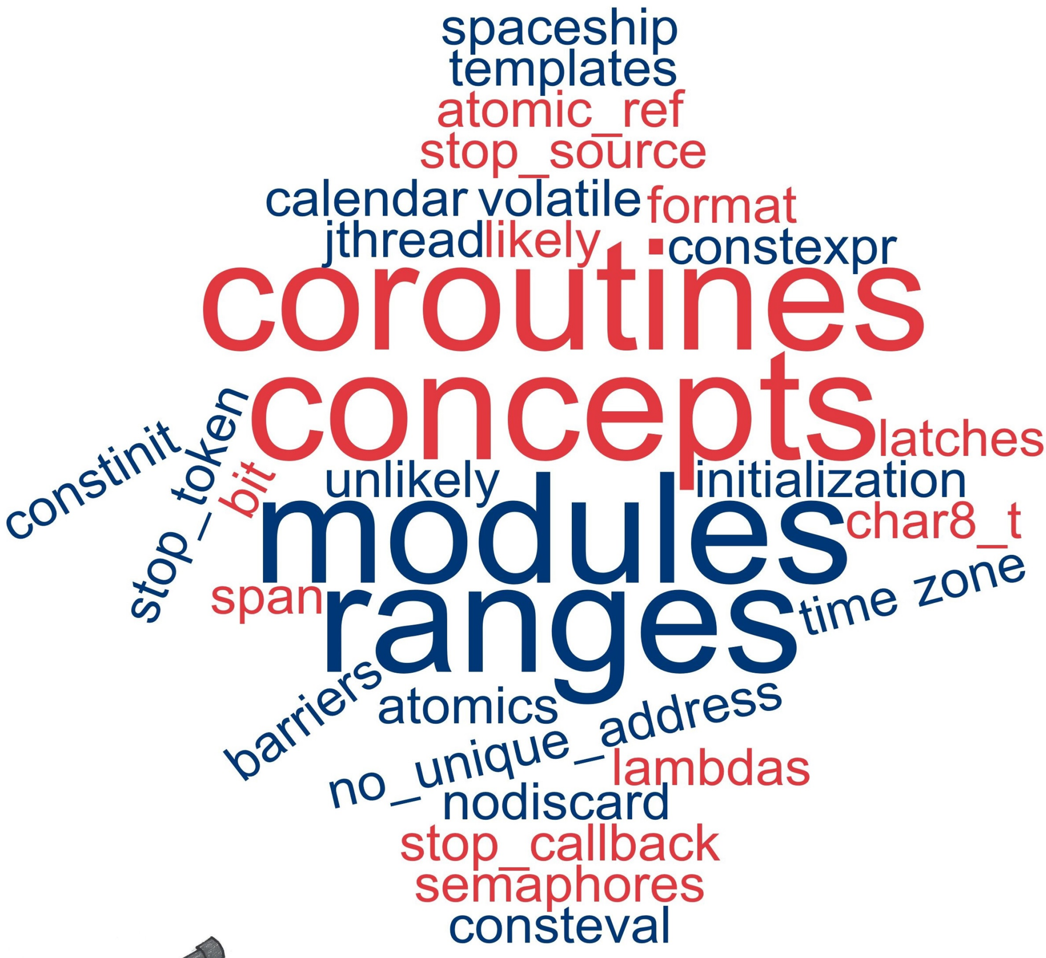
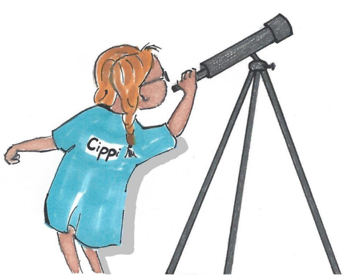
C++20

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 2021-06-06

Leanpub

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

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 background tips, warnings, and distilled information.

Tip Headline

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

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 can pollute namespaces. Consequently, I use them only when it improves the code's readability (e.g.: `using namespaces std::chrono_literals`). When necessary for layout reasons, I apply a `using`-declaration, such as `using std::chrono::system_clock`.

To summarize, I only use the following layout rules if necessary:

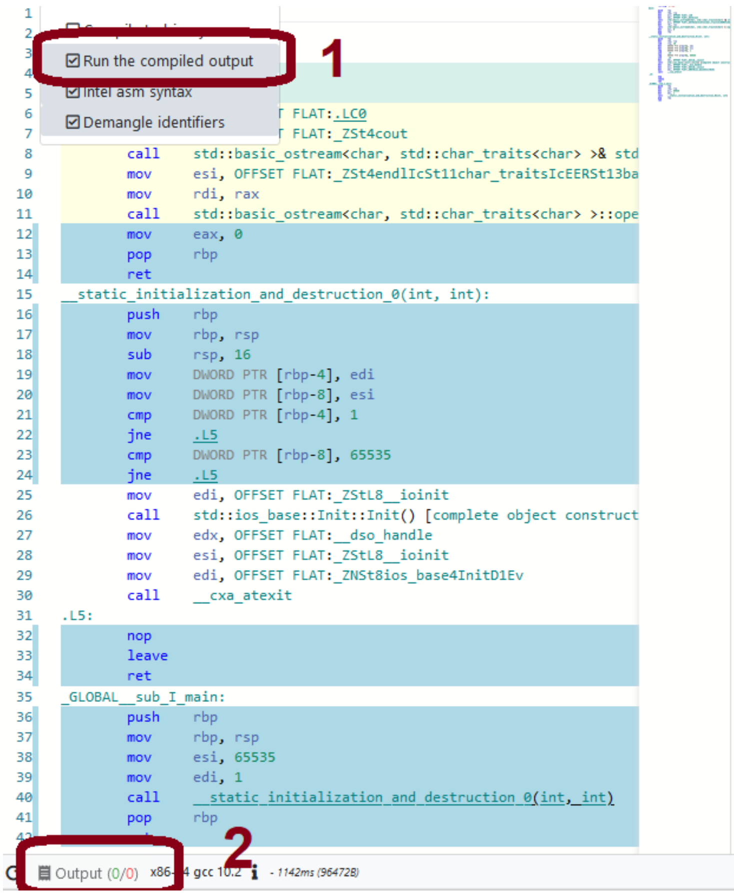
- I indent two characters instead of four.
- I apply the `using namespace std` directive.

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 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]. When you use Compiler Explorer with GCC or Clang, you can also execute the program. First, you have to enable `Run the compiled output` (1) and, second, open the `Output` window (2).

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



Run code in the Compiler Explorer

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

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

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

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, 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++.

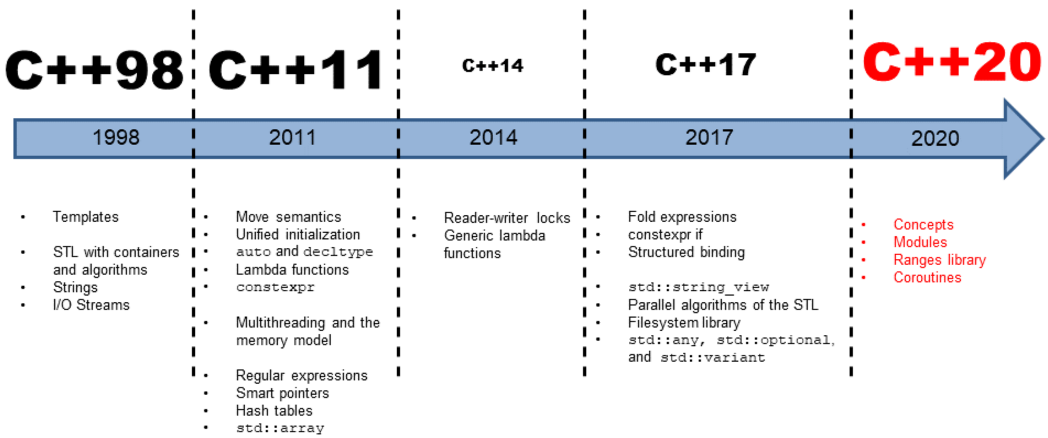
[8]https://www.modernescpp.com/
[9]https://www.grimm-jaud.de/index.php/blog

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 80's, 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

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

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 there is no place on the timeline above. In the community, C++03, which includes C++98, is called **legacy C++**.

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], which was founded by members of the C++ standardization committee. TR1 had 13 libraries that were 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.

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

1.6 C++17

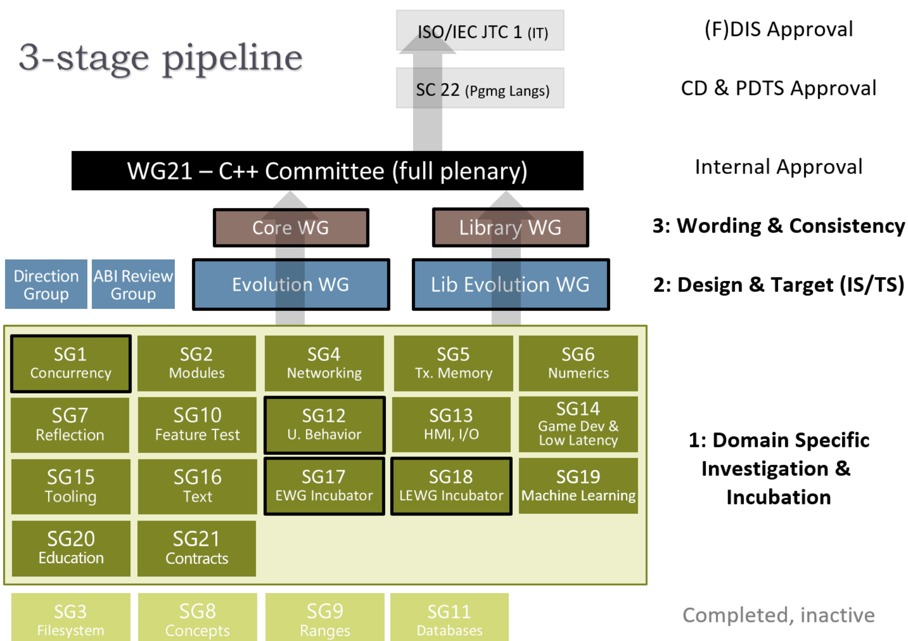
C++17 is neither a big nor a small C++ standard. It has two outstanding features: the parallel STL and the standardized filesystem API. About 80 algorithms of the Standard Template Library can be executed in parallel or vectorized. As with C++11, the boost libraries were highly influential for C++17.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

2. Standardization

The C++ standardization process is democratic. The committee is called WG21 (Working Group 21) and 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 have 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 a concise overview of the standardization in C++ and, in particular, the C++ committee. You can find more details about the standardization on https://isocpp.org/std[1].

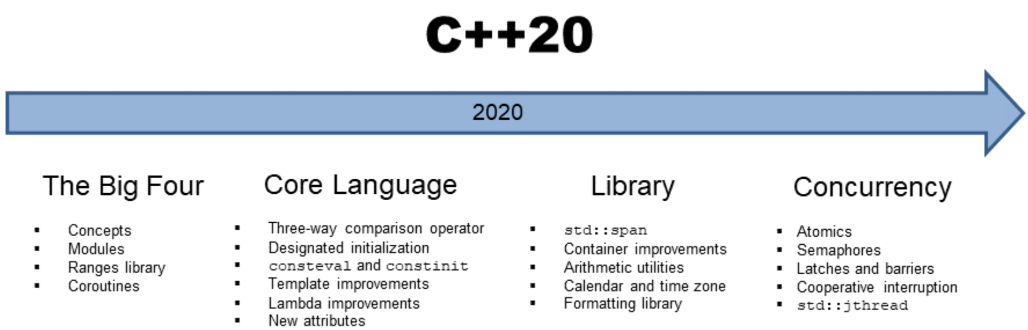
[1] https://isocpp.org/std

A Quick Overview of C++20

3. C++20

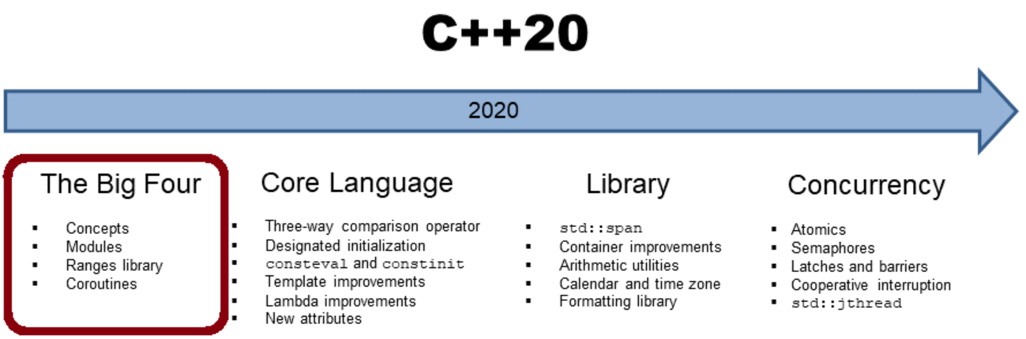
Before I dive into the details of C++20, I want to give a quick overview of the features in C++20. This overview should serve two purposes; to give a first impression, and to provide links to the relevant sections you can use to dive directly into the details. Consequently, this chapter has only code 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 that you 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 the way 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` be `true`. `std::is_integral<T>::value` is a value 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 kind of 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`.

Here is the semantically equivalent `gcd` algorithm, using a `requires` clause.

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

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:

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

```
int main() {
    std::vector<int> ints{0, 1, 2, 3, 4, 5};
    auto even = [](int i){ return i % 2 == 0; };
    auto square = [](int i) { return i * i; };

    for (int i : ints | std::views::filter(even) |
                        std::views::transform(square)) {
        std::cout << i << ' ';              // 0 4 16
    }
}
```

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, instead C++20 provides a framework for implementing coroutines. This framework consists of more than 20 functions, 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 chapter coroutines provides the implemenation of the `Generator`.

