than `end`. Thanks to `std::ranges_to`, the element can be directly pushed into the `std::vector`.

8.1.2.1.2 New Views

C++23 supports additional views:

Views in C++23

View	Description
`std::ranges::zip_view` `std::views::zip`	Creates a view of tuples.
`std::ranges::zip_transform_view` `std::views::zip_transform`	Creates a view of tuples by applying the transformation function.
`std::ranges::adjacent_view` `std::views::adjacent`	Creates a view of adjacent elements.
`std::ranges::adjacent_transform_view` `std::views::adjacent_transform`	Creates a view of adjacent elements by applying the transformation function.
`std::ranges::join_with_view` `std::views::join_with`	Joins existing ranges into a view by applying a delimiter.
`std::ranges::slide_view` `std::views::slide`	Creates N-tuples by taking a view and a number N.
`std::ranges::chunk_view` `std::views::chunk`	Creates N-chunks of a view and a number N.
`std::ranges::chunk_by_view` `std::views::chunk_by`	Creates chunks of a view based on a predicate.
`std::ranges::as_const_view` `std::views::as_const`	Converts a view into a constant range.
`std::ranges::as_rvalue_view` `std::views::as_rvalue`	Casts each element into an rvalue.
`std::ranges::stride_view` `std::views::stride`	Creates a view of the N-th elements of another view.

The following code snippet applies the new views.

New views in C++23

```
// cpp23Ranges.cpp
...
#include <ranges>
...

std::vector vec = {1, 2, 3, 4};

for (auto i : vec | std::views::adjacent<2>) {
  std::cout << '(' << i.first << ", " << i.second << ") "; // (1, 2) (2, 3) (3, 4)
}

for (auto i : vec | std::views::adjacent_transform<2>(std::multiplies())) {
  std::cout << i << ' ';                                  // 2 6 12
}

std::print("{}\n", vec | std::views::chunk(2));        // [[1, 2], [3, 4],
std::print("{}\n", vec | std::views::slide(2));        // [[1, 2], [2, 3], [3, 4]

for (auto i : vec | std::views::slide(2)) {
    std::cout << '[' << i[0] << ", " << i[1] << "] ";  // [1, 2] [2, 3] [3, 4] [4, 5]
}

std::vector vec2 = {1, 2, 3, 0, 5, 2};
std::print("{}\n", vec2 | std::views::chunk_by(std::ranges::less_equal{}));
                                                       // [[1, 2, 3], [0, 5], [2]]

for (auto i : vec | std::views::slide(2)) {
    std::cout << '[' << i[0] << ", " << i[1] << "] ";  // [1, 2] [2, 3] [3, 4] [4, 5]
}
```

8.1.2.1.3 `std::generator`

`std::generator` in C++23 is the first concrete `coroutine` generator. A `std::generator` generates a sequence of elements by repeatedly resuming the coroutine. The sequence of elements can be infinite.

The coroutine generator `std::generator`

```
// generator.cpp
...
#include <generator>
#include <ranges>

...

std::generator<int> fib() {
    co_yield 0;                                 // 1
    auto a = 0;
    auto b = 1;
    for(auto n : std::views::iota(0)) {
        auto next = a + b;
        a = b;
        b = next;
        co_yield next;                          // 2
    }
}

...

for (auto f : fib() | std::views::take(10)) {  // 3
    std::cout << f << " ";                      //  0 1 1 2 3 5 8 13 21 34
}
```

The function `fib` return a coroutine. This coroutine creates an infinite stream of Fibonacci numbers. The stream of numbers starts with 0 (1) and continues with the following Fibonacci number (2). The ranges-based for-loop requests explicitly the first 10 Fibonacci numbers (3).

8.1.2.2 Modularized Standard Library

C++23 has a modularized standard library. `import std` imports the entire standard library. Consequentially, you have to rewrite your hello world program in C++23.

```
import std;

int main() {
    std::print("Hello world!);      // "Hello world!"
}
```

8.1.2.3 `std::print`, and `std::println`

The C++23 convenience functions `std::print` and `std::println` write to the output console. `std::println` adds a newline character to the output. Additionally, both functions enable it to write to an output stream and support Unicode[8]. You must include the header `<print>` or import the modularized standard library.

```
#include <print>                          // or import std;
...
std::print("{1} {0}!", "world", "Hello");  // prints "Hello world!"

std::ofstream outFile("testfile.txt");
std::print(outFile, "{1} {0}!", "world", "Hello");  // writes "Hello world!" into outFile
```

[8] https://en.wikipedia.org/wiki/Unicode

8.1.2.4 Formatting Ranges

C++23 enables it to format ranges.

Applying the format specification to the elements of a `std::vector`

```
1  // formatVector.cpp
2
3  #include <format>
4  #include <iostream>
5  #include <string>
6  #include <vector>
7
8  int main() {
9
10   std::vector<int> myInts{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11   std::cout << std::format("{}\n", myInts);
12   std::cout << std::format("{::+}\n", myInts);
13   std::cout << std::format("{::02x}\n", myInts);
14   std::cout << std::format("{::b}\n", myInts);
15
16   std::cout << '\n';
17
18   std::vector<std::string> myStrings{"Only", "for", "testing", "purpose"};
19   std::cout << std::format("{}\n", myStrings);
20   std::cout << std::format("{::.3}\n", myStrings);
21
22 }
```

By default (lines 11 and 19), the `std::vector` is displayed with surrounding square brackets. You can also apply format specifiers to the elements of the range by using an additional colons `:`.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[+1, +2, +3, +4, +5, +6, +7, +8, +9, +10]
[001, 002, 003, 004, 005, 006, 007, 008, 009, 010]
[1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010]

[Only, for, testing, purpose]
[Onl, for, tes, pur]
```

Applying the format specification to the elements of a `std::vector`

8.1.2.5 Container Adapters

The four associative containers `std::flat_map`, `std::flat_multimap`, `std::flat_set`, and `std::set_multiset` in C++23 are a drop-in replacement for the ordered associative containers[9] `std::map`, `std::multimap`, `std::set`, and `std::multiset`. More precisely, a `std::flat_map` is a drop-in replacement for a `std::map`, a `std::flat_multimap` is a drop-in replacement for a `std::multimap`, and so forth.

The flat-ordered associative containers require separate sequence containers[10] for their keys and values. These sequence containers must support a random access iterator. By default, a `std::vector`[11] is used, but a `std::array`[12], or a `std::deque`[13] is also valid.

The following code snippet shows the declaration of `std::flat_map`, and `std::flat_set`.

```
template<class Key, class T,
        class Compare = less<Key>,
        class KeyContainer = vector<Key>, class MappedContainer = vector<T>>
class flat_map;

template<class Key,
         class Compare = less<Key>,
         class KeyContainer = vector<Key>>
class flat_set;
```

The flat-ordered associative containers provide better time and space properties than the ordered associative containers. The flat variants require less memory and are faster to read than their non-flat-ordered pendants. The following comparison goes more into the details about the flat and the non-flat-ordered associative containers.

Comparison of the Flat Ordered Associative Container and their Non-Flat Pendants

The flat variants provide better reading performance, such as iterating through the container, and require less memory. They also need that the elements must either be copyable or moveable. The flat variants support a random access iterator[14]).

The non-flat variants improve writing performance if you insert or delete elements. Additionally, the non-flat variants guarantee that the iterators stay valid after inserting or deleting elements. The non-flat variants support a bidirectional iterator[15].

[9] https://en.cppreference.com/w/cpp/container
[10] https://en.cppreference.com/w/cpp/container
[11] https://en.cppreference.com/w/cpp/container/vector
[12] https://en.cppreference.com/w/cpp/container/array
[13] https://en.cppreference.com/w/cpp/container/deque
[14] https://en.cppreference.com/w/cpp/iterator/random_access_iterator
[15] https://en.cppreference.com/w/cpp/iterator/bidirectional_iterator

8.1.2.5.1 `std::sorted_unique`

You can use the constant `std::sorted_unique` in a constructor call or in the member functions `insert` to specify that the to be inserted elements are already sorted and unique. This improves the performance of creating a flat-ordered associative container or inserting elements.

The following code snippet creates a `std::flat_map` from a sorted initializer list `{1, 2, 3, 4, 5}`.

```
std::flat_map myFlatMap = { std::sorted_unique, {1, 2, 3, 4, 5}, {10, 11, 1, 5, -4} };
```

Using the constant `std::sorted_unique` with non-sorted or unique elements is undefined behavior.

8.1.2.6 `std::expected`

`std::expected<T, E>` provides a way to store either of two values. An instance of `std::expected` always holds a value: either the expected value of type `T`, or the unexpected value of type `E`. This vocabulary type requires the header `<expected>`. Thanks to `std::expected`, you can implement functions that either return a value or an error. The stored value is allocated directly within the storage occupied by the expected object. No dynamic memory allocation takes place.

`std::expected` has a similar interface such as `std::optional`. In contrast to `std::optional`, `std::exptected` can return an error message.

The various constructors let you define an expected object `exp` with an expected value. `exp.emplace` will construct the contained value in-place. You can explicitly ask a `std::expected` if it has a value, or you can check it in a logical expression. `exp.value` returns the expected value, and `exp.value_or` returns the expected value, or a default value. If `exp` has an unexpected value, the call `exp.value` will throw a `std::bad_expected_access` exception.

`std::unexpected` represents the unexpected value stored in `std::expected`.

std::expected

```
// expected.cpp

#include <iostream>
#include <expected>
#include <vector>
#include <string>

std::expected<int, std::string> getInt(std::string arg) {
    try {
        return std::stoi(arg);
    }
    catch (...) {
        return std::unexpected{std::string(arg + ": Error")};
    }
}


int main() {

    std::vector<std::string> strings = {"66", "foo", "-5"};

    for (auto s: strings) {
        auto res = getInt(s);
        if (res) {
            std::cout << res.value() << ' ';     // 66 -5
        }
```

```
27          else {
28              std::cout << res.error() << ' ';     // foo: Error
29          }
30      }
31
32      std::cout << '\n';
33
34      for (auto s: strings) {
35          auto res = getInt(s);
36          std::cout << res.value_or(2023) << ' ';  // 66 2023 -5
37      }
38
39  }
```

The function `getInt` converts each string to an integer and returns a `std::expected<int, std::string>`. `int` represents the expected, and `std::string` the unexpected value. The two range-based for-loops (lines 22 and 34) iterate through the `std::vector<std::string>`. In the first range-based for-loop (line 22), the expected (line 25) or the unexpected value (line 28) is displayed. In the second range-based for-loop (line 34), either the expected or the default value `2023` (line 36) is displayed.

`std::exptected` supports monadic operations for convenient function composition: `exp.and_then`, `exp.transform`, `exp.or_else`, and `exp.transform_error`. `exp.and_then` returns the result of the given function call if it exists, or an empty `std::expected`. `exp.transform` returns a `std::expected` containing its transformed value, or an empty `std::expected`. Additionally, `exp.or_else` returns the `std::expected` if it contains a value or the result of the given function otherwise.

Monadic operations on `std::expected`

```
// expectedMonadic.cpp
...
#include <expected>

std::expected<int, std::string> getInt(std::string arg) {
    try {
        return std::stoi(arg);
    }
    catch (...) {
        return std::unexpected{std::string(arg + ": Error")};
    }
}

std::vector<std::string> strings = {"66", "foo", "-5"};

for (auto s: strings) {
    auto res = getInt(s)
```

```
18                .transform( [](int n) { return n + 100; })
19                .transform( [](int n) { return std::to_string(n); });
20    std::cout << *res << ' ';        // 166 foo: Error 95
21 }
```

The range-based for-loop (line 23) iterates through the `std::vector<std::string>`. First, the function `getInt` converts each string to an integer (line 24), adds `100` to it (line 25), converts it back to a string (line 26), and finally displays the string (line 27). If the initial conversion to `int` fails, the string `arg + ": Error"` is returned (line 14) and displayed.

8.1.2.7 Multidimensional Access

A `std::mdspan` is a non-owning multidimensional view of a contiguous sequence of objects. Often, this multidimensional view is called a multidimensional array. The contiguous sequence of objects can be a plain C-array, a pointer with a length, a `std::array`[16], a `std::vector`[17], or a `std::string`[18].

The number of dimensions and the size of each dimension determine the shape of the multidimensional array. The number of dimensions is called rank, and the size of each dimension extension. The size of the `std::mdspan` is the product of all dimensions that are not 0. You can access the elements of a `std::mdspan` using the multidimensional index operator `[]`.

Each dimension of a `std::mdspan` can have a *static extent* or a *dynamic extent*. *static extent* means that its length is specified at compile time; *dynamic extent* accordingly that its length is specified at run time.

Definition of `std::mdspan`