A generator for an infinite data-stream

```
Generator<int> getNext(int start = 0, int step = 1){
    auto value = start;
    while (true) {
        co_yield value;
        value += step;
    }
}

int main() {

    std::cout << '\n';

```

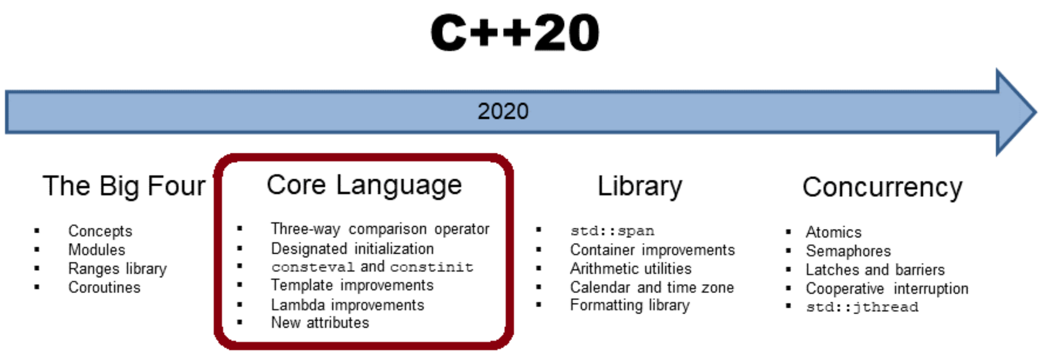
```
13     std::cout << "getNext():";
14     auto gen1 = getNext();
15     for (int i = 0; i <= 10; ++i) {
16         gen1.next();
17         std::cout << " " << gen1.getValue();
18     }
19 
20     std::cout << "\n\n";
21 
22     std::cout << "getNext(100, -10):";
23     auto gen2 = getNext(100, -10);
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 which 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 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]https://devblogs.microsoft.com/cppblog/simplify-your-code-with-rocket-science-c20s-spaceship-operator/

3.2.2 Designated Initialization

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.

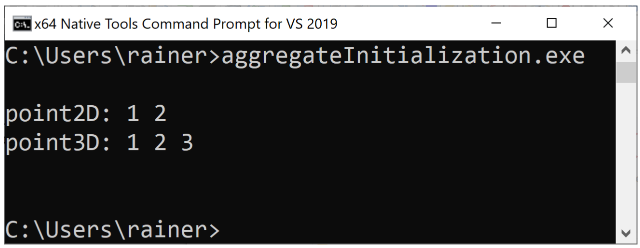
Before I discuss **designated initialization**, let me show more about aggregate initialization. Here is a straightforward example.

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, and you will never notice. 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, kick in.

Designated initialization

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

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

int main(){

```

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

```
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 << " " << p\
23 oint3D.z
24             << "\n";
25 
26 }
```

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, which was added in C++20, creates an immediate function. For an immediate function, every call 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 objects 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, 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
    };
```

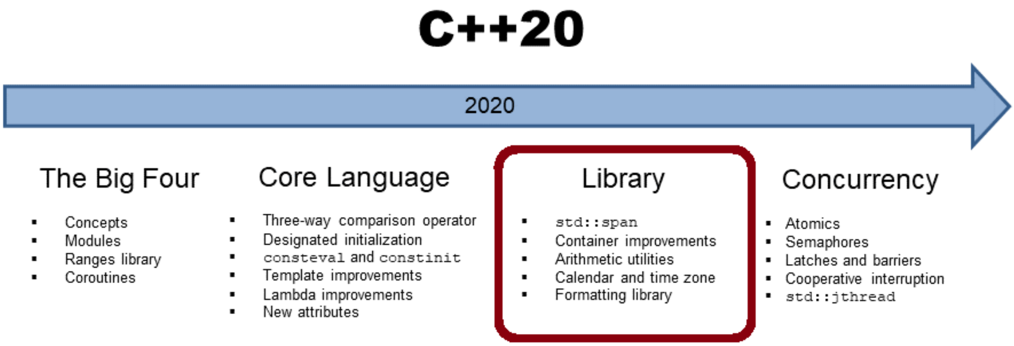
3.2.6 New Attributes

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 Calendar and Time Zones

The chrono library[5] from C++11 is extended with **calendar and time-zone** functionality. The calendar consists of types which represent a year, a month, a day of the week, and an n-th weekday of a month. These elementary types can be combined, forming complex types such as for example `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
```

3.3.5 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

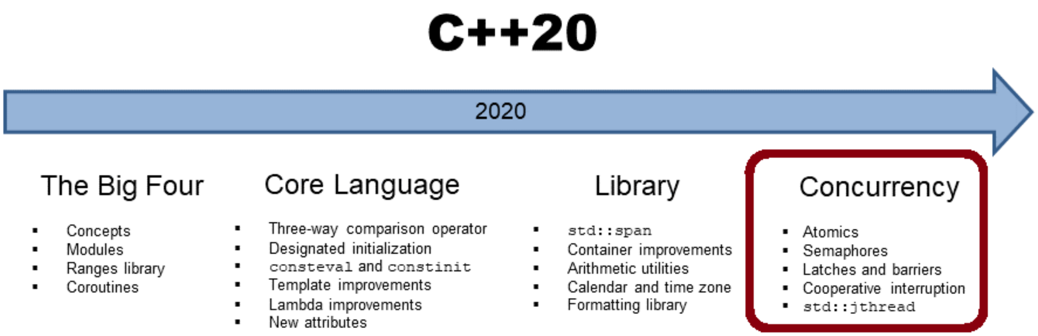
```
        std::string s = std::format("I'd rather be {1} than {0}.", "rig\
ht", "happy");
        // s == "I'd rather be happy than right."
```

- 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

3.4 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_-`

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

`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.

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 is 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 inital 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`

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

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`

```
1  void DoWork(threadpool* pool) {
2  
3      std::latch completion_latch(NTASKS);
4      for (int i = 0; i < NTASKS; ++i) {
5          pool->add_task([&] {
6              // perform work
7              ...
8              completion_latch.count_down();
9          });
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 have been finished.

3.4.4 Cooperative Interruption

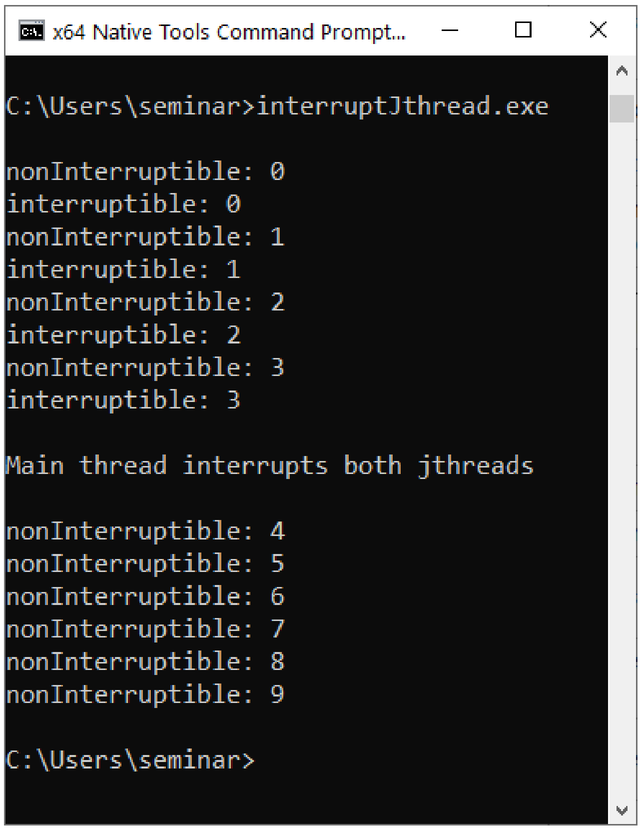
Thanks to **`std::stop_token`**, a `std::jthread` can be interrupted cooperatively.

[9] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4204.html

Interrupting a `std::jthread`

```
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" << std:: endl;
    nonInterruptible.request_stop();
    interruptible.request_stop();

    std::cout << '\n';

}
```

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`

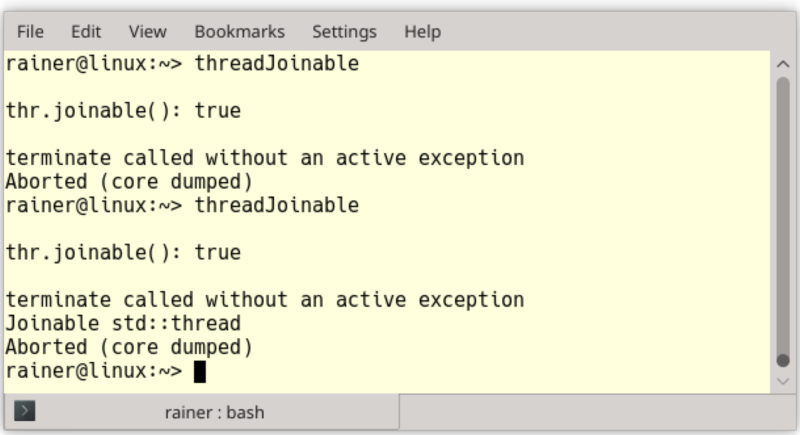
`std::jthread` stands for joining thread. std::jthread extends `std::thread`[10] by automatically joining the started thread. `std::jthread` can also be interrupted.

[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".

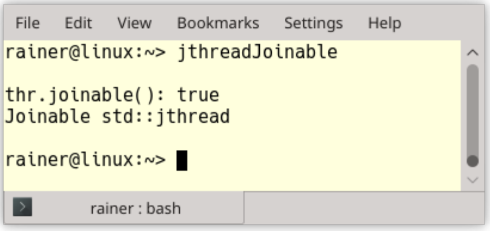
[11] https://en.cppreference.com/w/cpp/error/terminate

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

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';

}
```

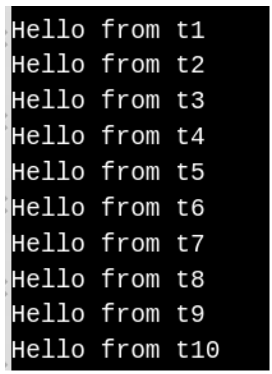
You may get a mess.

Unsynchronized writing to `std::cout`

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

Synchronized writing to `std::cout`

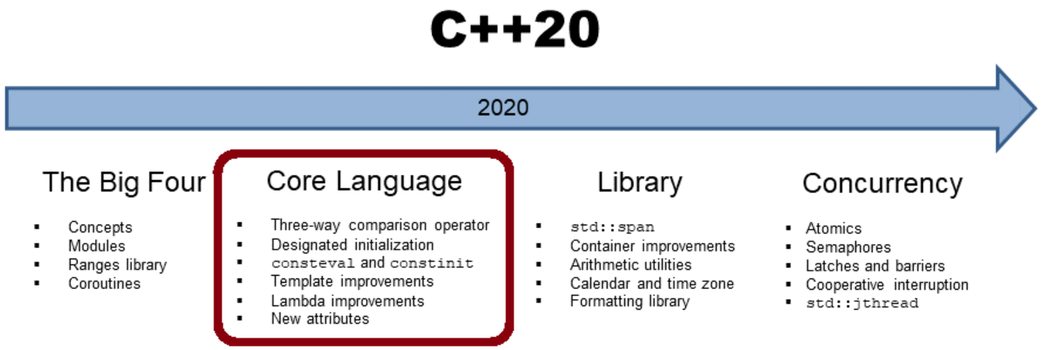
```
void sayHello(std::string name) {
    std::osyncstream(std::cout) << "Hello from " << name <<  '\n';
}
```



Synchronized writing to `std::cout`

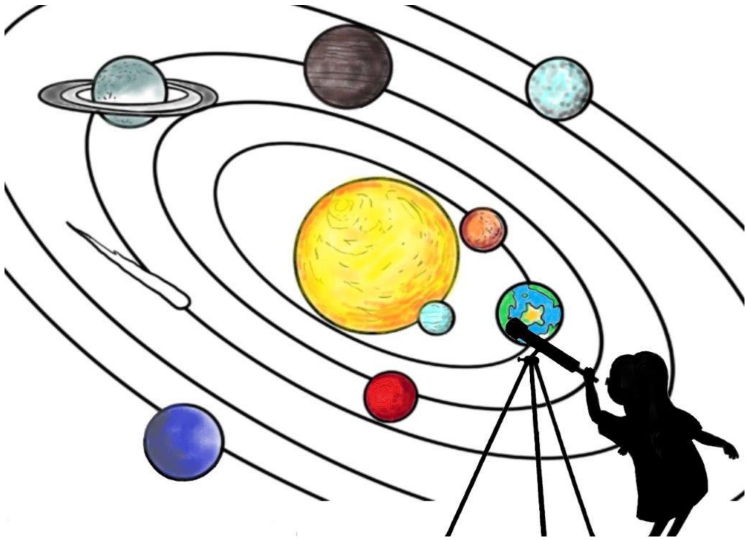
The Details

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 its full extent, I want to start with a short motivation for concepts.

4.1.1 Two Wrong Approaches

Prior to C++20, we had two diametrically 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:

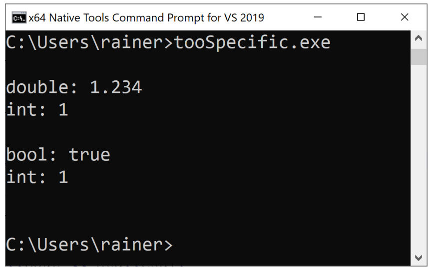
4.1.1.1 Too Specific

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 a 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 also end 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 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 conviction 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`

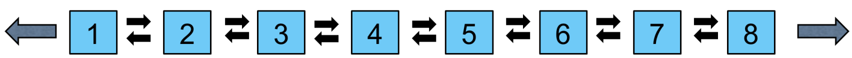
```cpp
// tooGeneric.cpp

#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.

```
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 key 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 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 `LegacyRandomAccessIterato`
- 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`.

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

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:

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 concept `bidirectional_iterator` 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`.

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

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, semantics, and time and space 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. Access to the container's elements is provided through iterators, as well as iterating through them, and the equality comparison of them. The abstraction of the STL is based on concepts such as semi-open range and iterator and allow 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

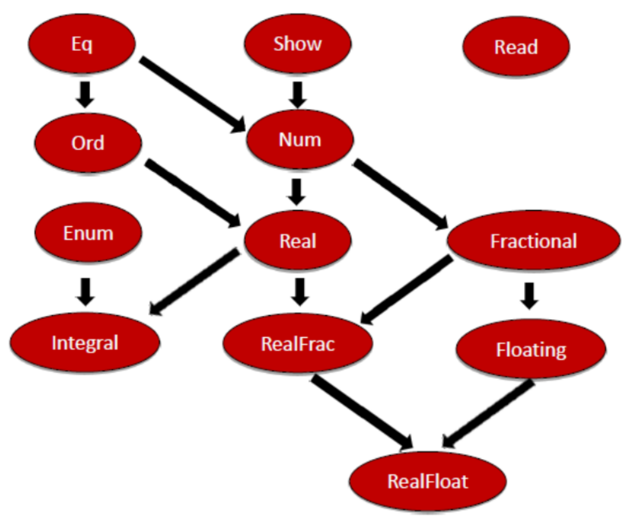
- Requirements for template parameters are part of the interface.

[5] https://www.fm2gp.com/

- 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.
- 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.

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[6] type classes hierarchy.



Haskell Type Classes Hierarchy

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

[6] https://en.wikipedia.org/wiki/Haskell_(programming_language)

- 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 `ints` 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 key 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.*"[7]. A few years later, the next try was also not successful: concepts lite were removed from the C++17 standard. They finally become part of C++20.

4.1.4 Use of Concepts

Essentially, there are four ways to use a concept.

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);
```

[7] https://isocpp.org/blog/2013/02/concepts-lite-constraining-templates-with-predicates-andrew-sutton-bjarne-s

```
11 }
12
13 template<typename T>
14 auto 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 auto gcd2(T a, T b) {
21     if( b == 0 ) return a;
22     else return gcd2(b, a % b);
23 }
24
25 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 int main(){
31
32     std::cout << '\n';
33
34     std::cout << "gcd(100, 10)= "  <<  gcd(100, 10)  << '\n';
35     std::cout << "gcd1(100, 10)= " <<  gcd1(100, 10)  << '\n';
36     std::cout << "gcd2(100, 10)= " <<  gcd2(100, 10)  << '\n';
37     std::cout << "gcd3(100, 10)= " <<  gcd3(100, 10)  << '\n';
38
39     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 the type integral[8]. The function name `gcd` stands for the

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

greatest-common-divisor algorithm based on the Euclidean[9] 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 just `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`, the 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 fulfil 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 which specifies constraints on a template argument (`gcd`) or on a function declaration (`gcd1`). `requires` must be followed by a compile-time predicate such as a named concept (`gcd`), a conjunction/disjunction of named concepts, or a requires expression.

The compile-time predicate can also be an expression:

[9] 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',  // ERR\
18 OR
19
20     std::cout << '\n';
21
22 }
```

The compile-time predicate used in line 6 exemplifies an interesting point: the requirement is applied on 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 parameter 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

```cpp
// 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";
```

```
24
25     static_assert(std::three_way_comparable<std::string>);
26
27     std::cout << "std::three_way_comparable<Test>: ";
28     if constexpr(std::three_way_comparable<Test>) std::cout << "True";
29     else std::cout << "False";
30
31     std::cout << '\n';
32
33     std::cout << "std::three_way_comparable<std::vector<int>>: ";
34     if constexpr(std::three_way_comparable<std::vector<int>>) std::cout\
35  << "True";
36     else std::cout << "False";
37
38     std::cout << '\n';
39
40 }
```

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`[10] (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

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

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

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`, meaning that `T` behaves such as an `int`. The formal definition of `regular` is provided in the define concepts section.

Using a concept in a class definition

```
1  // conceptsClassTemplate.cpp
2
3  #include <concepts>
4  #include <iostream>
5
6  template <std::regular T>
7  class MyVector{};
8
9  int main() {
10
11     MyVector<int> myVec1;
12     MyVector<int&> myVec2;  // ERROR because a reference is not regular
13
14 }
```

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) << '\
24 n';
25
26 }
```

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. The arbitrary 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[11]. In this example, the logical and `&&` (line 7), the logical or `||` (line 10), and the negation of the logical

[11] https://www.modernescpp.com/index.php/fold-expressions

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`[12] 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

```
// conceptsOverloadingFunctionTemplates.cpp

#include <concepts>
#include <iostream>
#include <forward_list>
#include <list>
#include <vector>

template<std::forward_iterator I>
void advance(I& iter, int n){
```

[12] https://en.cppreference.com/w/cpp/iterator/advance

```
11      std::cout << "forward_iterator" << '\n';
12  }
13
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. This 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

```cpp
// conceptsSpecialization.cpp

#include <concepts>
#include <iostream>

template <typename T>
struct Vector {
    Vector() {
        std::cout << "Vector<T>" << '\n';
    }
};

template <std::regular Reg>
struct Vector<Reg> {
    Vector() {
        std::cout << "Vector<std::regular>" << '\n';
    }
};
```

```
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, the uses of the concepts were straightforward, but most of the time more than one concept is used at the same 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

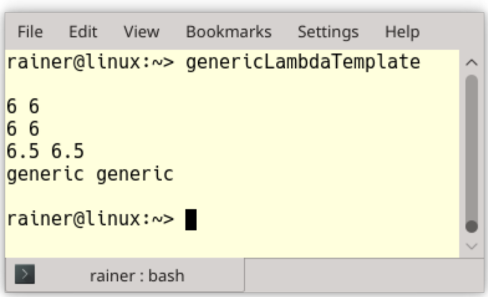
I often have a discussion in my classes, that goes the following way. With C++14, we had generic lambdas. Generic lambdas are lambdas that use `auto` instead of a concrete type.

Comparison of a generic lambda and a function template

```
// genericLambdaTemplate.cpp

#include <iostream>
#include <string>

auto addLambda = [](auto fir, auto sec){ return fir + sec; };

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

int main(){

```

```
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) << '\
17 \n';
18     std::cout << addLambda(1, 5.5) << " " << addTemplate(1, 5.5) << '\n\
19 ';
20
21     const std::string fir{"ge"};
22     const std::string sec{"neric"};
23     std::cout << addLambda(fir, sec) << " " << addTemplate(fir, sec) <<\
24  '\n';
25
26     std::cout << '\n';
27
28 }
```

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 could be. In each place where you can use an unconstrained placeholder `auto`, you can use a concept. I will detail this fully in the section on abbreviated function templates.

4.1.5.2 Placeholders

Use of constrained placeholders instead of unconstrained placeholders

```
// placeholders.cpp

#include <concepts>
#include <iostream>
#include <vector>

std::integral auto getIntegral(int val){
    return val;
}

int main(){

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

    std::vector<int> vec{1, 2, 3, 4, 5};
    for (std::integral auto i: vec) std::cout << i << " ";
    std::cout << '\n';

    std::integral auto b = true;
    std::cout << b << '\n';

    std::integral auto integ = getIntegral(10);
    std::cout << integ << '\n';

    auto integ1 = getIntegral(10);
    std::cout << integ1 << '\n';

    std::cout << '\n';

}
```

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 and this function declaration automatically becomes a function template.

Abbreviated function templates

```cpp
// abbreviatedFunctionTemplates.cpp

#include <concepts>
#include <iostream>

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

template<typename T>
T gcd1(T a, T b) requires std::integral<T> {
    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 becomes, therefore, 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 templates 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 demonstrying it in another example below.

By 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.

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`, on the concept `std::integral`, and on the type `long`.

Abbreviated function templates and overloading

```
// conceptsOverloading.cpp

#include <concepts>
#include <iostream>

void overload(auto t){
    std::cout << "auto : " << t << '\n';
}

void overload(std::integral auto t){
    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[13], and was an experimental implementation of concepts. GCC 6[14] fully implemented the concepts TS. Besides the syntactic differences from concepts in C++20, the concepts TS supported a concise way of defining templates.

In the following example 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.

[13]https://www.iso.org/standard/64031.html

4.1.7 Predefined Concepts

The golden rule "Don't reinvent the wheel" also applies to concepts. The C++ Core Guidelines[15] are very clear about this rule: T.11: Whenever possible, use standard concepts. Consequently, I want to give 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++20 working draft, N4860[16], and finding them all can be quite a challenge! Most of the 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 N4860 also has an index to 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`. This 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.

[14] https://en.wikipedia.org/wiki/GNU_Compiler_Collection
[15] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
[16] https://isocpp.org/files/papers/N4860.pdf

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[17]. Where deemed necessary, I provide additional explanation.

- `same_as`
- `derived_from`
- `convertible_to`
- `common_reference_with`: `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 `common_reference_t<T, U>`
- `common_with`: similar to `common_reference_with`, but the common type `C` is the same as `common_type_t<T, U>` and may not be a reference type
- `assignable_from`
- `swappable`

4.1.7.2.2 Arithmetic Concepts

- `integral`
- `signed_integral`
- `unsigned_integral`
- `floating_point`

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

[17]https://en.cppreference.com/w/cpp/header/type_traits

```
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

- `destructible`
- `constructible_from`
- `default_constructible`
- `move_constructible`
- `copy_constructible`

4.1.7.2.4 Comparison Concepts

- `equality_comparable`
- `totally_ordered`

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

- `movable`
- `copyable`
- `semiregular`
- `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 `movable` requires for `T` that `is_object_v<T>` holds. From the definition of the type-trait `is_object<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 similar to an `int`, and a `regular` type behaves similarly to an `int` and can be compared using ==.

4.1.7.2.6 Callable Concepts

- `invocable`

- `regular_invocable`: a type models `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
- `predicate`: a type models a predicate if it models `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:

- `input_iterator`
- `output_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`
- `contiguous_iterator`

The six categories of iterators correspond to the respective iterator concepts. The table below provides two interesting pieces of information. For the three 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`
		`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::array`
		`std::vector`
		`std::deque`
		`std::string`

The following relation holds: A random-access-iterator is a bidirectional iterator, and a bidirectional iterator is a forward iterator. A contiguous iterator is a random-access-iterator and requires that the elements of the container are stored contiguously in memory. This means `std::array`, `std::vector`, and `std::string`, but not `std::deque`, support contiguous iterators.

4.1.7.4.1 Algorithm Concepts

- `permutable`: in-place reordering of elements is possible
- `mergeable`: merging sorted sequences into an output sequence is possible
- `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

- `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.

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

- `input_range`: specifies a range whose iterator type satisfies `input_iterator` (e.g. can iterate from beginning to end at least once)
- `output_range`: specifies a range whose iterator type satisfies `output_iterator`
- `forward_range`: specifies a range whose iterator type satisfies `forward_iterator` (can iterate from beginning to end more than once)
- `bidirectional_range` : specifies a range whose iterator type satisfies `bidirectional_-iterator` (can iterate forward and backward more than once)
- `random_access_range`: specifies a range whose iterator type satisfies `random_-access_iterator` (can jump in constant time to an arbitrary element with the index operator [])
- `contiguous_range`: specifies a range whose iterator type satisfies `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`
	`It + n` , `It - n`
	`n + It`
	`It - It2`
	`It < It2`, `It <= It2`
	`It > It2`, `It >= It2`
		`std::deque`

`std::ranges::contiguous_range`	`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 container supporting the `std::ranges::contiguous_range` concept, supports all revious 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.

4.1.7.5.2 Views

A `std::ranges::view` typically something that you apply on a range and performs some operation. A view does not own data and the time a view takes to copy, move, or assign is constant. 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.*"

4.1.7.6 Numeric Library

The numeric library provides the concept of a `uniform_random_bit_generator` that is defined in the header `<random>`. A `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 Defining Concepts

When the concept you are looking for is not one of the predefined concepts in C++20, you must define your own concept. In this section I will define a few concepts which 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 for defining 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` 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 (`||`). When building your logical combination, you can negate components by using the exclamation mark (`!`). Thanks to the many compile-time predicates of the type-traits library[18], you have at your disposal all tools required to build powerful concepts.

[18]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`

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

template <typename T>
concept SignedIntegral = Integral<T> && std::is_signed<T>::value;

template <typename T>
concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

I used the type-traits function `std::is_integral`[19] 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.

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

```cpp
// SignedUnsignedIntegrals.cpp

#include <iostream>
#include <type_traits>

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

template <typename T>
concept SignedIntegral = Integral<T> && std::is_signed<T>::value;

template <typename T>
concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;

void func(SignedIntegral auto integ) {
    std::cout << "SignedIntegral: " << integ << '\n';
}

void func(UnsignedIntegral auto integ) {
    std::cout << "UnsignedIntegral: " << integ << '\n';
}

int main() {

    std::cout << '\n';

    func(-5);
```

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

```
28     func(5u);
29 
30     std::cout << '\n';
31 
32 }
```

I used 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_poi\
nt<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

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.

Avoid Anonymous Concepts: `requires requires`

You can define an anonymous concept and directly use it. Avoid it. This makes 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 the template parameters.

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`

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

template <typename>
struct Other;

template <>
struct Other<std::vector<int>> {};

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

int main() {

    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`[20] 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

Compound requirement

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

In addition to a simple requirement, a compound requirement can have a noexcept specifier[21] 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`

```
// conceptsDefinitionEqual.cpp

#include <concepts>
#include <iostream>

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

bool areEqual(Equal auto a, Equal auto b){
  return a == b;
```

[20] https://en.cppreference.com/w/cpp/container/vector
[21] https://en.cppreference.com/w/cpp/language/noexcept_spec

```
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
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 operator. 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`

```cpp
// nestedRequirements.cpp

#include <type_traits>

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

template <typename T>
concept SignedIntegral = Integral<T> && std::is_signed<T>::value;

// template <typename T>
// concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;

template <typename T>
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 sig\
24 ned literal
25 
26 }
```

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

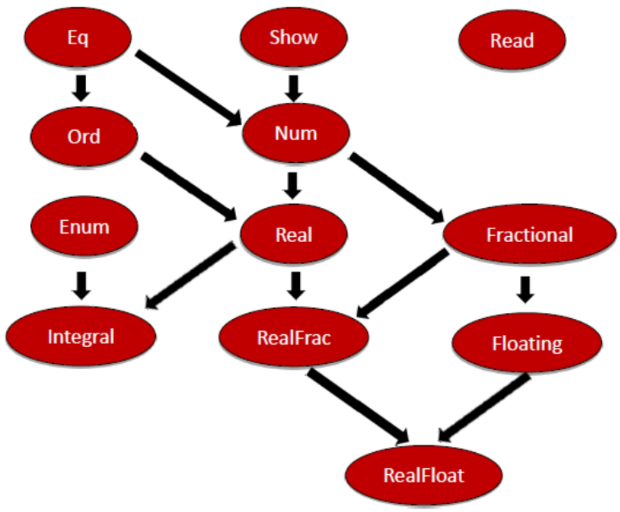
The concept `Ordering` in the following section demonstrates the use of nested requirements.

4.1.9 Application

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`.

4.1.9.1 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`.

4.1.9.1.1 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`

```
// conceptsDefinitionOrdering.cpp

#include <concepts>
#include <iostream>
#include <unordered_set>

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 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>;
    };

template <Equal T>
bool areEqual(const T& a, const T& b) {
    return a == b;
}

template <Ordering T>
T getSmaller(const T& a, const T& b) {
    return (a < b) ? a : b;
}

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, secSe\
47  t) << '\n';
48
49      // auto smallerSet = getSmaller(firSet, secSet);
50
51      std::cout << '\n';
52
53  }
```

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`[22], 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`

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`.

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

```
<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 three-way comparison operator chapter.

4.1.9.2 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`[23]. Now, only one important question is left 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.

---

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

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

`Regular` Types

Alexander Stepanov[24], 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[25], which Alexander Stepanov wrote together with Paul McJones[26], is devoted to `regular` types.

4.1.9.2.1 The Concept `SemiRegular`

A `semiregular` type `X` has to support the Big Six and has to 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()`

[24] https://en.wikipedia.org/wiki/Alexander_Stepanov
[25] http://elementsofprogramming.com/
[26] https://www.mcjones.org/paul/

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

Thanks to the type-traits library[27], defining the corresponding concept is a no-brainer. First, I define the type trait `isSemiRegular` and then use it to define the concept `SemiRegular`.

```
1  template<typename T>
2  struct isSemiRegular: std::integral_constant<bool,
3                                        std::is_default_constructible<T>:\
4  :value &&
5                                        std::is_copy_constructible<T>::va\
6  lue &&
7                                        std::is_copy_assignable<T>::value\
8   &&
9                                        std::is_move_constructible<T>::va\
10 lue &&
11                                       std::is_move_assignable<T>::value\
12  &&
13                                       std::is_destructible<T>::value &&
14                                       std::is_swappable<T>::value >{};
15
16
17 template<typename T>
18 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`.

4.1.9.2.2 The Concept `Regular`

There is only one step and we are done with defining the concept `Regular`. In addition to the requirements of the concept `SemiRegular`, the concept `Regular` requires that

[27] https://en.cppreference.com/w/cpp/header/type_traits

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?

4.1.9.2.3 `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`[28]. 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.