```
template<
    class T,
    class Extents,
    class LayoutPolicy = std::layout_right,
    class AccessorPolicy = std::default_accessor<T>
> class mdspan;
```

- `T`: the contiguous sequence of objects
- `Extents`: specifies the number of dimensions as their size; each dimension can have a *static extent* or a *dynamic extent*
- `LayoutPolicy`: specifies the layout policy to access the underlying memory
- `AccessorPolicy`: specifies how the underlying elements are referenced

Thanks to class template argument deduction (CTAG)[19] in C++17, the compiler can often automatically deduce the template arguments.

[16] https://en.cppreference.com/w/cpp/container/array
[17] https://en.cppreference.com/w/cpp/container/vector
[18] https://en.cppreference.com/w/cpp/string/basic_string
[19] https://en.cppreference.com/w/cpp/language/class_template_argument_deduction

Two 2-dimensional arrays

```
1  // mdspan.cpp
2
3  #include <mdspan>
4  #include <iostream>
5  #include <vector>
6
7  int main() {
8
9      std::vector myVec{1, 2, 3, 4, 5, 6, 7, 8};
10
11     std::mdspan m{myVec.data(), 2, 4};
12     std::cout << "m.rank(): " << m.rank() << '\n';
13
14     for (std::size_t i = 0; i < m.extent(0); ++i) {
15         for (std::size_t j = 0; j < m.extent(1); ++j) {
16             std::cout << m[i, j] << ' ';
17         }
18         std::cout << '\n';
19     }
20
21     std::cout << '\n';
22
23     std::mdspan m2{myVec.data(), 4, 2};
24     std::cout << "m2.rank(): " << m2.rank() << '\n';
25
26     for (std::size_t i = 0; i < m2.extent(0); ++i) {
27         for (std::size_t j = 0; j < m2.extent(1); ++j) {
28         std::cout << m2[i, j] << ' ';
29     }
30     std::cout << '\n';
31   }
32
33 }
```

I apply class template argument deduction three times in this example. Line 9 uses it for a `std::vector`, and lines 11 and 23 for a `std::mdspan`. The first 2-dimensional array `m` has a shape of (2, 4), the second one `m2` a shape of (4, 2). Lines 12 and 24 display the ranks of both `std::mdspan`. Thanks to the extent of each dimension (lines 14 and 15), and the index operator in line 16, it is straightforward to iterate through multidimensional arrays.

```
m.rank(): 2
1 2 3 4
5 6 7 8

m2.rank(): 2
1 2
3 4
5 6
7 8
```

Two 2-dimensional arrays

If your multidimensional array should have a *static extent*, you have to specify the template arguments.

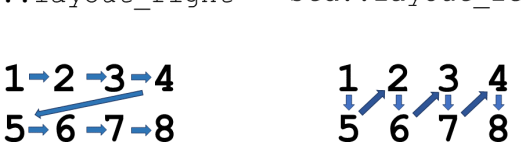
Explicitly specifying the template arguments of a `std::mdspan`

```
// staticDynamicExtent.cpp

#include <mdspan>
...

std::mdspan<int, std::extents<std::size_t, 2, 4>> m{myVec.data()};  // (1)
std::cout << "m.rank(): " << m.rank() << '\n';

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

std::mdspan<int, std::extents<std::size_t, std::dynamic_extent, std::dynamic_extent>>
            m2{myVec.data(), 4, 2};                                  // (2)
std::cout << "m2.rank(): " << m2.rank() << '\n';

for (std::size_t i = 0; i < m2.extent(0); ++i) {
    for (std::size_t j = 0; j < m2.extent(1); ++j) {
        std::cout << m2[i, j] << ' ';
    }
std::cout << '\n';
}
```

The program `staticDynamicExtent.cpp` is based on the previous program `mdspan.cpp`, and produces the same output. The difference is, that the `std::mdspan m` (1) has a *static extent*. For completeness,

`std::mdspan m2` (2) has a *dynamic extent*. Consequentially, the shape of `m` is specified with template arguments, but the shape of `m2` is with function arguments.

A `std::mdspan` allows you to specify the layout policy to access the underlying memory. By default, `std::layout_right` (C, C++ or Python[20] style) is used, but you can also specify `std::layout_left` (Fortran[21] or MATLAB[22] style). The following graphic exemplifies in which sequence the elements of the `std::mdspan` are accessed.



`std::layout_right` and `std::layout_left`

Traversing two `std::mdspan` with the layout policy `std::layout_right` and `std::layout_left` shows the difference.

Using a `std::mdspan` with `std::layout_right` and `std::layout_left`

```
1  // mdspanLayout.cpp
2  ...
3  #include <mdspan>
4
5  std::vector myVec{1, 2, 3, 4, 5, 6, 7, 8};
6
7  std::mdspan<int,
8      std::extents<std::size_t, std::dynamic_extent, std::dynamic_extent>,
9      std::layout_right> m2{myVec.data(), 4, 2};                    // (1)
10
11 std::cout << "m.rank(): " << m.rank() << '\n';
12
13 for (std::size_t i = 0; i < m.extent(0); ++i) {
14     for (std::size_t j = 0; j < m.extent(1); ++j) {
15         std::cout << m[i, j] << ' ';
16     }
17     std::cout << '\n';
18 }
19
20 std::cout << '\n';
21
22 std::mdspan<int,
```

[20] https://en.wikipedia.org/wiki/Python_(programming_language)
[21] https://en.wikipedia.org/wiki/Fortran
[22] https://en.wikipedia.org/wiki/MATLAB

```
23      std::extents<std::size_t, std::dynamic_extent, std::dynamic_extent>,
24      std::layout_left> m2{myVec.data(), 4, 2};                // (2)
25  std::cout << "m2.rank(): " << m2.rank() << '\n';
26
27  for (std::size_t i = 0; i < m2.extent(0); ++i) {
28      for (std::size_t j = 0; j < m2.extent(1); ++j) {
29          std::cout << m2[i, j] << ' ';
30      }
31      std::cout << '\n';
32  }
```

The `std::mdspan m` uses `std::layout_right` (1), the other `std::mdspan std::layout_left` (1). Thanks to class template argument deduction, the constructor call of `std::mdspan` (1) needs no explicit template arguments and is much easier to write: `std::mdspan m2{myVec.data(), 4, 2}`.

The output of the program shows the two different layout strategies.

```
m.rank(): 2
1 2
3 4
5 6
7 8

m2.rank(): 2
1 5
2 6
3 7
4 8
```

`std::mdspan` with `std::layout_left` and `std::layout_right`

The following table presents an overview of `std::mdspan`'s interface.

Functions of a `std::mdspan md`

Function	Description
`md[ind]`	Access the `ind`-th element.
`md.size`	Returns the size of the multidimensional array.
`md.rank`	Returns the dimension of the multidimensional array.
`md.extents(i)`	Returns the size of the `i`-th dimension.
`md.data_handle`	Returns a pointer to the contiguous sequence of elements.

8.2 Beyond C++23

"Prediction is very difficult, especially if it's about the future." (Niels Bohr[23]). Consequently, you should read this chapter as my best attempt to predict the C++ future.

It's highly likely that the four features reflection, pattern matching, and contracts are successively added to the C++ standard, starting with C++26.

8.2.1 Contracts

Contracts were planned to be the fifth great feature of C++20. Because of design issues, they were removed in the standardization committee meeting in July 2019 in Cologne. At the same time, the study group 21 for contracts[24] was created.

- What is a Contract?

A contract specifies in a precise and checkable way interfaces for software components. These software components are typically functions and member functions that have to fulfill preconditions, postconditions, or invariants. Here are the simplified definitions of these three terms:

- A **precondition**: a predicate that is supposed to hold upon entry into a function
- A **postcondition**: a predicate that is supposed to hold upon exit from the function
- An **assertion**: a predicate that is supposed to hold at its point in the computation

The precondition and the postcondition are placed outside the function definition, but the invariant (assertion) is placed inside. A predicate is a function which returns a boolean.

Here is a first example:

The function `push` uses contracts

```
int push(queue& q, int val)
    [[ expects: !q.full() ]]
    [[ ensures !q.empty() ]] {
    ...
    [[ assert: q.is_ok() ]]
    ...
}
```

The attribute `expects` is a precondition, the attribute `ensures` a postcondition, and the attribute `assert` an assertion. The contracts for the function `push` are that the queue is not full before adding an element, that it is not empty after adding and the assertion `q.is_ok()` holds.

Preconditions and postconditions are part of the function interface. That means they can't access local members of a function or private or protected members of a class. Assertions, however, are part of the implementation and can, therefore, access local members of a function of private or protected members of a class:

[23] https://www.goodreads.com/quotes/23796-prediction-is-very-difficult-especially-about-the-future
[24] https://isocpp.org/std/the-committee

Accessing a private attribute

```
class X {
public:
    void f(int n)
        [[ expects: n < m ]]  // error; m is private
    {
        [[ assert: n < m ]];  // OK
        // ...
    }
private:
    int m;
};
```

The attribute `m` is private and cannot, therefore, be part of a precondition. By default, a violation of a contract terminates the program.

You can adjust the behavior of the attributes.

8.2.1.1 Fine-tune Attributes

The syntax for adapting the attributes is quite elaborate: **[[contract-attribute modifier: conditional-expression]]**.

- **contract-attribute**: `expects`, `ensures`, and `assert`
- **modifier**: specifies the contract level or the enforcement of the contract; possible values are `default`, `audit`, and `axiom`
  - **default**: the cost of run-time checking should be small; it is the default modifier
  - **audit**: the cost of run-time checking is assumed to be large
  - **axiom**: the predicate is not checked at run time
- **conditional-expression**: the predicate of the contract

For the `ensures` attribute, there is additionally an identifier available: **[[ensures modifier identifier: conditional-expression]]**

The identifier lets you refer to the return value of the function.

Accessing the return value

```
int mul(int x, int y)
    [[expects: x > 0]]         // implicit default
    [[expects default: y > 0]]
    [[ensures audit res: res > 0]] {
    return x * y;
}
```

`res` as the identifier is an arbitrary name. As shown in the example, you can use more contracts of the same kind.

Let me dive deeper into the handling of contract violations.

8.2.1.2 Handling Contract Violations

A compilation has three assertion build levels:

- `off`: no contracts are checked
- `default`: default contracts are checked; this is the default
- `audit`: default and audit contracts are checked

When a contract violation occurs because the predicate returns `false`, the violation handler is invoked. The violation handler gets a value of type `std::contract_violation`. This value provides detailed information about the violation of the contract.

The class `contract_violation`

```
namespace std {
    class contract_violation{
     public:
        uint_least32_t line_number() const noexcept;
        string_view file_name() const noexcept;
        string_view function_name() const noexcept;
        string_view comment() const noexcept;
        string_view assertion_level() const noexcept;
    };
}
```

- `line_number`: the line number of the contract violation
- `file_name`: the file name of the contract violation
- `function_name`: the function name of the contract violation
- `comment`: the predicate of the contract
- `assertion_level`: the assertion level of the contract

8.2.1.3 Declaration of Contracts

A contract can be placed on the declaration of a function. This includes declarations of virtual functions or function templates.

- The contract declaration of a function must be identical. Any declaration different from the first one can omit the contract.

Conctract declarations must be idential

```
int f(int x)
    [[expects: x > 0]]
    [[ensures r: r > 0]];

int f(int x); // OK. No contract.

int f(int x)
    [[expects: x >= 0]]; // Error missing ensures and different expects condition
```

- A contract cannot be modified in an overriding function.

Overriding functions cannot modify a contract