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>::va\
lue &&
                                      std::is_copy_assignable<T>::value\
 &&
                                      std::is_move_constructible<T>::va\
lue &&
                                      std::is_move_assignable<T>::value\
```

28 https://en.cppreference.com/w/cpp/types/is_object

```
 &&
                                      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;
}

struct NotEqualityComparable { };

```

```
54 int main() {
55 
56     int myInt{};
57     behavesLikeAnInt(myInt);
58     behavesLikeAnInt2(myInt);
59 
60     std::vector<int> myVec{};
61     behavesLikeAnInt(myVec);
62     behavesLikeAnInt2(myVec);
63 
64     EqualityComparable equComp;
65     behavesLikeAnInt(equComp);
66     behavesLikeAnInt2(equComp);
67 
68     NotEqualityComparable notEquComp;
69     behavesLikeAnInt(notEquComp);
70     behavesLikeAnInt2(notEquComp);
71 
72 }
```

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[29]. These are essentially the error messages when I use the user-

[29]https://godbolt.org/

defined concept `Regular` (line 64):

```
<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`

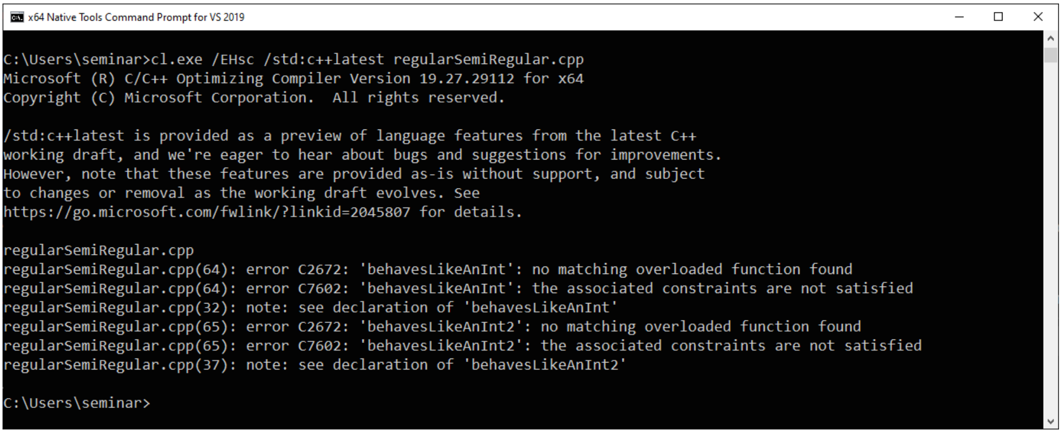
The C++20 concept `std::regular` is more comprehensive. Consequently, the call in line 65 gives a more comprehensive error message:

```
/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.



```
x64 Native Tools Command Prompt for VS 2019

C:\Users\seminar>cl.exe /EHsc /std:c++latest regularSemiRegular.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.27.29112 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

/std:c++latest is provided as a preview of language features from the latest C++
working draft, and we're eager to hear about bugs and suggestions for improvements.
However, note that these features are provided as-is without support, and subject
to changes or removal as the working draft evolves. See
https://go.microsoft.com/fwlink/?linkid=2045807 for details.

regularSemiRegular.cpp
regularSemiRegular.cpp(64): error C2672: 'behavesLikeAnInt': no matching overloaded function found
regularSemiRegular.cpp(64): error C7602: 'behavesLikeAnInt': the associated constraints are not satisfied
regularSemiRegular.cpp(32): note: see declaration of 'behavesLikeAnInt'
regularSemiRegular.cpp(65): error C2672: 'behavesLikeAnInt2': no matching overloaded function found
regularSemiRegular.cpp(65): error C7602: 'behavesLikeAnInt2': the associated constraints are not satisfied
regularSemiRegular.cpp(37): note: see declaration of 'behavesLikeAnInt2'

C:\Users\seminar>
```

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 for 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[30], SFINAE[31], and `static_assert`[32].
- `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, for the first time, to **verify template requirements**. Of course, you can also achieve the verification of template parameters with a combination of type-traits library[33], SFINAE[34], and `static_assert`[35], 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[36] world of ideas. *It is revolutionary that we can now reason about programming in such categories.*

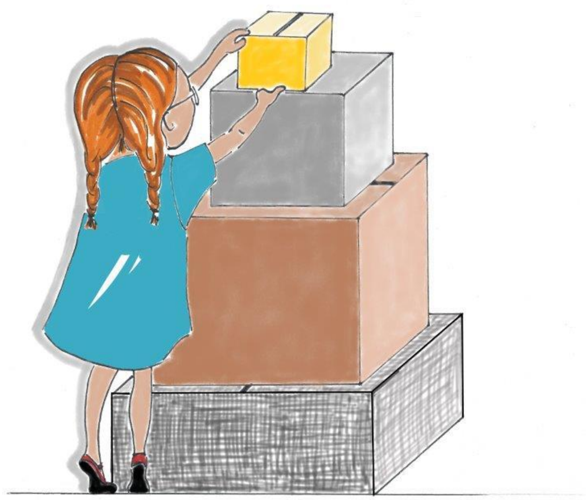
[30]https://en.cppreference.com/w/cpp/header/type_traits

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.
- Concept 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 own 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.

[31] https://en.cppreference.com/w/cpp/language/sfinae
[32] https://en.cppreference.com/w/cpp/language/static_assert
[33] https://en.cppreference.com/w/cpp/header/type_traits
[34] https://en.cppreference.com/w/cpp/language/sfinae
[35] https://en.cppreference.com/w/cpp/language/static_assert
[36] https://en.wikipedia.org/wiki/Plato

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 a lot: shorter compile times, macro isolation, abolishing header files, and avoiding ugly workarounds. Before I present the advantages of modules, I want to step back and explain their benefits.

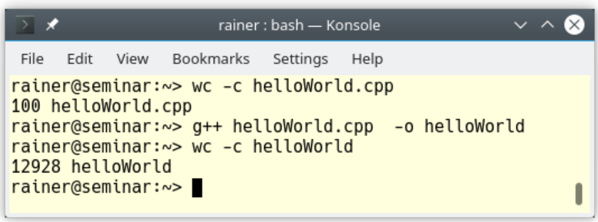
4.2.1 Why do we need Modules?

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

A simple hello world program

```cpp
// helloWorld.cpp

#include <iostream>

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

Making an executable `helloWorld` out of the program `helloWorld.cpp` with GCC[37] increases its size by factor 130.



Size of an object file

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

4.2.1.1 The Classical Build Process

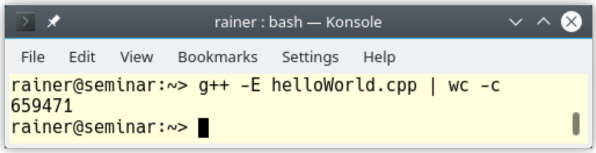
The build process consists of three steps: preprocessing, compilation, and linking.

4.2.1.1.1 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.

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

This straightforward text substitution process can be observed by using the compiler flag `-E` on GCC/Clang, or `/E` on Windows.



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.

4.2.1.1.2 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 it contains the compiled code in binary form. The object file can refer to symbols that don't have a definition. The object files can be put in archives for later reuse. These archives are called static libraries.

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

4.2.1.1.3 Linking

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

This build process that consists of the three steps is inherited from C. It works sufficiently well if you have only one translation unit. But when you have more than one, many issues can occur.

4.2.1.2 Issues of the Build Process

Here's an incomplete list of the flaws in a classical build process, which can be overcome with modules.

4.2.1.2.1 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 that the program has the same external behavior such 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

```cpp
// hello.h

#include <iostream>

void hello();
```

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

Implementation of world

```cpp
// world.cpp

#include "world.h"

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

Header of world

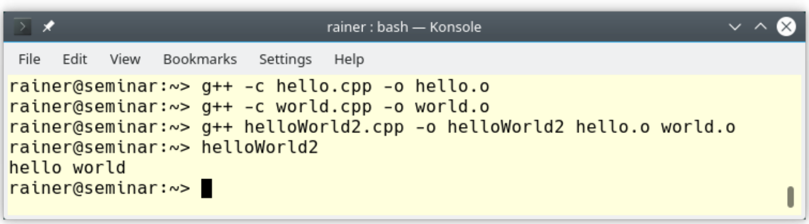
```cpp
// world.h

#include <iostream>

void world();
```

- `helloWorld2.cpp`

Use of hello and world

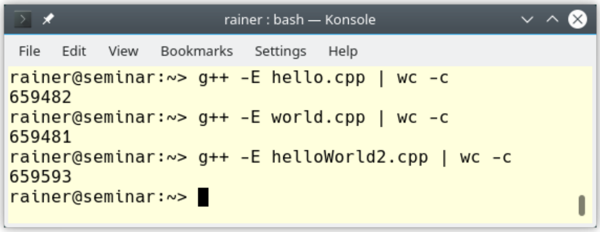
```cpp
// helloWorld2.cpp

#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. This means that the header file `<iostream>` is included a total of three times. Consequently, each source file is blown up to more than 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.

4.2.1.2.2 Isolation from Preprocessor Macros

If there is one consensus in the C++ community, it's the following one: we should get rid of 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 macro `RED` depends on the order of the included header. This dependency is very error-prone.

With modules, import order makes no difference.

4.2.1.2.3 Multiple Definition 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 have to satisfy the requirement that each definition has to be the same.

Let's see what my linker has to say when I try to link a program breaking 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 and, therefore, breaks the one-definition rule because two definitions of `func` are included.

Definition of the function func

```
// header.h

void func() {}
```

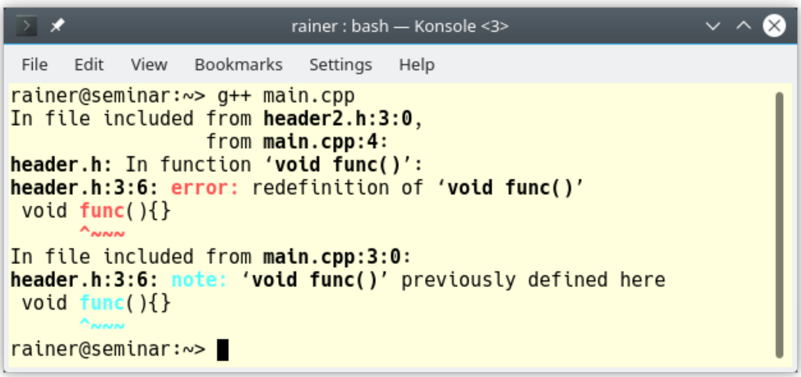
Indirect inclusion of the function definition to func

```
// header2.h

#include "header.h"
```

Double definitions of the function func

```cpp
// main.cpp

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

int main() {}
```

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



Breakion 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

```cpp
// 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 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 Long History

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

In 2004, Daveed Vandevoorde wrote proposal N1736.pdf[38], 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[39]) for modules. In 2019, the Modules TS and the ATOM proposal were merged into the C++20 committee draft (N4842[40]).

---

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

4.2.3 A First Example

The purpose of this section is straightforward: I will give you an introduction to modules. More advanced features of modules are in the following sections. 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 function `add`'s definition, `add` is exported and can, therefore, be used by a consumer of the module.

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 module visible to `client.cpp`.

Let me start with the module declaration file.

4.2.3.1 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** originally used the extension `cppm`. The `m` in the suffix probably stands for module. This convention changes in newer versions of Clang to the `cpp` extension.
- The **GCC compiler** uses no special extension.

The global module fragment is meant to compose module interfaces. It 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 interface can compile. The code in the global module fragment is not exported by the module interface.

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

A module definition with a global module fragment

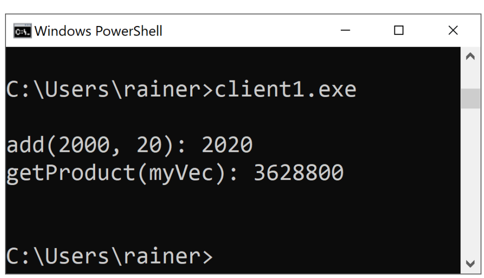
```
// math1.ixx

module;

#include <numeric>
#include <vector>

export module math;

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

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

I included the necessary headers between 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 module `math` and uses its functionality:



Execution of the program `client1.exe`

Now, let's dive into the details.

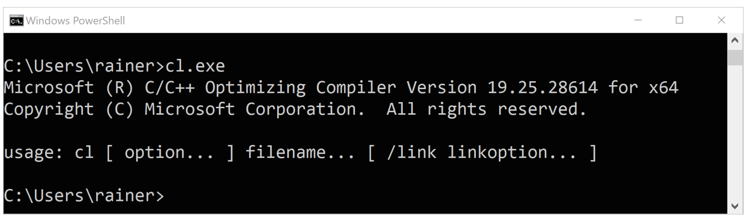
4.2.4 Compilation and Use

To compile the module `math.ixx` used by the client program `client.cpp`, you have to use a very recent Clang, GCC, or Microsoft compiler.

The compilation of a module is challenging. For that reason, I show as an example the compilation of the module with the Microsoft compiler and the Clang compiler.

4.2.4.1 Microsoft Visual Compiler

First, I use the cl.exe 19.25.28614 for 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 follow. Additionally, with an older Microsoft compiler, you have to use the flag `/std:cpplatest`.

Building the executable with the Microsoft compiler

```
1 cl.exe /experimental:module /c math.ixx
2 cl.exe /experimental:module client.cpp math.obj
```

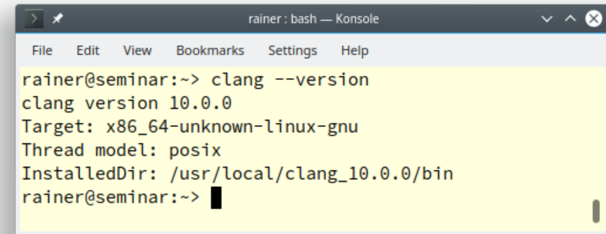
- Line 1 creates an obj file `math.obj` and an IFC file `math.ifc`. The IFC file contains the metadata description of the module interface. The binary format of the IFC is modeled after the Internal Program Representation[41] by Gabriel Dos Reis and Bjarne Stroustrup (2004/2005).
- Line 2 creates the executable `client.exe`. Without the implicitly used `math.ifc` file from the first step, the linker cannot find the module.

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

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

4.2.4.2 Clang Compiler

On Linux, I use the **Clang 10.0.0** compiler.



Clang compiler for modules

With the clang compiler, the module declaration file is simply a `cpp` file. Consequently, I have to rename the `math.ixx` file to `math.cpp`.

A simple math module

```
// math.cpp

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++2a -stdlib=libc++ -c math.cpp -Xclang -emit-module-inte\
2 rface \
3         -o math.pcm
4 
5 clang++ -std=c++2a -stdlib=libc++ -fprebuilt-module-path=. client.cpp m\
6 ath.pcm \
7         -o client
```

- Line 1 creates the module `math.pcm`. The suffix `pcm` stands for precompiled module. The flags `-std=c++2a` specifies the working draft of the C++20 standard and the `-stdlib=libc++` the used C++ standard library. The flag combination `-Xclang -emit-module-interface` is necessary for creating the precompiled module.
- Line 4 creates the executable `client`, which uses the module `math.pcm`. You specify the path to the module with the `-fprebuilt-module-path` flag.

4.2.4.3 Used Compiler

I use the cl.exe from Microsoft in this book. Microsoft currently has (end 2020) the best support for modules[42]. The Microsoft blog provides two excellent introductions to modules: Overview of modules in C++[43] and C++ Modules conformance improvements with MSVC in Visual Studio 2019 16.5[44]. Neither Clang nor GCC provide similar introductions, making it quite difficult to use modules with those compilers.

4.2.5 Export

There are three ways to export names in a module interface unit.

4.2.5.1 Export Specifier

You can export each name explicitly.