```
struct B {
    virtual void f(int x)[[expects: x > 0]];
    virtual void g(int x);
};

struct D: B{
    void f(int x)[[expects: x >= 0]];   // error
    void g(int x)[[expects: x != 0]];   // error
};
```

Both contract definitions of class `D` are erroneous. The contract of the member function `D::f` differs from the one from `B::f`. The member function `D::g` adds a contract to `B::g`.

Closing Thoughts from Herb Sutter

Contracts were planned to be part of C++20 but were delayed. Herb Sutter's thoughts on Sutter's Mill[25] give you an idea about their importance: "*contracts is the most impactful feature of C++20 so far, and arguably the most impactful feature we have added to C++ since C++11.*"

[25] https://herbsutter.com/2018/07/02/trip-report-summer-iso-c-standards-meeting-rapperswil/

8.2.2 Reflection

Reflection is the possibility of a program to analyze and modify itself. Reflection takes place at compile time and, therefore, adheres to the C++ metarule: "don't pay for anything you don't use". The type-traits library[26] is a powerful tool for reflection, but the proposal P0385[27] for static reflection goes much further.

The following code snippet should give you a first impression of reflection:

The reflection operator

```
1  template <typename T>
2  T min(constT& a,constT& b) {
3      log() << "function: min<"
4            << get_base_name_v<get_aliased_t<$reflect(T)>>
5            << ">("
6            << get_base_name_v<$reflect(a)> << ": "
7            << get_base_name_v<get_aliased_t<get_type_t<$reflect(a)>>>
8            << " = " << a << ", "
9            << get_base_name_v<$reflect(b)> << ": "
10           << get_base_name_v<get_aliased_t<get_type_t<$reflect(b)>>>
11           << " = " << b
12           << ")" << '\n';
13     return a < b ? a : b;
14 }
```

The new reflection operator `$reflect` is the crucial expression in the example. First, the new operator creates a special data type, which provides meta information on the template parameter `T` (line 4) and the values `a` (line 6), and `c` (line 9). Thanks to function composition, the metainformation can be used to provide more information: `get_base_name_v<get_aliased_t ...` (lines 7 and 10).

When you invoke the function `min` with the argument `min(12.34, 23.45)`, you get the following output:

```
function: min<double>(a: double = 12.34, b: double = 23.45)
```

Calling `min(12.34, 23.45)`

You may be curious and want to know: Which metainformation could you get with reflection? The following points give you the answer:

- Objects: the source-code line and column and the name of the file
- Classes: the private and public data members and member functions
- Aliases: the name of the resolved alias

The next example from proposal P0385 shows how reflection helps determine the private and public members of a class.

[26] https://en.cppreference.com/w/cpp/header/type_traits
[27] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0385r2.pdf

Determining the public and private members of the class `foo`

```
#include <reflect>
#include <iostream>

struct foo {
 private:
    int _i, _j;
 public:
    static constexpr const bool b = true;
    float x, y, z;
 private:
    static double d;
};

template <typename ... T>
void eat(T ... ) { }

template <typename Metaobjects, std::size_t I>
int do_print_data_member(void) {
    using namespace std;
    typedef reflect::get_element_t<Metaobjects, I> metaobj;
    cout << I << ": "
        << (reflect::is_public_v<metaobj>?"public":"non-public")
        << " "
        << (reflect::is_static_v<metaobj>?"static":"")
        << " "
        << reflect::get_base_name_v<reflect::get_type_t<metaobj>>
        << " "
        << reflect::get_base_name_v<metaobj>
        << '\n';
    }
    return 0;

    template <typename Metaobjects, std::size_t ... I>
    void do_print_data_members(std::index_sequence<I...>) {
        eat(do_print_data_member<Metaobjects, I>()...);
    }

    template <typename Metaobjects>
    void do_print_data_members(void) {
        using namespace std;

        do_print_data_members<Metaobjects>(
            make_index_sequence<
                reflect::get_size_v<Metaobjects>
```

```
            >()
        );
    }

    template <typename MetaClass>
    void print_data_members(void) {
        using namespace std;

        cout << "Public data members of " << reflect::get_base_name_v<MetaClass>
             << '\n';

        do_print_data_members<reflect::get_public_data_members_t<MetaClass>>();
    }

    template <typename MetaClass>
    void print_all_data_members(void) {
        using namespace std;

        cout << "All data members of " << reflect::get_base_name_v<MetaClass>
             << '\n';
        do_print_data_members<reflect::get_data_members_t<MetaClass>>();
    }

int main(void) {
    print_data_members<$reflect(foo)>();
    print_all_data_members<$reflect(foo)>();
    return 0;
}
```

The program produces the following output:

```
Public data members of foo
0: public static bool b
1: public  float x
2: public  float y
3: public  float z
All data members of foo
0: non-public  int _i
1: non-public  int _j
2: public static bool b
3: public  float x
4: public  float y
5: public  float z
6: non-public static double d
```

Displaying the public and private members of the class `foo`

8.2.3 Pattern Matching

New data types such as `std::tuple`[28] or `std::variant`[29] need new ways to work with their elements. Simple `if` or `switch` conditions or functions like `std::apply`[30] or `std::visit`[31] can only provide basic functionality. Pattern matching, heavily used in functional programming, allows the more efficient handling of the new data types.

The following code snippets from the proposal P1371R2[32] on pattern matching compare classical control structures with pattern matching. Pattern matching uses the keyword `inspect` and __ for a placeholder.

- `switch` statement

`switch` statement versus pattern matching

```
switch (x) {
    case 0: std::cout << "got zero"; break;
    case 1: std::cout << "got one"; break;
    default: std::cout << "don't care";
}

inspect (x) {
    0: std::cout << "got zero";
    1: std::cout << "got one";
    __: std::cout << "don't care";
}
```

- `if` condition

`if` statement versus pattern matching

```
if (s == "foo") {
    std::cout << "got foo";
} else if (s == "bar") {
    std::cout << "got bar";
} else {
    std::cout << "don't care";
}

inspect (s) {
    "foo": std::cout << "got foo";
    "bar": std::cout << "got bar";
    __: std::cout << "don't care";
}
```

[28] https://en.cppreference.com/w/cpp/utility/tuple
[29] https://en.cppreference.com/w/cpp/utility/variant
[30] https://en.cppreference.com/w/cpp/utility/apply
[31] https://en.cppreference.com/w/cpp/utility/variant/visit
[32] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1371r2.pdf

The application of pattern matching on `std::tuple`, `std::variant`, or polymorphy demonstrates its power.

- `std::tuple`

`std::tuple` versus pattern matching

```
auto&& [x, y] = p;
if (x == 0 && y == 0) {
    std::cout << "on origin";
} else if (x == 0) {
    std::cout << "on y-axis";
} else if (y == 0) {
    std::cout << "on x-axis";
} else {
    std::cout << x << ',' << y;
}

inspect (p) {
    [0, 0]: std::cout << "on origin";
    [0, y]: std::cout << "on y-axis";
    [x, 0]: std::cout << "on x-axis";
    [x, y]: std::cout << x << ',' << y;
}
```

- `std::variant`

`std::variant` versus pattern matching

```
struct visitor {
    void operator()(int i) const {
        os << "got int: " << i;
    }
    void operator()(float f) const {
        os << "got float: " << f;
    }
    std::ostream& os;
};
std::visit(visitor{strm}, v);

inspect (v) {
    <int> i: strm << "got int: " << i;
    <float> f: strm << "got float: " << f;
}
```

- Polymorphic data types

Polymorphy versus pattern matching

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };

virtual int Shape::get_area() const = 0;

int Circle::get_area() const override {
    return 3.14 * radius * radius;
}
int Rectangle::get_area() const override {
    return width * height;
}

int get_area(const Shape& shape) {
    return inspect (shape) {
        <Circle> [r] => 3.14 * r * r,
        <Rectangle> [w, h] => w * h
    }
}
```

The proposal P1371R2 on pattern matching offers more advanced use cases. For example, pattern matching can be used to traverse an expression tree[33].

[33] https://en.wikipedia.org/wiki/Binary_expression_tree

9. Feature Testing

The header `<version>` allows you to ask your compiler for its C++11 or later support. You can ask for attributes, features of the core language, or the library. `<version>` has about 200 macros defined, which expand to a number when the feature is implemented. The number represents the year and month in which the feature was added to the C++ standard. These are the numbers for `static_assert`, lambdas, and concepts.

Macros for `static_assert`, lambdas, and concepts

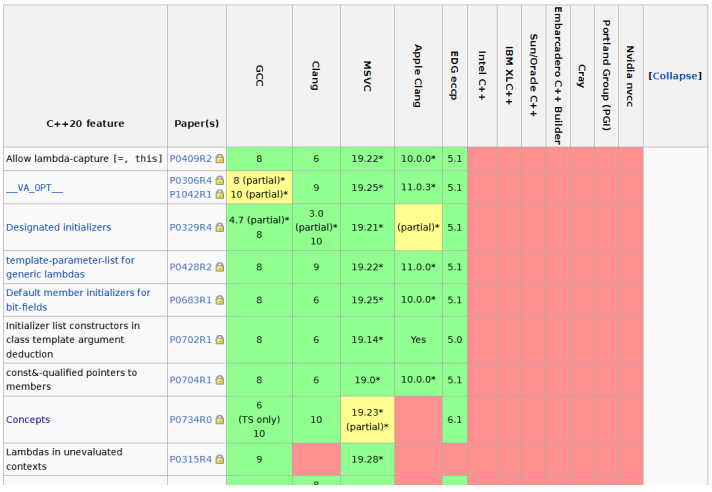
```
__cpp_static_assert  200410L
__cpp_lambdas  200907L
__cpp_concepts 201907L
```

Feature Support

When I experiment with brand-new C++ features, I check which compiler implements the feature I'm interested in. That's the time I visit cppreference.com/compiler_support[1], searching for the feature I want to try out, and hope that at least one compiler of the big three (GCC, Clang, MSVC) implements the new feature.

Getting the answer `partial` is not satisfying. In the end I don't know who I should contact when the compilation of a brand-new feature fails.

[1] https://en.cppreference.com/w/cpp/compiler_support



C++20 feature	Paper(s)	GCC	Clang	MSVC	Apple Clang	EDG eccp	Intel C++	IBM XL C++	Sun/Oracle C++	Embarcadero C++ Builder	Cray	Portland Group (PGI)	Nvidia nvcc	[Collapse]
Allow lambda-capture [=, this]	P0409R2	8	6	19.22*	10.0.0*	5.1								
__VA_OPT__	P0306R4 P1042R1	8 (partial)* 10 (partial)*	9	19.25*	11.0.3*	5.1								
Designated initializers	P0329R4	4.7 (partial)* 8	3.0 (partial)* 10	19.21*	(partial)*	5.1								
template-parameter-list for generic lambdas	P0428R2	8	9	19.22*	11.0.0*	5.1								
Default member initializers for bit-fields	P0683R1	8	6	19.25*	10.0.0*	5.1								
Initializer list constructors in class template argument deduction	P0702R1	8	6	19.14*	Yes	5.0								
const&-qualified pointers to members	P0704R1	8	6	19.0*	10.0.0*	5.1								
Concepts	P0734R0	6 (TS only) 10	10	19.23* (partial)*		6.1								
Lambdas in unevaluated contexts	P0315R4	9		19.28*										

Feature support for C++20 core language

The cppreference.com page for feature testing[2] uses all macros together in a long, long source file.

Use of all feature test macros

```
// featureTest.cpp
// from cppreference.com

#if __cplusplus < 201100
#  error "C++11 or better is required"
#endif

#include <algorithm>
#include <cstring>
#include <iomanip>
#include <iostream>
#include <string>

#ifdef __has_include
# if __has_include(<version>)
#   include <version>
# endif
#endif

#define COMPILER_FEATURE_VALUE(value) #value
#define COMPILER_FEATURE_ENTRY(name) { #name, COMPILER_FEATURE_VALUE(name) },

#ifdef __has_cpp_attribute
```

[2] https://en.cppreference.com/w/cpp/feature_test

```
# define COMPILER_ATTRIBUTE_VALUE_AS_STRING(s) #s
# define COMPILER_ATTRIBUTE_AS_NUMBER(x) COMPILER_ATTRIBUTE_VALUE_AS_STRING(x)
# define COMPILER_ATTRIBUTE_ENTRY(attr) \
  { #attr, COMPILER_ATTRIBUTE_AS_NUMBER(__has_cpp_attribute(attr)) },
#else
# define COMPILER_ATTRIBUTE_ENTRY(attr) { #attr, "_" },
#endif

// Change these options to print out only necessary info.
static struct PrintOptions {
    constexpr static bool titles              = 1;
    constexpr static bool attributes          = 1;
    constexpr static bool general_features    = 1;
    constexpr static bool core_features       = 1;
    constexpr static bool lib_features        = 1;
    constexpr static bool supported_features  = 1;
    constexpr static bool unsupported_features = 1;
    constexpr static bool sorted_by_value     = 0;
    constexpr static bool cxx11               = 1;
    constexpr static bool cxx14               = 1;
    constexpr static bool cxx17               = 1;
    constexpr static bool cxx20               = 1;
    constexpr static bool cxx23               = 0;
}   print;

struct CompilerFeature {
    CompilerFeature(const char* name = nullptr, const char* value = nullptr)
        : name(name), value(value) {}
    const char* name; const char* value;
};

static CompilerFeature cxx[] = {
COMPILER_FEATURE_ENTRY(__cplusplus)
COMPILER_FEATURE_ENTRY(__cpp_exceptions)
COMPILER_FEATURE_ENTRY(__cpp_rtti)
#if 0
COMPILER_FEATURE_ENTRY(__GNUC__)
COMPILER_FEATURE_ENTRY(__GNUC_MINOR__)
COMPILER_FEATURE_ENTRY(__GNUC_PATCHLEVEL__)
COMPILER_FEATURE_ENTRY(__GNUG__)
COMPILER_FEATURE_ENTRY(__clang__)
COMPILER_FEATURE_ENTRY(__clang_major__)
COMPILER_FEATURE_ENTRY(__clang_minor__)
COMPILER_FEATURE_ENTRY(__clang_patchlevel__)
#endif
```

```
};
static CompilerFeature cxx11[] = {
COMPILER_FEATURE_ENTRY(__cpp_alias_templates)
COMPILER_FEATURE_ENTRY(__cpp_attributes)
COMPILER_FEATURE_ENTRY(__cpp_constexpr)
COMPILER_FEATURE_ENTRY(__cpp_decltype)
COMPILER_FEATURE_ENTRY(__cpp_delegating_constructors)
COMPILER_FEATURE_ENTRY(__cpp_inheriting_constructors)
COMPILER_FEATURE_ENTRY(__cpp_initializer_lists)
COMPILER_FEATURE_ENTRY(__cpp_lambdas)
COMPILER_FEATURE_ENTRY(__cpp_nsdmi)
COMPILER_FEATURE_ENTRY(__cpp_range_based_for)
COMPILER_FEATURE_ENTRY(__cpp_raw_strings)
COMPILER_FEATURE_ENTRY(__cpp_ref_qualifiers)
COMPILER_FEATURE_ENTRY(__cpp_rvalue_references)
COMPILER_FEATURE_ENTRY(__cpp_static_assert)
COMPILER_FEATURE_ENTRY(__cpp_threadsafe_static_init)
COMPILER_FEATURE_ENTRY(__cpp_unicode_characters)
COMPILER_FEATURE_ENTRY(__cpp_unicode_literals)
COMPILER_FEATURE_ENTRY(__cpp_user_defined_literals)
COMPILER_FEATURE_ENTRY(__cpp_variadic_templates)
};
static CompilerFeature cxx14[] = {
COMPILER_FEATURE_ENTRY(__cpp_aggregate_nsdmi)
COMPILER_FEATURE_ENTRY(__cpp_binary_literals)
COMPILER_FEATURE_ENTRY(__cpp_constexpr)
COMPILER_FEATURE_ENTRY(__cpp_decltype_auto)
COMPILER_FEATURE_ENTRY(__cpp_generic_lambdas)
COMPILER_FEATURE_ENTRY(__cpp_init_captures)
COMPILER_FEATURE_ENTRY(__cpp_return_type_deduction)
COMPILER_FEATURE_ENTRY(__cpp_sized_deallocation)
COMPILER_FEATURE_ENTRY(__cpp_variable_templates)
};
static CompilerFeature cxx14lib[] = {
COMPILER_FEATURE_ENTRY(__cpp_lib_chrono_udls)
COMPILER_FEATURE_ENTRY(__cpp_lib_complex_udls)
COMPILER_FEATURE_ENTRY(__cpp_lib_exchange_function)
COMPILER_FEATURE_ENTRY(__cpp_lib_generic_associative_lookup)
COMPILER_FEATURE_ENTRY(__cpp_lib_integer_sequence)
COMPILER_FEATURE_ENTRY(__cpp_lib_integral_constant_callable)
COMPILER_FEATURE_ENTRY(__cpp_lib_is_final)
COMPILER_FEATURE_ENTRY(__cpp_lib_is_null_pointer)
COMPILER_FEATURE_ENTRY(__cpp_lib_make_reverse_iterator)
COMPILER_FEATURE_ENTRY(__cpp_lib_make_unique)
COMPILER_FEATURE_ENTRY(__cpp_lib_null_iterators)
```

```
114 COMPILER_FEATURE_ENTRY(__cpp_lib_quoted_string_io)
115 COMPILER_FEATURE_ENTRY(__cpp_lib_result_of_sfinae)
116 COMPILER_FEATURE_ENTRY(__cpp_lib_robust_nonmodifying_seq_ops)
117 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_timed_mutex)
118 COMPILER_FEATURE_ENTRY(__cpp_lib_string_udls)
119 COMPILER_FEATURE_ENTRY(__cpp_lib_transformation_trait_aliases)
120 COMPILER_FEATURE_ENTRY(__cpp_lib_transparent_operators)
121 COMPILER_FEATURE_ENTRY(__cpp_lib_tuple_element_t)
122 COMPILER_FEATURE_ENTRY(__cpp_lib_tuples_by_type)
123 };
124
125 static CompilerFeature cxx17[] = {
126 COMPILER_FEATURE_ENTRY(__cpp_aggregate_bases)
127 COMPILER_FEATURE_ENTRY(__cpp_aligned_new)
128 COMPILER_FEATURE_ENTRY(__cpp_capture_star_this)
129 COMPILER_FEATURE_ENTRY(__cpp_constexpr)
130 COMPILER_FEATURE_ENTRY(__cpp_deduction_guides)
131 COMPILER_FEATURE_ENTRY(__cpp_enumerator_attributes)
132 COMPILER_FEATURE_ENTRY(__cpp_fold_expressions)
133 COMPILER_FEATURE_ENTRY(__cpp_guaranteed_copy_elision)
134 COMPILER_FEATURE_ENTRY(__cpp_hex_float)
135 COMPILER_FEATURE_ENTRY(__cpp_if_constexpr)
136 COMPILER_FEATURE_ENTRY(__cpp_inheriting_constructors)
137 COMPILER_FEATURE_ENTRY(__cpp_inline_variables)
138 COMPILER_FEATURE_ENTRY(__cpp_namespace_attributes)
139 COMPILER_FEATURE_ENTRY(__cpp_noexcept_function_type)
140 COMPILER_FEATURE_ENTRY(__cpp_nontype_template_args)
141 COMPILER_FEATURE_ENTRY(__cpp_nontype_template_parameter_auto)
142 COMPILER_FEATURE_ENTRY(__cpp_range_based_for)
143 COMPILER_FEATURE_ENTRY(__cpp_static_assert)
144 COMPILER_FEATURE_ENTRY(__cpp_structured_bindings)
145 COMPILER_FEATURE_ENTRY(__cpp_template_template_args)
146 COMPILER_FEATURE_ENTRY(__cpp_variadic_using)
147 };
148 static CompilerFeature cxx17lib[] = {
149 COMPILER_FEATURE_ENTRY(__cpp_lib_addressof_constexpr)
150 COMPILER_FEATURE_ENTRY(__cpp_lib_allocator_traits_is_always_equal)
151 COMPILER_FEATURE_ENTRY(__cpp_lib_any)
152 COMPILER_FEATURE_ENTRY(__cpp_lib_apply)
153 COMPILER_FEATURE_ENTRY(__cpp_lib_array_constexpr)
154 COMPILER_FEATURE_ENTRY(__cpp_lib_as_const)
155 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_is_always_lock_free)
156 COMPILER_FEATURE_ENTRY(__cpp_lib_bool_constant)
157 COMPILER_FEATURE_ENTRY(__cpp_lib_boyer_moore_searcher)
158 COMPILER_FEATURE_ENTRY(__cpp_lib_byte)
```

```
159 COMPILER_FEATURE_ENTRY(__cpp_lib_chrono)
160 COMPILER_FEATURE_ENTRY(__cpp_lib_clamp)
161 COMPILER_FEATURE_ENTRY(__cpp_lib_enable_shared_from_this)
162 COMPILER_FEATURE_ENTRY(__cpp_lib_execution)
163 COMPILER_FEATURE_ENTRY(__cpp_lib_filesystem)
164 COMPILER_FEATURE_ENTRY(__cpp_lib_gcd_lcm)
165 COMPILER_FEATURE_ENTRY(__cpp_lib_hardware_interference_size)
166 COMPILER_FEATURE_ENTRY(__cpp_lib_has_unique_object_representations)
167 COMPILER_FEATURE_ENTRY(__cpp_lib_hypot)
168 COMPILER_FEATURE_ENTRY(__cpp_lib_incomplete_container_elements)
169 COMPILER_FEATURE_ENTRY(__cpp_lib_invoke)
170 COMPILER_FEATURE_ENTRY(__cpp_lib_is_aggregate)
171 COMPILER_FEATURE_ENTRY(__cpp_lib_is_invocable)
172 COMPILER_FEATURE_ENTRY(__cpp_lib_is_swappable)
173 COMPILER_FEATURE_ENTRY(__cpp_lib_launder)
174 COMPILER_FEATURE_ENTRY(__cpp_lib_logical_traits)
175 COMPILER_FEATURE_ENTRY(__cpp_lib_make_from_tuple)
176 COMPILER_FEATURE_ENTRY(__cpp_lib_map_try_emplace)
177 COMPILER_FEATURE_ENTRY(__cpp_lib_math_special_functions)
178 COMPILER_FEATURE_ENTRY(__cpp_lib_memory_resource)
179 COMPILER_FEATURE_ENTRY(__cpp_lib_node_extract)
180 COMPILER_FEATURE_ENTRY(__cpp_lib_nonmember_container_access)
181 COMPILER_FEATURE_ENTRY(__cpp_lib_not_fn)
182 COMPILER_FEATURE_ENTRY(__cpp_lib_optional)
183 COMPILER_FEATURE_ENTRY(__cpp_lib_parallel_algorithm)
184 COMPILER_FEATURE_ENTRY(__cpp_lib_raw_memory_algorithms)
185 COMPILER_FEATURE_ENTRY(__cpp_lib_sample)
186 COMPILER_FEATURE_ENTRY(__cpp_lib_scoped_lock)
187 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_mutex)
188 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_ptr_arrays)
189 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_ptr_weak_type)
190 COMPILER_FEATURE_ENTRY(__cpp_lib_string_view)
191 COMPILER_FEATURE_ENTRY(__cpp_lib_to_chars)
192 COMPILER_FEATURE_ENTRY(__cpp_lib_transparent_operators)
193 COMPILER_FEATURE_ENTRY(__cpp_lib_type_trait_variable_templates)
194 COMPILER_FEATURE_ENTRY(__cpp_lib_uncaught_exceptions)
195 COMPILER_FEATURE_ENTRY(__cpp_lib_unordered_map_try_emplace)
196 COMPILER_FEATURE_ENTRY(__cpp_lib_variant)
197 COMPILER_FEATURE_ENTRY(__cpp_lib_void_t)
198 };
199
200 static CompilerFeature cxx20[] = {
201 COMPILER_FEATURE_ENTRY(__cpp_aggregate_paren_init)
202 COMPILER_FEATURE_ENTRY(__cpp_char8_t)
203 COMPILER_FEATURE_ENTRY(__cpp_concepts)
```

```
204 COMPILER_FEATURE_ENTRY(__cpp_conditional_explicit)
205 COMPILER_FEATURE_ENTRY(__cpp_consteval)
206 COMPILER_FEATURE_ENTRY(__cpp_constexpr)
207 COMPILER_FEATURE_ENTRY(__cpp_constexpr_dynamic_alloc)
208 COMPILER_FEATURE_ENTRY(__cpp_constexpr_in_decltype)
209 COMPILER_FEATURE_ENTRY(__cpp_constinit)
210 COMPILER_FEATURE_ENTRY(__cpp_deduction_guides)
211 COMPILER_FEATURE_ENTRY(__cpp_designated_initializers)
212 COMPILER_FEATURE_ENTRY(__cpp_generic_lambdas)
213 COMPILER_FEATURE_ENTRY(__cpp_impl_coroutine)
214 COMPILER_FEATURE_ENTRY(__cpp_impl_destroying_delete)
215 COMPILER_FEATURE_ENTRY(__cpp_impl_three_way_comparison)
216 COMPILER_FEATURE_ENTRY(__cpp_init_captures)
217 COMPILER_FEATURE_ENTRY(__cpp_modules)
218 COMPILER_FEATURE_ENTRY(__cpp_nontype_template_args)
219 COMPILER_FEATURE_ENTRY(__cpp_using_enum)
220 };
221 static CompilerFeature cxx20lib[] = {
222 COMPILER_FEATURE_ENTRY(__cpp_lib_array_constexpr)
223 COMPILER_FEATURE_ENTRY(__cpp_lib_assume_aligned)
224 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_flag_test)
225 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_float)
226 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_lock_free_type_aliases)
227 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_ref)
228 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_shared_ptr)
229 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_value_initialization)
230 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_wait)
231 COMPILER_FEATURE_ENTRY(__cpp_lib_barrier)
232 COMPILER_FEATURE_ENTRY(__cpp_lib_bind_front)
233 COMPILER_FEATURE_ENTRY(__cpp_lib_bit_cast)
234 COMPILER_FEATURE_ENTRY(__cpp_lib_bitops)
235 COMPILER_FEATURE_ENTRY(__cpp_lib_bounded_array_traits)
236 COMPILER_FEATURE_ENTRY(__cpp_lib_char8_t)
237 COMPILER_FEATURE_ENTRY(__cpp_lib_chrono)
238 COMPILER_FEATURE_ENTRY(__cpp_lib_concepts)
239 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_algorithms)
240 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_complex)
241 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_dynamic_alloc)
242 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_functional)
243 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_iterator)
244 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_memory)
245 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_numeric)
246 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_string)
247 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_string_view)
248 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_tuple)
```

```
COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_utility)
COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_vector)
COMPILER_FEATURE_ENTRY(__cpp_lib_coroutine)
COMPILER_FEATURE_ENTRY(__cpp_lib_destroying_delete)
COMPILER_FEATURE_ENTRY(__cpp_lib_endian)
COMPILER_FEATURE_ENTRY(__cpp_lib_erase_if)
COMPILER_FEATURE_ENTRY(__cpp_lib_execution)
COMPILER_FEATURE_ENTRY(__cpp_lib_format)
COMPILER_FEATURE_ENTRY(__cpp_lib_generic_unordered_lookup)
COMPILER_FEATURE_ENTRY(__cpp_lib_int_pow2)
COMPILER_FEATURE_ENTRY(__cpp_lib_integer_comparison_functions)
COMPILER_FEATURE_ENTRY(__cpp_lib_interpolate)
COMPILER_FEATURE_ENTRY(__cpp_lib_is_constant_evaluated)
COMPILER_FEATURE_ENTRY(__cpp_lib_is_layout_compatible)
COMPILER_FEATURE_ENTRY(__cpp_lib_is_nothrow_convertible)
COMPILER_FEATURE_ENTRY(__cpp_lib_is_pointer_interconvertible)
COMPILER_FEATURE_ENTRY(__cpp_lib_jthread)
COMPILER_FEATURE_ENTRY(__cpp_lib_latch)
COMPILER_FEATURE_ENTRY(__cpp_lib_list_remove_return_type)
COMPILER_FEATURE_ENTRY(__cpp_lib_math_constants)
COMPILER_FEATURE_ENTRY(__cpp_lib_polymorphic_allocator)
COMPILER_FEATURE_ENTRY(__cpp_lib_ranges)
COMPILER_FEATURE_ENTRY(__cpp_lib_remove_cvref)
COMPILER_FEATURE_ENTRY(__cpp_lib_semaphore)
COMPILER_FEATURE_ENTRY(__cpp_lib_shared_ptr_arrays)
COMPILER_FEATURE_ENTRY(__cpp_lib_shift)
COMPILER_FEATURE_ENTRY(__cpp_lib_smart_ptr_for_overwrite)
COMPILER_FEATURE_ENTRY(__cpp_lib_source_location)
COMPILER_FEATURE_ENTRY(__cpp_lib_span)
COMPILER_FEATURE_ENTRY(__cpp_lib_ssize)
COMPILER_FEATURE_ENTRY(__cpp_lib_starts_ends_with)
COMPILER_FEATURE_ENTRY(__cpp_lib_string_view)
COMPILER_FEATURE_ENTRY(__cpp_lib_syncbuf)
COMPILER_FEATURE_ENTRY(__cpp_lib_three_way_comparison)
COMPILER_FEATURE_ENTRY(__cpp_lib_to_address)
COMPILER_FEATURE_ENTRY(__cpp_lib_to_array)
COMPILER_FEATURE_ENTRY(__cpp_lib_type_identity)
COMPILER_FEATURE_ENTRY(__cpp_lib_unwrap_ref)
};

static CompilerFeature cxx23[] = {
COMPILER_FEATURE_ENTRY(__cpp_cxx23_stub) //< Populate eventually
};
static CompilerFeature cxx23lib[] = {
COMPILER_FEATURE_ENTRY(__cpp_lib_cxx23_stub) //< Populate eventually
```

```
};

static CompilerFeature attributes[] = {
COMPILER_ATTRIBUTE_ENTRY(carries_dependency)
COMPILER_ATTRIBUTE_ENTRY(deprecated)
COMPILER_ATTRIBUTE_ENTRY(fallthrough)
COMPILER_ATTRIBUTE_ENTRY(likely)
COMPILER_ATTRIBUTE_ENTRY(maybe_unused)
COMPILER_ATTRIBUTE_ENTRY(nodiscard)
COMPILER_ATTRIBUTE_ENTRY(noreturn)
COMPILER_ATTRIBUTE_ENTRY(no_unique_address)
COMPILER_ATTRIBUTE_ENTRY(unlikely)
};

constexpr bool is_feature_supported(const CompilerFeature& x) {
    return x.value[0] != '_' && x.value[0] != '0' ;
}

inline void print_compiler_feature(const CompilerFeature& x) {
    constexpr static int max_name_length = 44; //< Update if necessary
    std::string value{ is_feature_supported(x) ? x.value : "------" };
    if (value.back() == 'L') value.pop_back(); //~ 201603L -> 201603
    // value.insert(4, 1, '-'); //~ 201603 -> 2016-03
    if ( (print.supported_features && is_feature_supported(x))
        || (print.unsupported_features && !is_feature_supported(x))) {
            std::cout << std::left << std::setw(max_name_length)
                      << x.name << " " << value << '\n';
    }
}

template<size_t N>
inline void show(char const* title, CompilerFeature (&features)[N]) {
    if (print.titles) {
        std::cout << '\n' << std::left << title << '\n';
    }
    if (print.sorted_by_value) {
        std::sort(std::begin(features), std::end(features),
            [](CompilerFeature const& lhs, CompilerFeature const& rhs) {
                return std::strcmp(lhs.value, rhs.value) < 0;
            });
    }
    for (const CompilerFeature& x : features) {
        print_compiler_feature(x);
    }
}
```

```
339
340 int main() {
341     if (print.general_features) show("C++ GENERAL", cxx);
342     if (print.cxx11 && print.core_features) show("C++11 CORE", cxx11);
343     if (print.cxx14 && print.core_features) show("C++14 CORE", cxx14);
344     if (print.cxx14 && print.lib_features ) show("C++14 LIB" , cxx14lib);
345     if (print.cxx17 && print.core_features) show("C++17 CORE", cxx17);
346     if (print.cxx17 && print.lib_features ) show("C++17 LIB" , cxx17lib);
347     if (print.cxx20 && print.core_features) show("C++20 CORE", cxx20);
348     if (print.cxx20 && print.lib_features ) show("C++20 LIB" , cxx20lib);
349     if (print.cxx23 && print.core_features) show("C++23 CORE", cxx23);
350     if (print.cxx23 && print.lib_features ) show("C++23 LIB" , cxx23lib);
351     if (print.attributes) show("ATTRIBUTES", attributes);
352 }
```