[42] https://en.cppreference.com/w/cpp/compiler_support
[43] https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-160&viewFallbackFrom=vs-2019
[44] https://devblogs.microsoft.com/cppblog/c-modules-conformance-improvements-with-msvc-in-visual-studio-2019-16-5/

Export specifier

```
export module math;

export int mult(int fir, int sec);

export void doTheMath();
```

4.2.5.2 Export Group

An export group exports all of its names.

Export group

```
export module math;

export {

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

}
```

4.2.5.3 Export Namespace

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

Export namespace

```
export module math;

export namespace math {

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

}
```

When a client uses names from an export namespace, they have to qualify those names.

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

4.2.6 Guidelines for a Module Structure

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

Guidelines for the structure of a module

```
module;                          // global module fragment

#include <headers for libraries not modularized so far>

export module math;              // module declaration; starts the module p\
urview

import <importing of other modules>

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

export namespace math {

    <exported declarations>   // exported names

}
```

This guideline serves one purpose: give you a simplified structure of a module and also 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 is the right place to include headers.
- The module declaration `export module math` starts the so-called module purview, which ends at the end of the translation unit.
- 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.

4.2.7 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.

4.2.7.1 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.

4.2.7.2 Module Implementation Unit

The module implementation unit

```
// mathImplementationUnit.cpp

module math;

#include <numeric>

namespace math {

```

```
 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::multipli\
15 es<int>());
16     }
17 }
```

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

4.2.7.3 Main Program

The client uses module math

```
// client3.cpp

#include <iostream>
#include <vector>

import math;

int main() {

    std::cout << '\n';

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

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

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

```
18
19     std::cout << '\n';
20
21 }
```

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, in general, highly valuable if you want to try it out.

Building the Executable with the Microsoft Compiler

Manually building the executable includes a few steps.

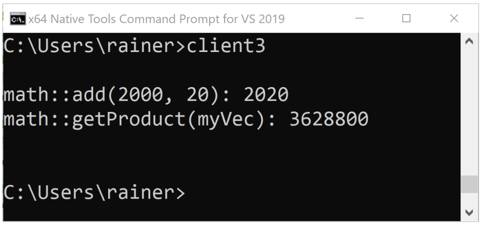
Building a module with a module interface unit and a module implementation unit

```
1 cl.exe /c /experimental:module mathInterfaceUnit.ixx /EHsc
2 cl.exe /c /experimental:module mathImplementationUnit.cpp /EHsc
3 cl.exe /c /experimental:module 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, you have to specify the exception handling model (`/EHsc`), and enable modules: `/experimental:module`.

Finally, here is the output of the program:



Execution of the program `client2.exe`

4.2.8 Submodules and Module Partitions

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

4.2.8.1 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;
    }
}
```

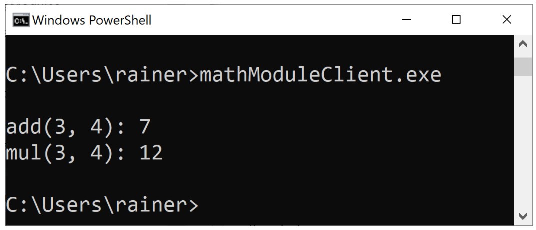
If you look carefully, you recognize a small difference in the `export` statements in the modules `math`. While `math.math1` uses an export specifier, `math.math2` uses an export group or export block.

From the client's perspective, using the `math` module is straightforward.

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 /experimental:module mathModule1.ixx /EHsc
cl.exe /c /experimental:module mathModule2.ixx /EHsc
cl.exe /c /experimental:module mathModule.ixx /EHsc
cl.exe /EHsc /experimental:module mathModuleClient.cpp mathModule1.obj \
mathModule2.obj mathModule.obj
```

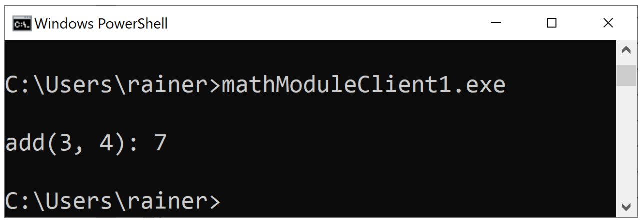
Each compilation process of the three modules creates two artifacts: The IFC file (interface file) `*.ifc`, which is implicitly used in the last line, and the `*.obj file`, which is explicitly used in the last line.

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 only interested 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.

4.2.8.2 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 names that the partitions export are imported and re-exported by the primary module interface unit (primary interface file). The names of a partition must begin with the name of the module. The partitions cannot exist on their own.

The description of module partitions is more difficult 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 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

```
// mathPartition1.ixx

export module math:math1;

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

Second module partition

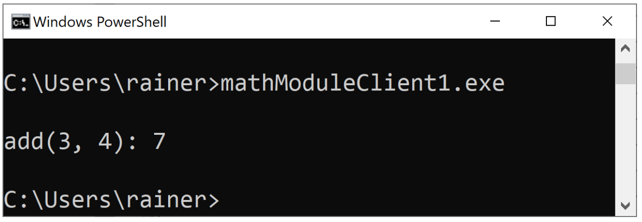
```
// mathPartition2.ixx

export module math:math2;

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

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. The name `math` stands for the module and the names `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 client program 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

4.2.9 Templates in Modules

I often hear the question: How are templates exported by modules? When you instantiate a template, its definition must be available. This is the reason that template definitions are hosted in headers. Conceptually, the usage of a template has the following structure

4.2.9.0.1 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 directly 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[45].

4.2.9.1 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 an abstract syntax tree[46] (AST). Thanks to this AST, the template definition is

[45] https://cppinsights.io/
[46] https://en.wikipedia.org/wiki/Abstract_syntax_tree

available during template instantiation.

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

- `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;
    }

}
```

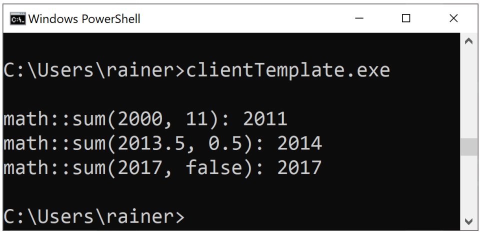
- `clientTemplate.cpp`

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';

}
```

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.

4.2.10 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 that are 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.

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 small 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

```
// mathModuleTemplate1.ixx

module;

#include <iostream>
#include <typeinfo>
#include <utility>

export module math;

template <typename T>
auto showType(T&& t) {
    return typeid(std::forward<T>(t)).name();
}

export namespace math {

    template <typename T, typename T2>
    auto sum(T fir, T2 sec) {
        auto res = fir + sec;
        return std::make_pair(res, showType(res));
    }

}
```

Instead of the sum of the numbers, the function template `sum` returns a `std::pair`[47]

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

(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 not possible. Function template `showType` uses perfect forwarding[48] to preserve the value categories of the function argument `t`. The `typeid`[49] operator queries information about the type at run time (run time type identification (RTTI)[50]).

Use of the improved function template `sum`

```cpp
// 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: " << messag\
e << '\n';

    auto [val1, message1] =  math::sum(2013.5, 0.5);
    std::cout << "math::sum(2013.5, 0.5): " << val1 << "; type: " << me\
ssage1
              << '\n';

    auto [val2, message2] =  math::sum(2017, false);
    std::cout << "math::sum(2017, false): " << val2 << "; type: " << me\
ssage2
              << '\n';

}
```

[48] https://www.modernescpp.com/index.php/perfect-forwarding
[49] https://en.cppreference.com/w/cpp/language/typeid
[50] https://en.cppreference.com/w/cpp/types

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

4.2.11 Header Units

At the end of 2020, no compiler, so far, supports header units. Header units are a smooth way to transition from headers to modules. You just have to replace the `#include` directive with the new `import` directive.

Replacing #include directives with import directives

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

First, import respects the same lookup rules as `include`. This 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 out of the `import` directive and treats the result as if it would be 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 to precompiled headers.

4.2.11.1 One Drawback

There is one drawback with header units. Not all headers are importable. Which headers are importable is implementation-defined[51], but the C++ standard guarantees that

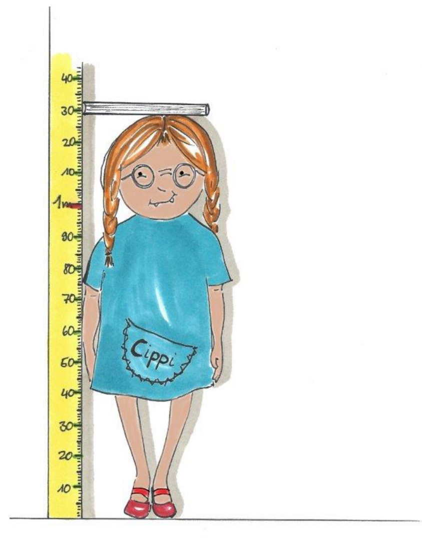
[51] https://en.cppreference.com/w/cpp/language/ub

all standard library headers are importable headers. The ability to import excludes C headers. They are just wrapped in the `std` namespace. For example <cstring> is the C++ wrapper for <string.h>. You can easily identify the wrapped C header because the pattern is: `xxx.h` gets `cxxx`.

Distilled Information

- Modules overcome the deficiencies of headers and macros, in particular. Their import is literally for free, and in contrast to macros, the sequence in which you import does not matter. Additionally, they overcome name collisions.
- A module consists of a module interface unit and a module implementation unit. There must be one module interface unit having the exporting module declaration and arbitrarily many module implementation units. 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. Building your modules is with C++20 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.

4.3 Three-Way Comparison Operator



Cippi measures how big she is

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. You can define the spaceship operator or the compiler can autogenerates it for you.

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

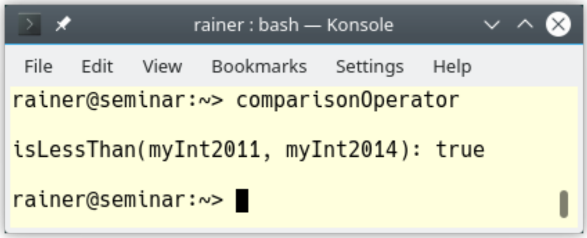
4.3.1 Ordering before C++20

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

```cpp
// comparisonOperator.cpp

#include <iostream>

struct MyInt {
    int value;
    explicit 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 three-way comparison operator.

4.3.2 Ordering since C++20

You can define the three-way comparison operator or request it from the compiler with `= default`. In both cases you automatically get all six comparison operators: ==,

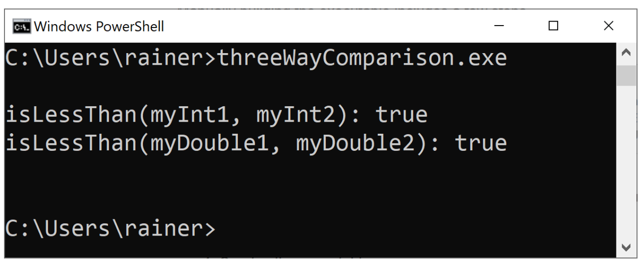
!=, <, <=, >, and >=.

Implement or request the three-way comparison operator

```cpp
// threeWayComparison.cpp

#include <compare>
#include <iostream>

struct MyInt {
    int value;
    explicit MyInt(int val): value{val} { }
    auto operator<=>(const MyInt& rhs) const {
        return value <=> rhs.value;
    }
};

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';

    MyInt myInt1(2011);
    MyInt myInt2(2014);

    std::cout << "isLessThan(myInt1, myInt2): "
              << 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.

Automatic Comparision of Pointers

The compiler-generated three-way 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 Comparision Categories

The names of the three comparison categories are strong ordering, weak ordering, and partial 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)

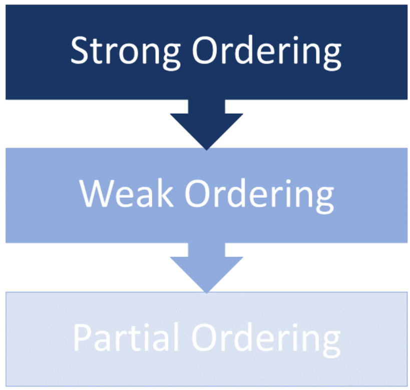
When you use as a sorting criterion the case-insensitive representation of a string, equivalent values need not be different. Additionally, two arbitrary floating-point values need not to be comparable: for `a = 5.5`, and `b = NaN` (Not a Number) neither of the following expressions returns `true`: `a < Nan`, `a == Nan` , or `a > Nan`.

Based on the three properties, distinguishing the three comparison strategies 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

If the declared return type is `auto`, then 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 = defaul\
t;
};

```

```
18 struct StrongWeakPartial {
19 
20     Strong s;
21     Weak w;
22     Partial p;
23 
24     auto operator <=> (const StrongWeakPartial&)  const = default;
25 
26     // FINE
27     // std::partial_ordering operator <=> (const StrongWeakPartial&) co\
28 nst = default;
29 
30     // ERROR
31     // std::strong_ordering operator <=> (const StrongWeakPartial&) con\
32 st = default;
33     // std::weak_ordering operator <=> (const StrongWeakPartial&) const\
34  = default;
35 
36 };
37 
38 int main() {
39 
40     StrongWeakPartial a1, a2;
41 
42     a1 < a2;
43 
44 }
```

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.

Now, I want to focus on the compiler-generated spaceship operator.

4.3.4 The Compiler-Generated Spaceship Operator

The compiler-generated three-way comparison operator needs the header `<compare>`, is implicitly `constexpr` and `noexcept`[52], and performs a lexicographical comparison.

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.

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';
```

[52]https://www.modernescpp.com/index.php/c-core-guidelines-the-noexcept-specifier-and-operator

```
24     else if (res2 > 0) std::cout << "str1 > str2" << '\n';
25 
26     std::vector<int> vec1{1, 2, 3};
27     std::vector<int> vec2{1, 2, 3};
28     auto res3 = vec1 <=> vec2;
29     if (res3 < 0) std::cout << "vec1 < vec2" << '\n';
30     else if (res3 == 0) std::cout << "vec1 == vec2" << '\n';
31     else if (res3 > 0) std::cout << "vec1 > vec2" << '\n';
32 
33     std::cout << '\n';
34 
35 }
```

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 done at compile time.