Of course, the length of the source file is overwhelming. When you want to know more about each macro, visit the page for feature testing[3]. In particular, that page provides a link for each macro so that you can get more information about a feature. For example, here is the table on attributes:

attribute-token	Attribute	Value	Standard
carries_dependency	[[carries_dependency]]	200809L	(C++11)
deprecated	[[deprecated]]	201309L	(C++14)
fallthrough	[[fallthrough]]	201603L	(C++17)
likely	[[likely]]	201803L	(C++20)
maybe_unused	[[maybe_unused]]	201603L	(C++17)
no_unique_address	[[no_unique_address]]	201803L	(C++20)
nodiscard	[[nodiscard]]	201603L	(C++17)
		201907L	(C++20)
noreturn	[[noreturn]]	200809L	(C++11)
unlikely	[[unlikely]]	201803L	(C++20)

Macros for the attributes

Here is a demonstration of the `<version>` header and its macros. I executed the program on the brand-new GCC, Clang, and MSVC compilers. I used the Compiler Explorer for the GCC and Clang compilers. The `/Zc:__cplusplus` flag enables the `__cplusplus` macro reports the recent C++ language standards support. Additionally, I enabled C++20 support on all three platforms. For obvious reasons, I only display the support of the C++20 core language.

- **GCC 10.2**

[3] https://en.cppreference.com/w/cpp/feature_test

```
C++20 CORE
__cpp_aggregate_paren_init                  201902
__cpp_char8_t                               201811
__cpp_concepts                              201907
__cpp_conditional_explicit                  201806
__cpp_consteval                             ------
__cpp_constexpr                             201907
__cpp_constexpr_dynamic_alloc               201907
__cpp_constexpr_in_decltype                 201711
__cpp_constinit                             201907
__cpp_deduction_guides                      201907
__cpp_designated_initializers               201707
__cpp_generic_lambdas                       201707
__cpp_impl_coroutine                        ------
__cpp_impl_destroying_delete                201806
__cpp_impl_three_way_comparison             201907
__cpp_init_captures                         201803
__cpp_modules                               ------
__cpp_nontype_template_args                 201411
__cpp_using_enum                            ------
```

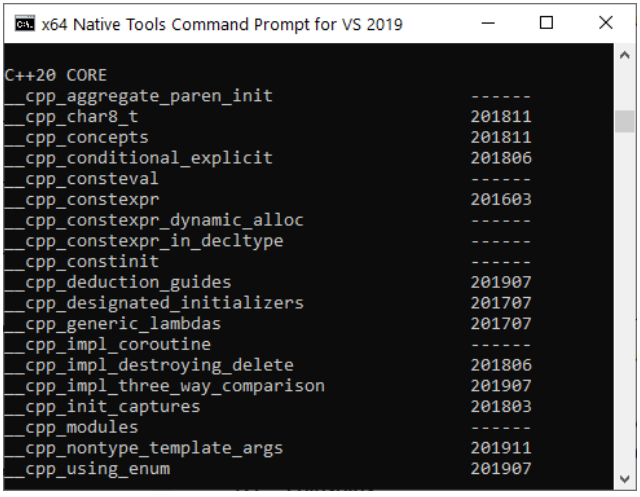
C++20 core language support available on the GCC compiler

- **Clang 11.0**

```
C++20 CORE
__cpp_aggregate_paren_init                  ------
__cpp_char8_t                               201811
__cpp_concepts                              201907
__cpp_conditional_explicit                  201806
__cpp_consteval                             ------
__cpp_constexpr                             201907
__cpp_constexpr_dynamic_alloc               201907
__cpp_constexpr_in_decltype                 201711
__cpp_constinit                             201907
__cpp_deduction_guides                      201703
__cpp_designated_initializers               201707
__cpp_generic_lambdas                       201707
__cpp_impl_coroutine                        ------
__cpp_impl_destroying_delete                201806
__cpp_impl_three_way_comparison             201907
__cpp_init_captures                         201803
__cpp_modules                               ------
__cpp_nontype_template_args                 201411
__cpp_using_enum                            ------
```

C++20 core language support available on the Clang compiler

- **MSVC 19.27**



C++20 core language support available on the MSVC compiler

The three screenshots speak a clear message about the big three: Their C++20 core language support is quite good at the end of 2020.

10. Glossary

The idea of this glossary is by no means to be exhaustive but to provide a reference for the essential terms.

10.1 Aggregate

Aggregates are arrays and class types. A class type is a class, a struct, or a union.

With C++20, the following condition must hold for a class type to be an aggregate.

- No private or protected non-static data members
- No user-declared or inherited constructors
- No virtual, private, or protected base classes
- No virtual member functions

10.2 Automatic Storage Duration

Object storage with automatic storage duration is automatically allocated at the beginning of the enclosing scope and deallocated at its end. All locals except objects with static storage duration have automatic storage duration.

10.3 Awaitable

An Awaitable is something you can await on. It is the argument of `co_await`: `co_await awaitable`. The awaitable determines if the coroutine pauses or not.

10.4 Awaiter

The `co_await` operator needs an awaitable as an argument. Typically, the awaitable becomes the awaiter. The concept awaiter requires three functions.

The Concept Awaiter

Function	Description
`await_ready`	Indicates if the result is ready. When it returns `false`, `await_suspend` is called.
`await_suspend`	Schedules the coroutine to resume or destroy.
`await_resume`	Provides the result for the `co_await exp` expression.

The C++20 standard already defines two basic awaitables: `std::suspend_always`, and `std::suspend_never`.

10.5 Callable

see Callable Unit.

10.6 Callable Unit

A callable unit, or callable for short, is something that behaves like a function. Not only are these named functions but also function objects or lambda expressions. If a callable accepts one argument, it's called a unary callable, and with two arguments, it's called a binary callable.

Predicates are special callables that return a boolean as a result.

10.7 Concurrency

Concurrency means that the execution of several tasks overlaps. Concurrency is a superset of parallelism.

10.8 Critical Section

A critical section is a section of code that contains shared variables and must be protected to avoid a data race. At most one thread at a time should enter a critical section.

10.9 Data Race

A data race is a situation in which at least two threads access a shared variable at the same time. At least one thread tries to modify the variable, and the other tries to read or modify the variable. If your program has a data race, it has undefined behavior. This means all outcomes are possible.

10.10 Deadlock

A deadlock is a state in which at least one thread is blocked forever because it waits for the release of a resource that it will never get.

There are two main reasons for deadlocks:

1. A mutex has not been unlocked.
2. You lock your mutexes in an incorrect order.

10.11 Dynamic Storage Duration

Objects with dynamic storage duration are explicitly allocated and deallocated using dynamic memory allocation functions such as new[1] or delete[2].

10.12 Eager Evaluation

In the case of eager evaluation, the expression is evaluated immediately. This evaluation strategy is the opposite to lazy evaluation. Eager evaluation is often called greedy evaluation.

10.13 Executor

An executor is an object associated with a specific execution context. It provides one or more execution functions for creating execution agents from a callable function object.

10.14 Function Objects

First of all, don't call them functors[3]. That's a *well-defined* term from a branch of mathematics called category theory[4].

Function objects are objects that behave like functions. They achieve this by implementing the function call operator. As function objects are objects, they can have attributes and, therefore, state.

[1] https://en.cppreference.com/w/cpp/memory/new/operator_new
[2] https://en.cppreference.com/w/cpp/memory/new/operator_delete
[3] https://en.wikipedia.org/wiki/Functor
[4] https://en.wikipedia.org/wiki/Category_theory

```
struct Square{
  void operator()(int& i){i= i*i;}
};

std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::for_each(myVec.begin(), myVec.end(), Square());

for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
```

Instantiate function objects to use them

It's a common error that the name of the function object (`Square`) is used in an algorithm instead of an instance of the function object (`Square()`) itself: `std::for_each(myVec.begin(), myVec.end(), Square)`. Of course, that's a typical error. You have to use the instance: `std::for_each(myVec.begin(), myVec.end(), Square())`

10.15 Lambda Expressions

Lambda expressions provide their functionality in place. The compiler gets all the necessary information to optimize the code optimally. Lambda functions can receive their arguments by value or by reference. They can capture the variables of their defining environment by value or by reference as well.

```
std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::for_each(myVec.begin(), myVec.end(), [](int& i){ i= i*i; });
                                    // 1 4 9 16 25 36 49 64 81 100
```

10.16 Lazy Evaluation

In the case of lazy evaluation[5], the expression is only evaluated if needed. This evaluation strategy is opposite to eager evaluation. Lazy evaluation is often called call-by-need.

10.17 Literal Type

A literal type is according to cppreference.com/LiteralType[6] any of the following type with C++20:

[5] https://en.wikipedia.org/wiki/Lazy_evaluation
[6] https://en.cppreference.com/w/cpp/named_req/LiteralType

- scalar type;
- reference type;
- an array of literal types;
- possibly cv-qualified class type that has all of the following properties:
  - has a `constexpr` destructor,
  - is either
    - an aggregate type,
    - a type with at least one `constexpr` (possibly template) constructor that is not a copy or move constructor,
    - a closure type (lambda)
  - for unions, at least one non-static data member is of non-volatile literal type,
  - for non-unions, all non-static data members and base classes are of non-volatile literal types.

10.18 Lock-free

A non-blocking algorithm is lock-free if there is guaranteed system-wide progress.

10.19 Lost Wakeup

A lost wakeup is a situation in which a thread misses its wake-up notification due to a race condition.

10.20 Math Laws

A binary operation (*) on some set X is

- **associative**, if it satisfies the associative law for all x, y, z in X: (x * y) * z = x * (y * z)
- **commutative**, if it satisfies the commutative law for all x, y in X: x * y = y * x
- **distributive**, if it satisfies the distributive law for all x, y, z in X: x(y + z) = xy + xz

10.21 Memory Location

A memory location is according to cppreference.com[7]

- an object of scalar type (arithmetic type, pointer type, enumeration type, or `std::nullptr_t`),
- or the largest contiguous sequence of bit fields of non-zero length.

[7] http://en.cppreference.com/w/cpp/language/memory_model

10.22 Memory Model

The memory model defines the relationship between objects and memory locations and deals with the question: What happens if two threads access the same memory locations?

10.23 Non-blocking

An algorithm is called non-blocking if a failure or suspension of any thread cannot cause failure or suspension of another thread. This definition is from the excellent book Java concurrency in practice[8].

10.24 Object

A type is an object if it is either a scalar, an array, a union, or a class.

10.25 Parallelism

Parallelism means that several tasks are performed at the same time. Parallelism is a subset of Concurrency. In contrast to concurrency, parallelism requires multiple cores.

10.26 POD (Plain Old Data)

A POD type is trivial and has standard layout.

10.27 Predicate

Predicates are callable units that return something convertible to a boolean as a result. If a predicate has one argument, it's called a unary predicate. If a predicate has two arguments, it's called a binary predicate.

10.28 RAII

Resource **A**cquisition **I**s **I**nitialization, or RAII for short, stands for a popular technique in C++ in which the resource acquisition and release are bound to the lifetime of an object. This means for a lock that the mutex will be locked in the constructor and unlocked in the destructor.

Typical use cases in C++ are locks that handle the lifetime of its underlying mutex, smart pointers that handle the lifetime of its resource (memory), or containers of the standard template library[9] that handle the lifetime of their elements.

[8] http://jcip.net/
[9] https://en.cppreference.com/w/cpp/container

10.29 Race Conditions

A race condition is a situation in which the result of an operation depends on the interleaving (ordering of operations) of certain individual operations.

Race conditions are quite difficult to see. Whether they occur depends on the interleaving of the threads. That means the number of your cores, your system's utilization, or your executable's optimization level may all be reasons why a race condition appears or does not.

10.30 Regular Type

In addition to the requirements of the concept `SemiRegular`, the concept `Regular` requires that the type is equally comparable.

10.31 Scalar Type

A scalar type is either an arithmetic type (see `std::is_arithmetic`[10]), an `enum`, a pointer, a member pointer, or a `std::nullptr_t`.

10.32 SemiRegular

A `semiregular` type `X` has to support the Big Six and has to be swappable: `swap(X&, X&)`

10.33 Short-Circuit Evaluation

Short circuit evaluation means that the evaluation of a logical expression automatically stops when its overall result is already determined.

10.34 Standard-Layout Type

A standard-layout type does not use features that are not available in C. All its members must have the same access specifier. User-defined special members are allowed. The following characteristic holds for standard layout types.

A standard-layout type can only have

- non-virtual functions or non-virtual base classes

[10] https://en.cppreference.com/w/cpp/types/is_arithmetic

- non-static data members with the same access specifiers
- non-static members or bases classes that are standard layout
- Meets one of these conditions:
  - no non-static data member in the most-derived class and no more than one base class with non-static data members, or
  - has no base classes with non-static data members

A standard-layout Type is in contrast to a trivial type C compatible.

10.35 Static Storage Duration

Global (namespace) variables, static variables, or static class members have static storage duration. These objects are allocated when the program starts and are deallocated when the program ends.

10.36 Spurious Wakeup

A spurious wakeup is an erroneous notification. The waiting component of a condition variable or atomic flag can receive a notification, even though the notification component did not send the signal.

10.37 The Big Four

The Big Four are the four key features of C++20: concepts, modules, the ranges library, and coroutines.

- **Concepts** change the way we think about and program with templates. They are semantic categories for template parameters. They enable you to express your intention directly in the type system. If something goes wrong, the compiler gives you a clear error message.
- **Modules** overcome the restrictions of header files. They promise a lot. For example, the separation of header and source files becomes as obsolete as the preprocessor. In the end, we have faster build times and an easier way to build packages.
- The new **ranges library** supports performing algorithms directly on the containers, composing algorithms with the pipe symbol, and applying algorithms lazily on infinite data streams.
- Thanks to **coroutines**, asynchronous programming in C++ becomes mainstream. Coroutines are the basis for cooperative tasks, event loops, infinite data streams, or pipelines.

10.38 The Big Six

The Big Six consists of the following functions:

- Default constructor: `X()`
- Copy constructor: `X(const X&)`
- Copy assignment: `X& operator = (const X&)`
- Move constructor: `X(X&&)`
- Move assignment: `X& operator = (X&&)`
- Destructor: `~X()`

10.39 Thread

In computer science, a thread of execution is the smallest sequence of programmed instructions that a scheduler can manage independently typically. It is typically part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases, a thread is a process component. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. For the details, read the Wikipedia article about threads[11].

10.40 Thread Storage Duration

`thread_local` variables have thread storage duration. Thread-local data is created for each thread that uses this data. `thread_local` data exclusively belongs to the thread. They are created at their first usage and its lifetime is bound to the lifetime of the thread it belongs to. Often thread-local data is called thread-local storage.

10.41 Time Complexity

O(i) stands for the time complexity (run time) of an operation. With *O(1)*, the run time of an operation on a container is constant and is, hence, independent of its size. Conversely, *O(n)* means that the run time depends linearly on the number of container elements.

10.42 Translation Unit

A translation unit is the source file after processing of the C preprocessor. The C preprocessor includes the header files using `#include directives`, and performs conditional inclusion with directives such as `#ifdef`, or `#ifndef`, and expands macros. The compiler uses the translation unit to create an object file.

[11] https://en.wikipedia.org/wiki/Thread_(computing)

10.43 Trivial Type

A trivial type is a type for which the compiler creates all the special member functions implicitly or explicitly they are defaulted by the user. The member of a trivial type can have different access specifiers and occupies a contiguous memory block.

A trivial type cannot have

- virtual functions or virtual base classes.
- non-trivial base classes.
- non-trivial members.

A trivial type is in contrast to a standard-layout[12] type not compatible with C.

10.44 Type Erasure

Type erasure is a type-safe generic way to provide a unique interface for different types. The different types don't need a common base class and are unrelated.