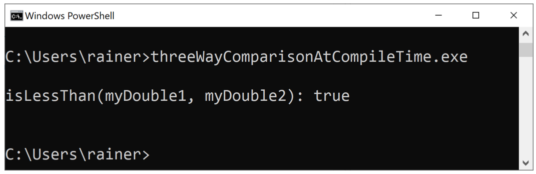
4.3.4.2 Comparison at Compile Time

The three-way comparison operator is implicitly `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

```
1  // threeWayComparisonAtCompileTime.cpp
2
3  #include <compare>
4  #include <iostream>
5
6  struct MyDouble {
7      double value;
8      explicit constexpr MyDouble(double val): value{val} { }
9      auto operator<=>(const MyDouble&) const = default;
10 };
11
12 template <typename T>
13 constexpr bool isLessThan(const T& lhs, const T& rhs) {
14     return lhs < rhs;
15 }
16
17 int main() {
18
19     std::cout << std::boolalpha << '\n';
20
21     constexpr MyDouble myDouble1(2011);
22     constexpr MyDouble myDouble2(2014);
23
24     constexpr bool res = isLessThan(myDouble1, myDouble2);
25
26     std::cout << "isLessThan(myDouble1, myDouble2): "
27               << res << '\n';
28
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 three-way comparison operator performs a 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 == and != operator behave 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"[53] from the Microsoft C++ Team Blog provides an impressive example of lexicographical comparison. For readability, I added a few comments.

Lexicographical comparison

```
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;
```

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

```
15 };
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. },                      // Ba\
23 sics
24                         { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, // Ar\
25 rays
26                         { { 1., 2. }, { 3., 4. } } } };
27   constexpr Bases b = { { 0, 'c', 1.f, 1. },                      // Ba\
28 sics
29                         { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, // Ar\
30 rays
31                         { { 1., 2. }, { 3., 4. } } } };
32   static_assert(a == b);
33   static_assert(!(a != b));
34   static_assert(!(a < b));
35   static_assert(a <= b);
36   static_assert(!(a > b));
37   static_assert(a >= b);
38 }
```

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 a string-like or vector-like types. In this case, a `==` and `!=` may be faster than the compiler-generated three-way 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. The standardization committee was aware of this performance issue and fixed it with the paper P1185R2[54]. 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.

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.

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:

[54] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1185r2.html

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 < myInt2\
014" << '\n';

    std::cout << '\n';

    if (myInt2011 < int2014) std:: cout << "myInt2011 < int2014" << '\n\
';
    if ((myInt2011 <=> int2014) < 0) std:: cout << "myInt2011 < int2014\
" << '\n';

```

```
36      std::cout << '\n';
37
38      if (int2011 < myInt2014) std::cout << "int2011 < myInt2014" << '\n'\
39  ;
40      if (0 < (myInt2014 <=> int2011)) std:: cout << "int2011 < myInt2014\
41  " << '\n';
42
43      std::cout << '\n';
44
45  }
```

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 such 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

```
// threeWayComparisonForInt.cpp

#include <compare>
#include <iostream>

class MyInt {
 public:
    constexpr explicit MyInt(int val): value{val} { }

    auto operator<=>(const MyInt& rhs) const = default;

    constexpr auto operator<=>(const int& rhs) const {
        return value <=> rhs;
    }
 private:
    int value;
};

template <typename T, typename T2>
constexpr bool isLessThan(const T& lhs, const T2& rhs) {
    return lhs < rhs;
}

int main() {

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

    constexpr MyInt myInt2011(2011);
    constexpr MyInt myInt2014(2014);

    constexpr int int2011(2011);
    constexpr int int2014(2014);

    std::cout << "isLessThan(myInt2011, myInt2014): "
              << isLessThan(myInt2011, myInt2014) << '\n';
```

```
36
37     std::cout << "isLessThan(int2011, myInt2014): "
38               << isLessThan(int2011, myInt2014) << '\n';
39
40     std::cout << "isLessThan(myInt2011, int2014): "
41               << isLessThan(myInt2011, int2014) << '\n';
42
43     constexpr auto res = isLessThan(myInt2011, int2014);
44
45     std::cout << '\n';
46
47 }
```

I defined in (line 10) the three-way comparison operator and declared it `constexpr`. The user-defined three-way comparison operator is not implicitly `constexpr`, unlike the compiler-generated three-way comparison operator. 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, thanks to reordering, you have to define only 2 operators to get 18 = 3 * 6 combinations of comparison operators. The 3 stands for the combinations `int OP MyInt`, `MyInt OP MyInt`, and `MyInt OP int` and the 6 for six comparison operators.

4.3.6 User-Defined and Auto-Generated Comparison Operators

When you can define one of the six comparison operators and also 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 also the compiler-generated six comparison operators.

Let's see what happens.

The interplay of user-defined and auto-generated operators

```cpp
// 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;
```

```
32     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) out of 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 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 six comparison operators. The compiler-generated comparison operators 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 both 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 and class types. A class type is a class, a struct, or a union.

With C++20, the following condition must hold for class types supporting 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

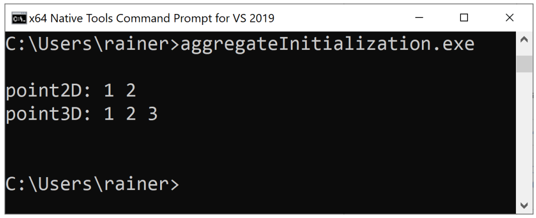
The next program exemplifies aggregate initialization.

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.

In the section covering the three-way comparison operator is a more sophisticated example of aggregate initialization.



Aggregate initialization

Based on aggregate initialization in C++11, we get designed initializers in C++20. At the end of 2020, only the Microsoft compiler supports designated initialization completely.

4.4.2 Named Initialization of Class Members

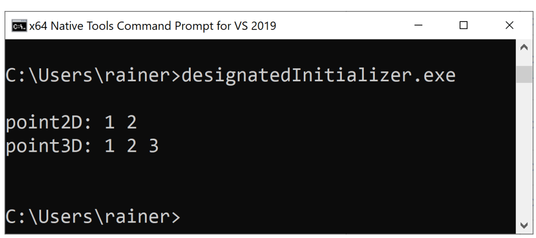
Designated initialization enables the direct initialization of members of a class type using their names. For a union, only one initializer can be provided. As for aggregate initialization, the sequence of initializers in the curly braces has to match the declaration order of the members.

Designated initialization

```
// designatedInitializer.cpp

#include <iostream>

struct Point2D{
    int x;
    int y;
};

class Point3D{
public:
    int x;
    int y;
    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 such as `.x` or `.y` are often called designators.



Designated Initializers

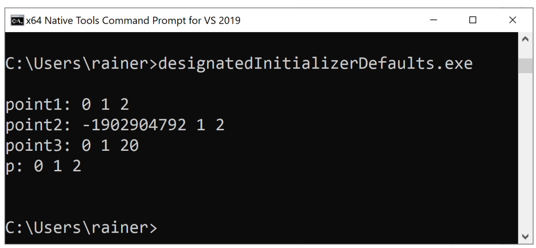
The members of the aggregate can already have a default value. This default value is used when the initializer is missing. This does not hold for a union.

Designated initializers with defaults

```cpp
// designatedInitializersDefaults.cpp

#include <iostream>

class Point3D{
public:
    int x;
    int y = 1;
    int z = 2;
};

void needPoint(Point3D p) {
     std::cout << "p: " << p.x << " " << p.y << " " << p.z << '\n';
}

int main(){

    std::cout << '\n';

    Point3D point1{.x = 0, .y = 1, .z = 2};
    std::cout << "point1: " << point1.x << " " << point1.y << " "
                            << point1.z << '\n';

    Point3D point2;
    std::cout << "point2: " << point2.x << " " << point2.y << " "
                            << point2.z << '\n';

    Point3D point3{.x = 0, .z = 20};
    std::cout << "point3: " << point3.x << " " << point3.y << " "
                            << point3.z << '\n';

    // Point3D point4{.z = 20, .y = 1}; ERROR

    needPoint({.x = 0});

```

```
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 line 34. The expression in line 32 would not compile because `z` and `y` are in the wrong order.



Designated initializers with defaults

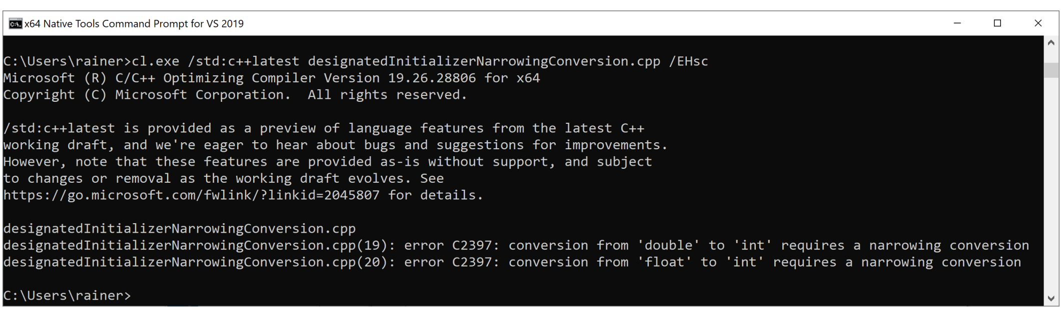
Designated initializers detect narrowing conversions. Narrowing conversion results in the loos of precision.

Designated initializers detect narrowing conversion

```cpp
// designatedInitializerNarrowingConversion.cpp

#include <iostream>

struct Point2D{
    int x;
    int y;
};

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

```
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 line 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[55] 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)
```

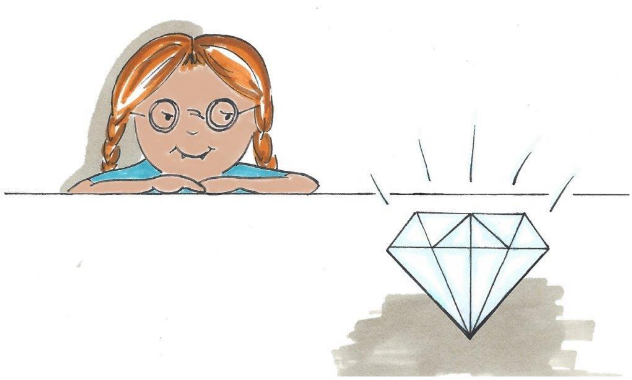
The rationale for this difference between C and C++ is also part of the proposal: "*In C++, members are destroyed in reverse construction order and the elements of an initializer list are evaluated in lexical order, so field initializers must be specified in order. Array designators conflict with lambda-expression syntax. Nested designators are seldom used.*" The paper continues to argue that only out-of-order initialization of an aggregate is commonly used.

Distilled Information

- Designated initialization is a special case of aggregate initialization and enables it to initialize the class members using their name. The initialization order must match the declaration order.

[55] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0329r4.pdf

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 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

```
// constevalSqr.cpp

#include <iostream>

consteval int sqr(int n) {
    return n * n;
}

int main() {

    std::cout << "sqr(5): " << sqr(5) << '\n';

```

```
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

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 its 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.

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

Initialization with constinit

```cpp
// 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

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 Function Execution

The following program `consteval.cpp` has three versions of a `square` function.

Three versions of a square function

```
1  // consteval.cpp
2  
3  #include <iostream>
4  
5  int sqrRunTime(int n) {
6      return n * n;
7  }
8  
9  consteval int sqrCompileTime(int n) {
10     return n * n;
11 }
12 
13 constexpr int sqrRunOrCompileTime(int n) {
14     return n * n;
15 }
16 
17 int main() {
18 
19     // constexpr int prod1 = sqrRunTime(100); ERROR
20     constexpr int prod2 = sqrCompileTime(100);
21     constexpr int prod3 = sqrRunOrCompileTime(100);
22 
23     int x = 100;
24 
25     int prod4 = sqrRunTime(x);
26     // int prod5 = sqrCompileTime(x); ERROR
27     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, accordingly, 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);                  // c\
2  ompile time
3  int arrayNewWithConstExpressiomFunction[sqrRunOrCompileTime(100)]; // c\
4  ompile time
5  constexpr int prod = sqrRunOrCompileTime(100);                   // c\
6  ompile time
7
8  int a = 100;
9  int runTime = sqrRunOrCompileTime(a);                   // run time
10
11 int runTimeOrCompiletime = sqrRunOrCompileTime(100);   // run time or co\
12 mpile time
13
14 int alwaysCompileTime = sqrCompileTime(100);           // compile time
```

The 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.

4.5.4 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
22     // constinit auto localConstinit = 1000; ERROR
23
24 }
```

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

4.5.5 Solving the Static Initialization Order Fiasco

According to the FAQ at isocpp.org[56], 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.

4.5.5.1 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 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.

4.5.5.1.1 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 were zero-initialized.

[56] https://isocpp.org/wiki/faq/ctors#static-init-order

The static initialization order fiasco

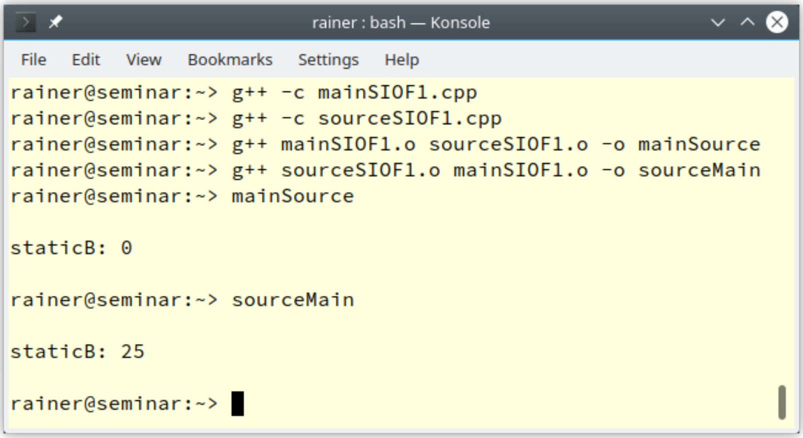
```cpp
// sourceSIOF1.cpp

int square(int n) {
    return n * n;
}

auto staticA  = square(5);
```

The static initialization order fiasco

```cpp
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?

4.5.5.1.2 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[57] singleton is based on this additional guarantee.

The lazy initialization can also be used to overcome the static initialization order fiasco.

[57]https://en.wikipedia.org/wiki/Scott_Meyers

Lazy initialization of a static with local scope

```
1  // sourceSIOF2.cpp
2
3  int square(int n) {
4      return n * n;
5  }
6
7  int& staticA() {
8
9      static auto staticA  = square(5);
10     return staticA;
11
12 }
```

Lazy initialization of a static with local scope

```
1  // mainSOIF2.cpp
2
3  #include <iostream>
4
5  int& staticA();
6
7  auto staticB = staticA();
8
9  int main() {
10
11     std::cout << '\n';
12
13     std::cout << "staticB: " << staticB << '\n';
14
15     std::cout << '\n';
16
17 }
```

`staticA` (line 9 in file `sourceSIOF2.cpp`) is, in this case, a static in a local scope. The 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.

4.5.5.1.3 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

```cpp
// sourceSIOF3.cpp

constexpr int square(int n) {
    return n * n;
}

constinit auto staticA = square(5);
```

Compile-time initialization of a static

```
1  // mainSOIF3.cpp
2
3  #include <iostream>
4
5  extern constinit int staticA;
6
7  auto staticB = staticA;
8
9  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.

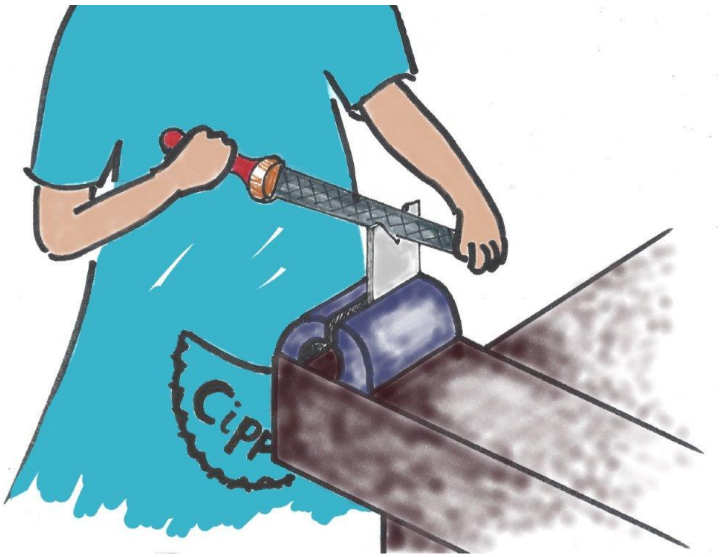
Solving the static initializaion order fiasco with `constinit`

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 you are 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

```
// implicitExplicitGenericConstructor.cpp

#include <iostream>
#include <string>

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';
    }
};

int main() {

    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

```
1  // conditionallyConstructor.cpp
2
3  #include <iostream>
4  #include <type_traits>
5  #include <typeinfo>
6
7  struct MyBool {
8      template <typename T>
9      explicit(!std::is_same<T, bool>::value) MyBool(T t) {
10         std::cout << typeid(t).name() << '\n';
11     }
12 };
13
14 void needBool(MyBool b){ }
15
16 int main() {
17
18     MyBool myBool1(true);
19     MyBool myBool2 = false;
20
21     needBool(myBool1);
22     needBool(true);
23     // needBool(5);
24     // needBool("true");
25
26 }
```

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[58]. 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).