10.45 Undefined Behavior

All bets are off. Your program can produce the correct or the wrong result, can crash at run time, or may not even compile. That behavior might change when porting to a new platform, upgrading to a new compiler, or as a result of an unrelated code change.

[12] https://en.cppreference.com/w/cpp/named_req/StandardLayoutType

Index

Entries in capital letters stand for sections and subsections.

** **
(operator)
#
(formatting)
‘
‘jthread (jthread)
-
-fmodule-header
-fmodule-mapper
-fmodules-ts
/
/exportHeader
/headerName
/headerUnit
/ifcOnly
/ifcOutput
/ifcSearchDir
/interface
/internalPartition
/reference
/std:c++latest
/TP
/translateInclude
/validateIfcChecksum[-]
0
0 (formatting)
[
[[carries_dependency]]
[[deprecated]]
[[fallthrough]]
[[likely]]
[[maybe_unused]]
[[nodiscard]]
[[noreturn]]
[[unlikely]]
[i] (mdspan in C++23)
[i] (span)
[i] (subrange)
[i] (view_interface)
_
__cplusplus
__cpp_aggregate_bases
__cpp_aggregate_nsdmi
__cpp_aggregate_paren_init
__cpp_alias_templates
__cpp_aligned_new
__cpp_attributes
__cpp_binary_literals
__cpp_capture_star_this
__cpp_char8_t
__cpp_concepts
__cpp_conditional_explicit
__cpp_consteval
__cpp_constexpr
__cpp_constinit
__cpp_decltype
__cpp_decltype_auto
__cpp_deduction_guides
__cpp_delegating_constructors
__cpp_designated_initializers
__cpp_enumerator_attributes
__cpp_exceptions
__cpp_fold_expressions
__cpp_generic_lambdas
__cpp_generic_lambdas
__cpp_guaranteed_copy_elision
__cpp_hex_float
__cpp_if_constexpr

__cpp_impl_coroutine
__cpp_impl_destroying_delete
__cpp_impl_three_way_comparison
__cpp_inheriting_constructors
__cpp_inheriting_constructors
__cpp_init_captures
__cpp_init_captures
__cpp_initializer_lists
__cpp_inline_variables
__cpp_lambdas
__cpp_lib_addressof_constexpr
__cpp_lib_allocator_traits_is_always_equal
__cpp_lib_any
__cpp_lib_apply
__cpp_lib_array_constexpr
__cpp_lib_as_const
__cpp_lib_assume_aligned
__cpp_lib_atomic_flag_test
__cpp_lib_atomic_float
__cpp_lib_atomic_is_always_lock_free
__cpp_lib_atomic_lock_free_type_aliases
__cpp_lib_atomic_ref
__cpp_lib_atomic_shared_ptr
__cpp_lib_atomic_value_initialization
__cpp_lib_atomic_wait
__cpp_lib_barrier
__cpp_lib_bind_front
__cpp_lib_bit_cast
__cpp_lib_bitops
__cpp_lib_bool_constant
__cpp_lib_bounded_array_traits
__cpp_lib_boyer_moore_searcher
__cpp_lib_byte
__cpp_lib_char8_t
__cpp_lib_chrono
__cpp_lib_chrono
__cpp_lib_chrono_udls
__cpp_lib_clamp
__cpp_lib_complex_udls
__cpp_lib_concepts
__cpp_lib_constexpr_algorithms
__cpp_lib_constexpr_complex
__cpp_lib_constexpr_dynamic_alloc
__cpp_lib_constexpr_functional
__cpp_lib_constexpr_iterator
__cpp_lib_constexpr_memory
__cpp_lib_constexpr_numeric
__cpp_lib_constexpr_string
__cpp_lib_constexpr_string_view
__cpp_lib_constexpr_tuple
__cpp_lib_constexpr_utility
__cpp_lib_constexpr_vector
__cpp_lib_coroutine
__cpp_lib_destroying_delete
__cpp_lib_enable_shared_from_this
__cpp_lib_endian
__cpp_lib_erase_if
__cpp_lib_exchange_function
__cpp_lib_execution
__cpp_lib_filesystem
__cpp_lib_format
__cpp_lib_gcd_lcm
__cpp_lib_generic_associative_lookup
__cpp_lib_generic_unordered_lookup
__cpp_lib_hardware_interference_size
__cpp_lib_has_unique_object_representations
__cpp_lib_hypot
__cpp_lib_incomplete_container_elements
__cpp_lib_int_pow2
__cpp_lib_integer_comparison_functions
__cpp_lib_integer_sequence
__cpp_lib_integral_constant_callable
__cpp_lib_interpolate
__cpp_lib_invoke
__cpp_lib_is_aggregate
__cpp_lib_is_constant_evaluated
__cpp_lib_is_final
__cpp_lib_is_invocable
__cpp_lib_is_layout_compatible
__cpp_lib_is_nothrow_convertible
__cpp_lib_is_null_pointer
__cpp_lib_is_pointer_interconvertible
__cpp_lib_is_swappable
__cpp_lib_jthread

A

__cpp_lib_latch
__cpp_lib_launder
__cpp_lib_list_remove_return_type
__cpp_lib_logical_traits
__cpp_lib_make_from_tuple
__cpp_lib_make_reverse_iterator
__cpp_lib_make_unique
__cpp_lib_map_try_emplace
__cpp_lib_math_constants
__cpp_lib_math_special_functions
__cpp_lib_memory_resource
__cpp_lib_node_extract
__cpp_lib_nonmember_container_access
__cpp_lib_not_fn
__cpp_lib_null_iterators
__cpp_lib_optional
__cpp_lib_parallel_algorithm
__cpp_lib_polymorphic_allocator
__cpp_lib_quoted_string_io
__cpp_lib_ranges
__cpp_lib_raw_memory_algorithms
__cpp_lib_remove_cvref
__cpp_lib_result_of_sfinae
__cpp_lib_robust_nonmodifying_seq_ops
__cpp_lib_sample
__cpp_lib_scoped_lock
__cpp_lib_semaphore
__cpp_lib_shared_mutex
__cpp_lib_shared_ptr_arrays
__cpp_lib_shared_ptr_weak_type
__cpp_lib_shared_timed_mutex
__cpp_lib_shift
__cpp_lib_smart_ptr_for_overwrite
__cpp_lib_source_location
__cpp_lib_span
__cpp_lib_ssize
__cpp_lib_starts_ends_with
__cpp_lib_string_udls
__cpp_lib_string_view
__cpp_lib_syncbuf
__cpp_lib_three_way_comparison
__cpp_lib_to_address
__cpp_lib_to_array
__cpp_lib_to_chars
__cpp_lib_transformation_trait_aliases
__cpp_lib_transparent_operators
__cpp_lib_transparent_operators
__cpp_lib_tuple_element_t
__cpp_lib_tuples_by_type
__cpp_lib_type_identity
__cpp_lib_type_trait_variable_templates
__cpp_lib_uncaught_exceptions
__cpp_lib_unordered_map_try_emplace
__cpp_lib_unwrap_ref
__cpp_lib_variant
__cpp_lib_void_t
__cpp_modules
__cpp_namespace_attributes
__cpp_noexcept_function_type
__cpp_nontype_template_args
__cpp_nontype_template_parameter_auto
__cpp_nsdmi
__cpp_range_based_for
__cpp_raw_strings
__cpp_ref_qualifiers
__cpp_return_type_deduction
__cpp_rtti
__cpp_rvalue_references
__cpp_sized_deallocation
__cpp_static_assert
__cpp_structured_bindings
__cpp_template_template_args
__cpp_threadsafe_static_init
__cpp_unicode_characters
__cpp_unicode_literals
__cpp_user_defined_literals
__cpp_using_enum
__cpp_variable_templates
__cpp_variadic_templates
__cpp_variadic_using
_dynamic_alloc
_in_decltype
A
A Flavor of Python

B

A General Mechanism to Send Signals
A Generator Function
A Quick Overview
A thread-safe singly linked list
Abbreviated Function Templates
acquire
Addable
address
adjacent_transform_view
adjacent_view
advance (subrange)
Aggregate (Glossary)
Aggregate Initialization
alignment
all (views)
All Atomic Operations (std::atomic_ref)
all_t (views)
An Infinite Data Stream
and_then (exptected)
Anonymous Concepts
April
Argument ID
Arithmetic
arrive
arrive_and_drop
arrive_and_wait (barrier)
arrive_and_wait (latch)
as_bytes (span)
as_const_view
as_rvalue_view
as_writable_bytes (span)
assertion (contracts)
assignable_from (concepts)
associative (Glossary)
atomic Extensions
Atomic Smart Pointer
atomic<shared_ptr<T>>
atomic<weak_ptr<T>>
atomic_flag Extensions
ATOMIC_FLAG_INIT
atomic_ref
atomic_shared_ptr
atomic_weak_ptr
Atomics
August
auto[beg, end] (subrange)
Automatic Storage Duration (Glossary)
Automatically Joining
await_ready
await_resume
await_suspend
await_transform
Awaitable (Glossary)
Awaitables (coroutines)
Awaitables and Awaiters (coroutines)
Awaiter (coroutines)
Awaiter (Glossary)
Awaiter
B
back (span)
back (subrange)
back (view_interface)
bad_expected_access (expected)
barrier
basic_istream (views)
basic_istream_view
basic_osyncstream
basic_streambuf
basic_syncbuf
Becoming a Coroutine
begin (format_parse_context)
begin (ranges)
begin (subrange)
bidirectional_iterator (concepts)
bidirectional_range (concepts)
big (endian)
big-endian
binary_semaphore
bind_front
bit field
Bit Manipulation
bit_cast
bit_ceil
bit_floor

C

bit_width
borrowed_range (concepts)
C
C++03
C++11
C++14
C++17
C++23 and Beyond
C++23
C++98
Calendar and Timezone
Calendar Dates
calendar
Callable (Glossary)
callable (Glossary)
Callable Unit
Case Studies
cbegin (ranges)
cdata (ranges)
cend (ranges)
char16_t
char32_t
char8_t
char
chunk_by_view
chunk_view
Cippi
Class Template Argument Deduction Guide
clear (atomic_flag)
clock
clock_cast
cmp_equal
cmp_greater
cmp_greater_equal
cmp_less
cmp_less_equal
cmp_not_equal
co_await operator
co_await
co_return
co_yield
column
common (views)
common_iterator (iterator)
common_range (concepts)
common_reference_with (concepts)
common_view
common_with (concepts)
commutative (Glossary)
comparable
Comparison
compilation (source code)
compile-time predicate
Compiler Support (modules)
Compound Requirements
Concepts
Concurrency (Glossary)
Concurrency
condition_variable_any
Conditionally Explicit Constructor
Consistent Container Erasure
consteval lambda
consteval
constexpr Container
constexpr if (concepts)
constinit
constrained placeholders
constrained template parameter
constraint-expression
constructible_from (concepts)
Container Adapters (C++23)
Container and Algorithm Improvements
contains
contiguous_iterator (concepts)
contiguous_range (concepts)
contract_violation (contracts)
Contracts (Beyond C++23)
convertible_to (concepts)
copy_constructible (concepts)
copyable (concepts)
Core Language
coroutine factory
Coroutine Frame (coroutines)
Coroutine Handle (coroutines)

DE

coroutine handle
coroutine object
coroutine state
coroutine_traits
Coroutines
count (span)
count_down
counted (views)
counted_iterator (iterator)
counting semaphores
countl_one
countl_zero
countr_one
countr_zero
cppm (file extension)
crbegin (ranges)
crend (ranges)
Critical Section (Glossary)
current
current_zone
Cute Syntax
CWG
D
d (built-in literal)
data (ranges)
data (span)
data (subrange)
data (view_interface)
Data Race (Glossary)
data_handle (mdspan in C++23)
day
Deadlock (Glossary)
December
Deducing This (C++23)
Default Member Initializers Bit Fields
default_constructible (concepts)
default_initializable (concepts)
default_sentinel (sentinel)
define (macro)
Define Concepts
derived_from (concepts)
Design Goals (coroutines)
Designated Initialization
designators
destroy
destructible (concepts)
detach
Details (coroutines)
distributive (Glossary)
done
drop (views)
drop_view
drop_while (views)
drop_while_view
dynamic extent (mdspan in C++23)
dynamic extent (span)
Dynamic Storage Duration (Glossary)
E
e
Eager evaluation (Glossary)
Edsger W. Dijkstra
egamma
elements (views)
elements_view
elif (macro)
else (macro)
emit
emplace (expected)
empty (ranges)
empty (span)
empty (subrange)
empty (view_interface)
empty (views)
empty_view
end (format_parse_contextformat_parse_context)
end (ranges)
end (subrange)
endian
endif (macro)
ends_with
Epilogue
epoch
Equal
equal_to (ranges)

FG

Equality Comparison and Three-Way Comparison
equality operator
equality
equality_comparable (concepts)
equivalence
erase-remove idiom
erase
erase_if
EWG
exchange (atomic_ref)
Executor (Glossary)
expected (C++23)
export group
export import
export namespace
export specifier
export
exportHeader (compiler option)
extension (mdspan in C++23)
extents (mdspan in C++23)
external linkage
F
Fast Synchronization of Threads
Feature Testing
February
fetch_add (atomic_ref)
fetch_and (atomic_ref)
fetch_or (atomic_ref)
fetch_sub (atomic_ref)
fetch_xor (atomic_ref)
file_clock
file_name
file_time[duration]
fill character
filter (Python)
filter (views)
filter_view
final_suspend
first (span)
flat_map (C++23)
flat_multimap (C++23)
flat_multiset (C++23)
flat_set (C++23)
floating_point (concept definition)
flush_emit
format (user-defined type)
Format String
format
format_error
format_parse_context (user-defined type)
format_to (user-defined type)
format_to
format_to_n
Formatted Input (chrono)
Formatted Output (chrono)
formatted_size
formatter (user-defined type)
Formatting Library
forward_iterator (concepts)
forward_range (concepts)
Four Ways to use a Concept
fractional_width
Friday
From Mathematics to Generic Programming
from_address
from_promise
from_stream (chrono)
front (span)
front (subrange)
front (view_interface)
Function Objects (Glossary)
function_name
Further Improvements
Further Information
G
generator (ranges in C++23)
generic lambdas
get_id
get_return_object
get_return_object_on_allocation_failure
get_stop_source
get_stop_token
get_token (stop_source)
get_tzdb

HIJKL

get_tzdb_list
get_wrapped
global module fragment
Glossary
gps_clock
gps_seconds
gps_time[duration]
greater (ranges)
greater_equal (ranges)
Guideline for a Module Structure
H
h (built-in literal)
has_single_bit
has_value (expected)
Haskell type classes
header units
headerName (compiler option)
headerUnit (compiler option)
hh_mm_ss
high_resolution_clock
Historical Context of C++
hours
I
identity (algorithm)
if (macro)
ifc (file extension)
IFC file
ifcOnly (compiler option)
ifcOutput (compiler option)
ifcSearchDir (compiler option)
ifdef (macro)
immediate function
import
in_range
include (macro)
indef (macro)
Initalizers
initial_suspend
input_iterator (concepts)
input_output_iterator (concepts)
input_range (concepts)
inspect
integral (concept definition)
integral (concepts)
Integral
interface (compiler option)
interface partition
internal linkage
internal partition
internalPartition (compiler option)
internationalization
inv_pi
inv_sqrt3
inv_sqrtpi
invariant (contracts)
invocable (concepts)
iota (views)
iota_view
is_always_lock_free (atomic_ref)
is_am
is_constant_evaluated
is_lock_free (atomic_ref)
is_negative
is_pm
Iterator
ixx (file extension)
J
January
join (views)
join
join_view
join_with_view
joinable
Joining Threads
jthread
July
June
K
keys (views)
keys_view
L
Lambda Functions (Glossary)
Lambda Improvements
language linkage

M

last (span)
last
last
latch
Latches and Barriers
layout_left (mdspan in C++23)
layout_right (mdspan in C++23)
Lazy Evaluation (Glossary)
lazy_split (views)
lazy_split_view
leap_second
LegacyRandomAccessIterator
lerp
less (ranges)
less_equal (ranges)
LEWG
lexicographical comparison
line
linking
list comprehension (Python)
Literal Type (Glossary)
little (endian)
little-endian
ln10
ln2
load (atomic_ref)
local_days (Time Points)
local_days
local_info
local_seconds
local_t
local_time[duration]
locale::global
locale
localization
locate_zone
lock-free (Glossary)
log10e
log2e
Lost Wakeup (Glossary)
lsys_days
LWG

M

make12
make14
make_format_args
make_shared
map (Python)
March
Math Laws (Glossary)
Mathematical Constants
max (barrier)
max (counting_semaphore)
max (latch)
May
Memory Location (Glossary)
Memory Model (Glossary)
mergeable (concepts)
midpoint
min (built-in literal)
minutes
Modication and Generalization of a Generator
Modularized Standard Library (C++23)
module declaration file
module declaration
module implementation unit
module interface partition
module interface unit
module linkage
module partitions
module purview
module unit
module-header (compiler option)
module-mapper (compiler option)
module
modules-ts (compiler option)
Modules
Monday
month
month_day
month_day_last
month_weekday
month_weekday_last
movable (concepts)

NOPR

move_constructiblee (concepts)
move_sentinel (sentinel)
ms (built-in literal)
Multidimensional Access (C++23)
N
named module
NaN
Nested Requirements
New Attributes
next (subrange)
no-module-lazy (compiler option)
no_unique_address (attribute)
Non-blocking (Glossary)
Non-Type Template Parameters
nonexistent_local_time
noop_coroutine
noop_coroutine_handle
nostopstate_t
Not a Number
not_equal_to (ranges)
notify_all (atomic_flag)
notify_all (atomic_ref)
notify_one (atomic_flag)
notify_one (atomic_ref)
November
ns (built-in literal)
NTTP
O
Object (Glossary)
October
ODR
ok
one definition rule
One Time Synchronization of Threads
operator bool (coroutines)
operator coroutine_handle<> (coroutines)
operator delete (coroutines)
operator new (coroutines)
operator T (atomic_ref)
Optimized == and != Operators
or_else (exptected)
ordinal dates
osyncstream
output_iterator (concepts)
output_range (concepts)
owning_view
P
Pack Expansion in Init-Capture
Parallelism (Glossary)
parse (chrono)
parse (user-defined type)
partial ordering
partial_ordering
partition interface file
Pattern Matching (Beyond C++23)
PCH
permutable (concepts)
phi
pi
placeholders
Plain Old Data (Glossary)
POD
popcount
postcondition (contracts)
precision
precompiled header
precondition (contracts)
Predefined Concepts
predicate (concepts)
Predicate (Glossary)
preprocessing
prev (subrange)
primary interface file
primary module interface unit
primary module interface
print (C++23)
println (C++23)
private module fragment
projection
promise object (coroutine)
Promise Object (coroutines)
promise
Pull Pipelines
R

S

Race Condition (Glossary)
RAII (Glossary)
random_access_iterator (concepts)
random_access_range (concepts)
range (concepts)
Range Adaptor
Range Adaptor
Range-based for-loop
Ranges Extensions(C++23)
Ranges Library
rank (mdspan in C++23)
rbegin (ranges)
reachability
ref_view
reference (compiler option)
Reference PCs
Reflection (Beyond C++23)
reflection operator
regular (concepts)
Regular Type (Glossary)
regular_invocable (concepts)
relational operator
release (counting_semaphore)
reload_tzdb
remote_version
rend (ranges)
request_stop (jthread)
request_stop (stop_source)
Requires Clauses
Requires Expressions
requires requires
Restrictions (coroutines)
resumable function
resumable object
resume
return_value
return_void
reverse (views)
reverse_view
rotl
rotr
S
s (built-in literal)
Safe Comparison of Integers integral
same_as (concepts)
Saturday
Scalar Type (Glossary)
scalar type
seconds
Semaphores
semiregular (concepts)
SemiRegular (Glossary)
September
SG10
SG11
SG12
SG13
SG14
SG15
SG16
SG17
SG18
SG19
SG1
SG20
SG21
SG22
SG2
SG2
SG3
SG4
SG5
SG6
SG7
SG8
SG9
SG
shift_left
shift_rigth
Short-Circuit Evaluation Type (Glossary)
sign
signed_integral (concept definition)
SignedIntegral
Simple Requirements

T

single (views)
single_view
size (mdspan in C++23)
size (ranges)
size (span)
size (subrange)
size (view_interface)
size
size_bytes (span)
sized_range (concepts)
slide_view
sortable (concepts)
sorted_unique (C++23)
source_location
spacehip operator (concepts)
spaceship
span
Specilisations of std::atomic_ref
split (views)
split_view
Spurious wakeup (Glossary)
sqrt2
sqrt3
ssize (ranges)
ssize
Standard Library
Standard-Layout Type (Glossary)
Standardization
starts_with
stateless lambda
static extent (mdspan in C++23)
static extent (span)
static initialization order fiasco
Static Storage Duration (Glossary)
static_assert (concepts)
std:c++latest (compiler option)
steady_clock
stop_callback
stop_possible (stop_source)
stop_possible (stop_token)
stop_requested (stop_source)
stop_requested (stop_token)
stop_source
stop_token
store (atomic_ref)
stride_view
strong ordering
strong_ordering
Study Group
submodules
subseconds
subspan (span)
Sunday
suspend_always
suspend_never
swappable (concepts)
symmetric transfer
Synchronized Output Streams
sys_days
sys_info
sys_seconds
sys_time[duration]
system_clock
T
tai_clock
tai_seconds
tai_time[duration]
take (views)
take_view
take_while (views)
take_while_view
tdzb_list
Template Improvements
Template Introduction
template lambdas
Templates in Modules
test (atomic_flag)
Test of Concepts
test_and_set (atomic_flag)
The Awaiter Workflow
The Big Four (Glossary)
The Big Six (Glossary)
The Concepts Equal andOrdering
The Concepts SemiRegular and Regular

UV

The Details
The Framework (coroutines)
The Promise Workflow
The structure of a std::list
The Workflow
this_thread::get_id
this_thread::sleep_for
this_thread::sleep_until
this_thread::yield
Thread (Glossary)
Thread Storage Duration (Glossary)
thread::hardware_concurrency
three-way comparison operator
Three-Way Comparison operator
Thuesday
Thursday
Time Complexity (Glossary)
time duration
time of day
time point
time zone
time_zone
time_zone_link
to (ranges)
to_address
to_array
to_duration
total ordering
totally_ordered (concepts)
TP (compiler option)
TR1
trailing requires clause
transform (exptected)
transform (views)
transform_error (exptected)
transform_view
Transient Allocation
translateInclude (compiler option)
Translation Unit (Glossary)
Trivial Type (Glossary)
try_acquire
try_acquire_for
try_acquire_until
try_wait
Type Erasure (Glossary)
Type Requirements
Typical Use-Cases (coroutines)
tzdb
U
unconstrained placeholders
Undefined Behavior (Glossary)
Underlying Concepts (coroutines)
unevaluated context
unexpected
Unformatted Output (chrono)
unhandled_exception
Unix time
unreachable_sentinel (sentinel)
unseq (execution)
unsigned_integral (concept definition)
UnsignedIntegral
us (built-in literal)
using enum in local Scopes
UTC time
utc_clock
utc_seconds
utc_time[duration]
V
validateIfcChecksum[-] (compiler option)
value (expected)
value_or (expected)
values (views)
values_view
Variations of
Various Job Workflows
vformat
vformat_to
view (concepts)
view
view_interface
viewable_range (concepts)
Views on Temporary Ranges
views::adjacent
views::adjacent_transform

WYZ

views::all_t
views::as_const
views::as_rvalue
views::chunk
views::chunk_by
views::join_with
views::slide
views::stride
views::transform
views::zip
Virtual constexpr function
visibility
volatile
W
wait (atomic_flag)
wait (atomic_ref)
wait (barrier)
wait (condition_variable_any)
wait (latch)
wait_for (condition_variable_any)
wait_until (condition_variable_any)
weak ordering
weak_ordering
Wednesday
weekday
weekday_indexed
weekday_last
WG21
width
with
Working Group 21
wosyncstream
Y
y (built-in literal)
year
year_month
year_month_day
year_month_day_last
year_month_weekday
year_month_weekday_last
yield_value
Z
Z-fno-module-lazy
zip_transform_view
zip_view
zoned time
zoned_time
zoned_traits