[58] https://en.cppreference.com/w/cpp/header/type_traits

4.6.2 Non-Type Template Parameters

C++ supports non-types as template parameters. Essentially non-types could be

- integers and enumerators
- pointers or references to objects, to functions and to attributes of a class
- `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-Points and Literal Types

Literal Types have 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. The following program uses floating-point types and literal types as non-type template parameters.

Floating-points and literal types as non-type template parameters

```
1  // nonTypeTemplateParameter.cpp
2
3  struct ClassType {
4      constexpr ClassType(int) {}
5  };
6
7  template <ClassType cl>
8  auto getClassType() {
9      return cl;
10 }
11
12 template <double d>
13 auto getDouble() {
14     return d;
15 }
16
17 int main() {
18
19     auto c1 = getClassType<ClassType(2020)>();
20
21     auto d1 = getDouble<5.5>();
22     auto d2 = getDouble<6.5>();
23
24 }
```

`ClassType` has a `constexpr` constructor (line 4) and can, therefore, be used as a template argument (line 19). The same holds for the function template `getDouble` (line 13), which accepts only `double`. 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++20, strings can be used as non-type template arguments.

4.6.2.2 String Literals

The class `StringLiteral` has a `constexpr` constructor.

String literals as non-type template parameters

```cpp
// nonTypeTemplateParameterString.cpp

#include <algorithm>
#include <iostream>

template <int N>
class StringLiteral {
 public:
    constexpr StringLiteral(char const (&str)[N]) {
        std::copy(str, str + N, data);
    }
    char data[N];
};

template <StringLiteral str>
class ClassTemplate {};

template <StringLiteral str>
void FunctionTemplate() {
    std::cout << str.data << '\n';
}

int main() {

    std::cout << '\n';

    ClassTemplate<"string literal"> cls;
    FunctionTemplate<"string literal">();

    std::cout << '\n';

}
```

`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[59]. There is already a proposal for C++23 in the pipeline: P1433R0: Compile-Time Regular Expressions[60]. 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[61]] 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'.*

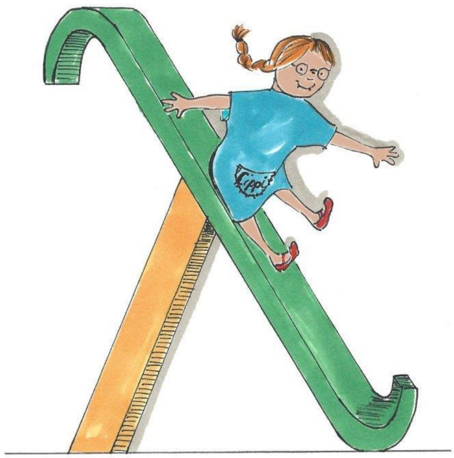
If the RE [regular expression] is known at compile time, the pattern should be checked during the compilation. The design of `std::regex` doesn't allow for this[compile-time evaluation,] as the RE input is a run-time string and syntax errors are reported as exceptions.".

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-points as non-type template parameters.

[59] https://github.com/hanickadot/compile-time-regular-expressions
[60] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1433r0.pdf
[61] 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. Additionally, lambda expressions can be used in unevaluated contexts. With C++20, they detect when you implicitly copy the `this` pointer. This 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

```cpp
// 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 the use 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>& v\
ec) {
    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(vecDoubl\
e) << '\n';
```

```
36      std::cout << "lambdaVector(vecDouble): " << lambdaVector(vecDouble)\
37   << '\n';
38      // std::cout << "lambdaVectorIntegral(vecDouble): "
39      //           << lambdaVectorIntegral(vecDouble) << '\n';
40
41      std::cout << '\n';
42
43      std::cout << "lambdaGeneric(vecInt): " << lambdaGeneric(vecInt) << \
44  '\n';
45      std::cout << "lambdaVector(vecInt): " << lambdaVector(vecInt) << '\\
46  n';
47      std::cout << "lambdaVectorIntegral(vecInt): "
48                << lambdaVectorIntegral(vecInt) << '\n';
49
50      std::cout << '\n';
51
52  }
```

Function `lambdaGeneric` (line 9) can be invoked with any data type that has 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

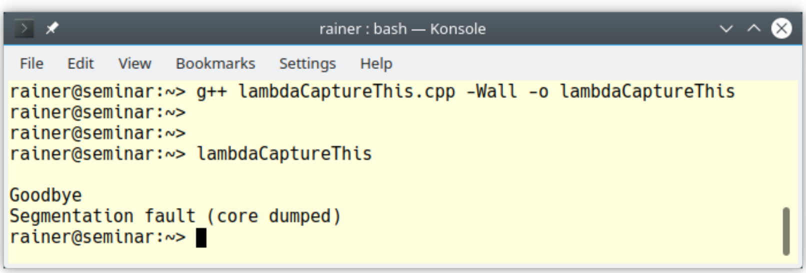
The C++20 compiler detects when you implicitly copy the `this` pointer. Implicitly capturing the `this` pointer by copy can cause undefined behavior. Undefined behavior essentially means that there are no guarantees for the behavior of the program, such as for the following:

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();
```

```
20 }
21
22
23 int main() {
24
25     std::cout << '\n';
26
27     auto lam = makeLambda();
28     lam();
29
30     std::cout << '\n';
31
32 }
```

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
```

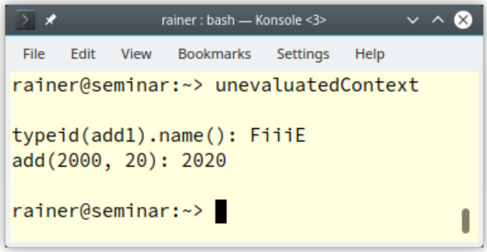
Function `add1` is declared, while `add2` is defined. This means, if you use `add1` in an evaluated context, for example, by invoking it, you get a link-time error. The key observation is that you can use `add1` in unevaluated contexts, such as `typeid`[62] or `decltype`[63]. Both operators accept unevaluated operands.

[62]https://en.cppreference.com/w/cpp/language/typeid
[63]https://en.cppreference.com/w/cpp/language/decltype

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
16     decltype(*add1) add = add2;                                        \
17
18
19     std::cout << "add(2000, 20): " << add(2000, 20) << '\n';
20
21     std::cout << '\n';
22
23 }
```

`typeid(add1).name()` (line 13) returns a string representation of the type and `decltype` (line 15) deduces the type of its argument.



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 have to 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`) by default use the function object `std::less` to sort the keys. `std::less` sorts all keys lexicographically in ascending order. The declaration of `std::set`[64] 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.

[64] https://en.cppreference.com/w/cpp/container/set

Lambdas used in an unevaluated context

```
// 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", "m\
ichael"};
    printContainer(set1);

    using SetDecreasing = std::set<std::string,
                                   decltype([](const auto& l, const aut\
o& 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();
                               })>;
```

```
36     SetLength set3 = {"scott", "Bjarne", "Herb", "Dave", "michael"};
37     printContainer(set3);
38 
39     std::cout << '\n';
40 
41     std::set<int> set4 = {-10, 5, 3, 100, 0, -25};
42     printContainer(set4);
43 
44     using setAbsolute = std::set<int, decltype([](const auto& l, const \
45 auto& r) {
46                                                   return std::abs(l)< \
47 std::abs(r);
48                                               })>;
49     setAbsolute set5 = {-10, 5, 3, 100, 0, -25};
50     printContainer(set5);
51 
52     std::cout << "\n\n";
53 
54 }
```

`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 unique 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
```

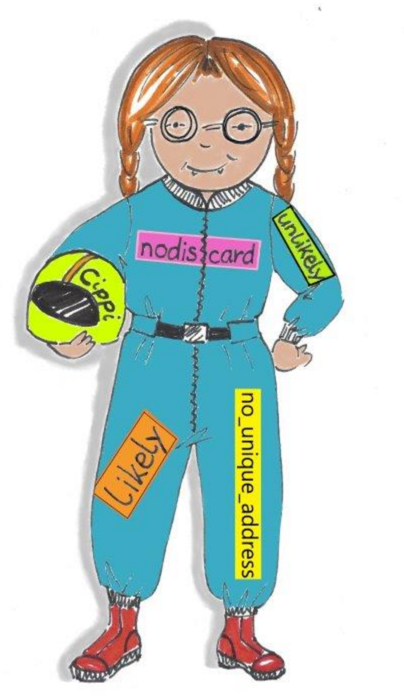
Use of a lambda in an unevaluated context

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`, that was added first. `std::set` supports unique keys, and the keys are in this case identical using the special predicate. If I had used `std::multiset`, this wouldn't have happened.

Distilled Information

- With C++20, lambdas can have template parameters. In addition, lambdas detect when the `this` pointer is implicitly referenced.

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

```
[[attribute1]]  [[attribute2]] [[attribute3]]
int func1();

[[attribute1, attribute2, attribute3]]
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[65]
- `[[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.

[65] https://en.cppreference.com/w/cpp/atomic/memory_order#Release-Consume_ordering

Discarding objects and error codes

```cpp
// withoutNodiscard.cpp

#include <utility>

struct MyType {

     MyType(int, bool) {}

};

template <typename T, typename ... Args>
T* create(Args&& ... args) {
  return new T(std::forward<Args>(args)...);
}

enum class ErrorCode {
    Okay,
    Warning,
    Critical,
    Fatal
};

ErrorCode errorProneFunction() { return ErrorCode::Fatal; }

int main() {

    int* val = create<int>(5);
    delete val;

    create<int>(5);

    errorProneFunction();

    MyType(5, true);

```

```
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. This 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`.

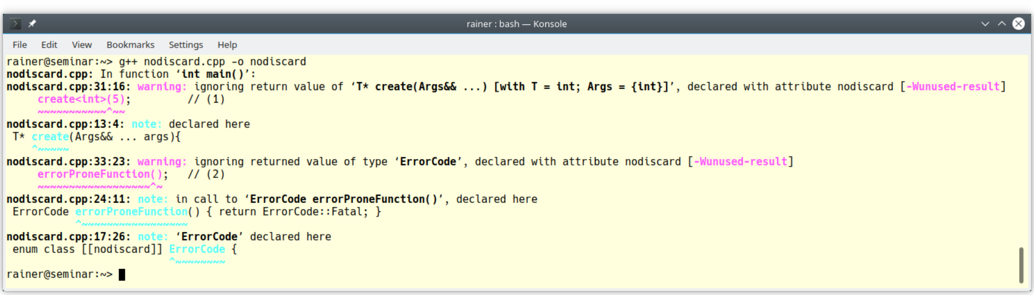
Let us see what this means. In the following example, I use the C++17 syntax of the attribute `[[nodiscard]]`.

Use of the attribute [[nodiscard]] in C++17

```
// nodiscard.cpp

#include <utility>

struct MyType {

     MyType(int, bool) {}

};

template <typename T, typename ... Args>
[[nodiscard]]
T* create(Args&& ... args){
  return new T(std::forward<Args>(args)...);
}

```

```
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.

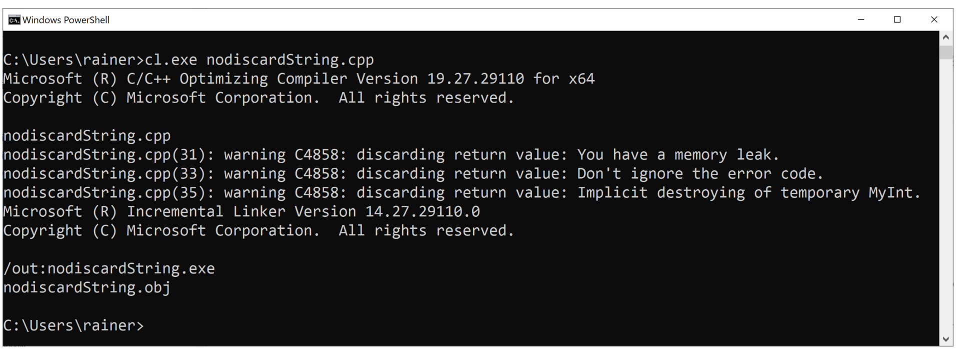
Both issues can be solved with C++20. Constructors can be declared as `[[nodiscard]]`, and the warning can have additional information.

Use of the attribute [[nodiscard]] in C++20

```cpp
// nodiscardString.cpp

#include <utility>

struct MyType {

     [[nodiscard("Implicit destroying of temporary MyInt.")]] MyType(in\
t, bool) {}

};

template <typename T, typename ... Args>
[[nodiscard("You have a memory leak.")]]
T* create(Args&& ... args){
  return new T(std::forward<Args>(args)...);
}

enum class [[nodiscard("Don't ignore the error code.")]] ErrorCode {
    Okay,
    Warning,
    Critical,
    Fatal
};

ErrorCode errorProneFunction() { return ErrorCode::Fatal; }

int main() {

```

```
    int* val = create<int>(5);
    delete val;

    create<int>(5);

    errorProneFunction();

    MyType(5, true);

}
```

Now, the user of the functions gets specific messages. Here is the output of the Microsoft compiler.


```
C:\Users\rainer>cl.exe nodiscardString.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.27.29110 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

nodiscardString.cpp
nodiscardString.cpp(31): warning C4858: discarding return value: You have a memory leak.
nodiscardString.cpp(33): warning C4858: discarding return value: Don't ignore the error code.
nodiscardString.cpp(35): warning C4858: discarding return value: Implicit destroying of temporary MyInt.
Microsoft (R) Incremental Linker Version 14.27.29110.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:nodiscardString.exe
nodiscardString.obj

C:\Users\rainer>
```

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"[66].

While studying the `[[nodiscard]]` syntax on cppreference.com/nodiscard[67], I noticed that the declarations of `std::async`[68] 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[69] for the attributes `[[likely]]` and `[[unlikely]]` is the shortest proposal I know of. To give you an idea, this is the 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*

[66] https://www.modernescpp.com/index.php/the-special-futures
[67] https://en.cppreference.com/w/cpp/language/attributes/nodiscard
[68] https://en.cppreference.com/w/cpp/thread/async
[69] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0479r5.html

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.

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]]

```
// uniqueAddress.cpp

#include <iostream>

struct Empty {};

struct NoUniqueAddress {
    int d{};
    [[no_unique_address]] Empty e{};
```

```
};

struct UniqueAddress {
    int d{};
    Empty e{};
};

int main() {

    std::cout << '\n';

    std::cout << std::boolalpha;

    std::cout << "sizeof(int) == sizeof(NoUniqueAddress): "            \

              << (sizeof(int) == sizeof(NoUniqueAddress)) << '\n';

    std::cout << "sizeof(int) == sizeof(UniqueAddress): "              \

              << (sizeof(int) == sizeof(UniqueAddress)) << '\n';

    std::cout << '\n';

    NoUniqueAddress NoUnique;

    std::cout << "&NoUnique.d: " <<  &NoUnique.d << '\n';
    std::cout << "&NoUnique.e: " <<  &NoUnique.e << '\n';

    std::cout << '\n';

    UniqueAddress unique;

    std::cout << "&unique.d: " <<  &unique.d << '\n';
    std::cout << "&unique.e: " <<  &unique.e << '\n';

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

```
46
47 }
```

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).

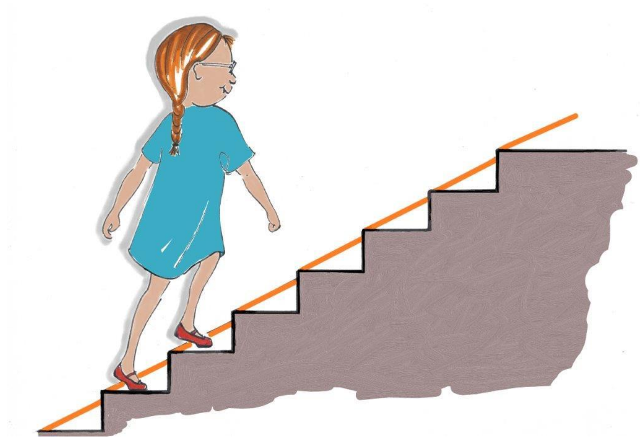
```
sizeof(int) == sizeof(NoUniqueAddress): true
sizeof(int) == sizeof(UniqueAddress): false

&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[70] 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.*" This 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.

[70] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1152r0.html

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

If you want to know all the sophisticated details, I strongly suggest you watch the CppCon 2019 talk "Deprecating volatile"[71] 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
```

[71] https://www.youtube.com/watch?v=KJW_DLaVXIY

```
//              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
    // ...
}
```

`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

```
1  // rangeBasedForLoopInitializer.cpp
2  
3  #include <iostream>
4  #include <string>
5  #include <vector>
6  
7  int main() {
8  
9      for (auto vec = std::vector{1, 2, 3}; auto v : vec) {
10         std::cout << v << " ";
11     }
12 
13     std::cout << "\n\n";
14 
15     for (auto initList = {1, 2, 3}; auto e : initList) {
16         e *= e;
17         std::cout << e << " ";
18     }
19 
20     std::cout << "\n\n";
21 
22     using namespace std::string_literals;
23     for (auto str = "Hello World"s; auto c: str) {
24         std::cout << c << " ";
25     }
26 
27     std::cout << '\n';
28 
29 }
```

The range-based for loop uses in line 9 a `std::vector`, in line 15 a `std::initializer_list`, and in line 23 a `std::string`. Furthermore, in line 9 and line 15 I apply automatic type deduction for class templates, which we have 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

```cpp
// virtualConstexpr.cpp

#include <iostream>

struct X1 {
    virtual int f() const = 0;
};

struct X2: public X1 {
    constexpr int f() const override { return 2; }
};

struct X3: public X2 {
    int f() const override { return 3; }
};

struct X4: public X3 {
    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 `char`s.

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

```
1 std::u8string std::basic_string<char8_t>
2 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

4.9.5 `using enum` in Local Scopes

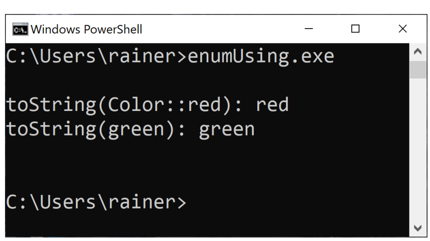
A `using enum` declaration introduces the enumerators of the named enumeration in the local scope.

Introducing enumerators in the local scope

```
// enumUsing.cpp

#include <iostream>
#include <string_view>

enum class Color {
    red,
    green,
    blue
};

std::string_view toString(Color col) {
  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) << '\\
27 n';
28
29     using enum Color;                                                \
30
31
32     std::cout << "toString(green): " << toString(green) << '\n';
33
34     std::cout << '\n';
35
36 }
```

The `using enum` declaration (line 14) 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).



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[72]: "*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 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:

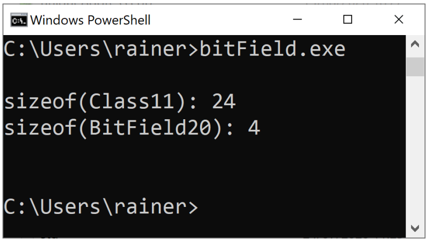
Default initializers for the members of a bit field

```
// bitField.cpp

#include <iostream>

struct Class11 {
    int i = 1;
    int j = 2;
    int k = 3;
    int l = 4;
    int m = 5;
    int n = 6;
};

struct BitField20 {
    int i : 3 = 1;
    int j : 4 = 2;
    int k : 5 = 3;
    int l : 6 = 4;
    int m : 7 = 5;
    int n : 7 = 6;
};

int main () {

```

[72] https://en.wikipedia.org/wiki/Bit_field

```
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 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`:

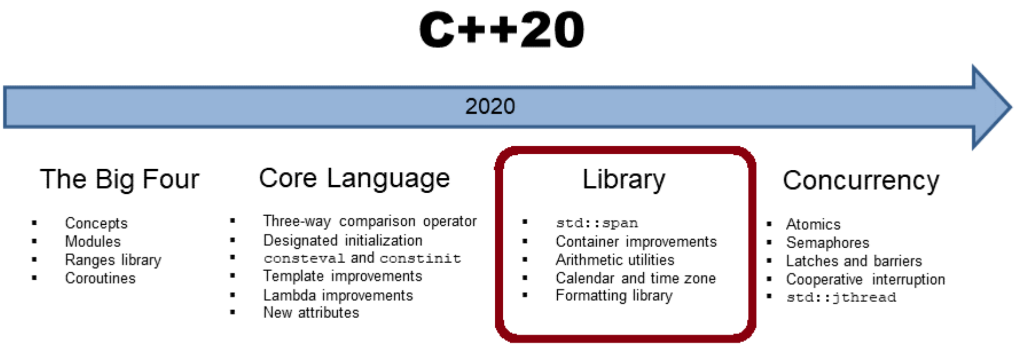


Size information to a bit field

Distilled Information

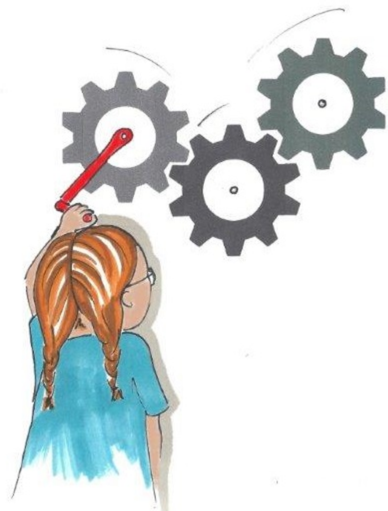
- The meaning of `volatile` is clarified in C++20. `volatile` has no multithreading semantics and should only be used to avoid aggressive optimization because an object may be changed independently of the regular program flow.
- Range-based for loops can use an initializer.
- The new character type `char8_t` 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, such as 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 can easily be composed. To make it short: The comfort and the power of the ranges library is 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 The Concepts Ranges and Views

I already presented the concepts ranges and views in the chapter on concepts. Consequently, here's a brief refresher.

- `range`: A range is 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.

A `view` is something that you apply on a range and performs some operation. A view does not own data, and its time complexity to copy, move, or assign is constant.

Views 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 `std::views::filter` and `std::views::transform` are the views.

Thanks to views, C++20 allows programming in a functional style. Views can be combined and are lazy. I already presented two views, but C++20 offers more.

Views 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::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.

Views in C++20

View	Description
`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.
`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::basic_istream_view` `std::ranges::istream_view`	Applies `operator>>` on the input stream.
`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.

In general, you can use a view such as `std::views::transform` with the alternative name `std::ranges:: transform_view`.

5.1.2 Direct on the Container

The algorithms of the Standard Template Library (STL) are sometimes a little inconvenient. They need both begin and end iterators. 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

}
```

Those algorithms of the algorithm library[1], which are included in the `<algorithm>`[2] header such as `std::sort` have a ranges pendant `std::ranges::sort`.

When you study the overloads of `std::ranges::sort`, you notice that they support a projection.

5.1.2.1 Projection

`std::ranges::sort` has two overloads:

Overload of ‘std::ranges::sort

```
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 = {}, P\
roj 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. A projection is a mapping of a set into a subset. Let me show you what that means:

[1] https://en.cppreference.com/w/cpp/algorithm
[2] https://en.cppreference.com/w/cpp/header/algorithm

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", 44\
4},
    {"Grimm", 666}, {"Butcher", 222}, {"Taylor", 555}, {"Wilson", 333} \
};

    std::ranges::sort(phoneBook, {}, &PhoneBookEntry::name);    // ascen\
ding by name
    printPhoneBook(phoneBook);

    std::ranges::sort(phoneBook, std::ranges::greater() ,
                      &PhoneBookEntry::name);                  // desce\
```

```
nding by name
    printPhoneBook(phoneBook);

    std::ranges::sort(phoneBook, {}, &PhoneBookEntry::number); // ascen\
ding by number
    printPhoneBook(phoneBook);

     std::ranges::sort(phoneBook, std::ranges::greater(),
                       &PhoneBookEntry::number);              // descen\
ding by number
    printPhoneBook(phoneBook);

    std::cout << '\n';

}
```

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).

```
(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

Most ranges algorithms support projections.

5.1.2.2 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`

```
// rangesEntireContainer.cpp

#include <iostream>
#include <ranges>
#include <string>
#include <unordered_map>


int main() {

  std::unordered_map<std::string, int> freqWord{ {"witch", 25}, {"wizar\
d", 33},
                                                 {"tale", 45}, {"dog", \
4},
                                                 {"cat", 34}, {"fish", \
23} };

  std::cout << "Keys:" << '\n';
  auto names = std::views::keys(freqWord);
  for (const auto& name : names){ std::cout << name << " "; }
  std::cout << '\n';
  for (const auto& name : std::views::keys(freqWord)){ std::cout << nam\
e << " "; }

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

  std::cout << "Values: " << '\n';
  auto values = std::views::values(freqWord);
  for (const auto& value : values){ std::cout << value << " "; }
  std::cout << '\n';
  for (const auto& value : std::views::values(freqWord)) {
                              std::cout << value << " ";
                            }

}
```

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.

5.1.3 Function Composition

In the example `rangesComposition.cpp`, I use a `std::map`, because the ordering of the keys is crucial.

Composition of views

```
// rangesComposition.cpp

#include <iostream>
#include <ranges>
#include <string>
#include <map>


int main() {

  std::map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33},
                                       {"tale", 45}, {"dog", 4},
                                       {"cat", 34}, {"fish", 23} };
```

```
14
15   std::cout << "All words: ";
16   for (const auto& name : std::views::keys(freqWord)) { std::cout << na\
17 me << " "; }
18
19   std::cout << '\n';
20
21   std::cout << "All words, reverses: ";
22   for (const auto& name : std::views::keys(freqWord)
23                          | std::views::reverse) { std::cout << name << "\
24  "; }
25
26   std::cout << '\n';
27
28   std::cout << "The first 4 words: ";
29   for (const auto& name : std::views::keys(freqWord)
30                          | std::views::take(4)) { std::cout << name << "\
31  "; }
32
33   std::cout << '\n';
34
35   std::cout << "All words starting with w: ";
36   auto firstw = [](const std::string& name){ return name[0] == 'w'; };
37   for (const auto& name : std::views::keys(freqWord)
38                          | std::views::filter(firstw)) { std::cout << na\
39 me << " "; }
40
41   std::cout << '\n';
42
43 }
```

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[3] 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;
```

5.1.4 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

```
// rangesIota.cpp

#include <iostream>
#include <numeric>
#include <ranges>
#include <vector>

int main() {

    std::cout << std::boolalpha;

    std::vector<int> vec;
    std::vector<int> vec2;
```

[3] https://en.wikipedia.org/wiki/Syntactic_sugar

```
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_b\
18  ack(i);
19
20      std::cout << "vec == vec2: " << (vec == vec2) << '\n';
21
22      for (int i: vec) std::cout << i << " ";
23
24  }
```

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

```
// rangesLazy.cpp

#include <iostream>
#include <ranges>


bool isPrime(int i) {
    for (int j=2; j*j <= i; ++j){
        if (i % j == 0) return false;
    }
```

```
    return true;
}

int main() {

    std::cout << "Numbers from 1'000'000 to 1'001'000 (displayed each 1\
00th): "
              << '\n';
    for (int i: std::views::iota(1'000'000, 1'001'000)) {
        if (i % 100 == 0) std::cout << i << " ";
    }

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

    auto odd = [](int i){ return i % 2 == 1; };
    std::cout << "Odd numbers from 1'000'000 to 1'001'000 (displayed ea\
ch 100th): "
              << '\n';
    for (int i: std::views::iota(1'000'000, 1'001'000) | std::views::fi\
lter(odd)) {
         if (i % 100 == 1) std::cout << i << " ";
    }

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

    std::cout << "Prime numbers from 1'000'000 to 1'001'000: " << '\n';
    for (int i: std::views::iota(1'000'000, 1'001'000) | std::views::fi\
lter(odd)
                                                      | std::views::filte\
r(isPrime)) {
        std::cout << i << " ";
    }

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

    std::cout << "20 prime numbers starting with 1'000'000: " << '\n';
```

```
47      for (int i: std::views::iota(1'000'000) | std::views::filter(odd)
48                                                | std::views::filter(isPrim\
49  e)
50                                                | std::views::take(20)) {
51          std::cout << i << " ";
52      }
53
54      std::cout << '\n';
55
56  }
```

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

5.1.5 Define a View

You can define your own view.

5.1.5.1 `std::ranges::view_interface`

Thanks to the `std::ranges::view_interface`[4] helper class, defining a view is easy. To fulfil the concept view, your view needs at least a default constructor, and member functions `begin()` and `end()`:

[4] https://en.cppreference.com/w/cpp/ranges/view_interface

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[5] (short CRTP).

I use this technique in the next example to create a view out of a container of the Standard Template Library.

5.1.5.2 A Container View

The view `ContainerView` creates a view on an arbitrary container.

Creating a view from a container

```
// containerView.cpp

#include <iostream>
#include <ranges>
#include <string>
#include <vector>

template<std::ranges::input_range Range>
requires std::ranges::view<Range>
class ContainerView : public std::ranges::view_interface<ContainerView<\
Range>> {
private:
   Range range_{};
   std::ranges::iterator_t<Range> begin_{ std::begin(range_) };
```

[5] https://www.modernescpp.com/index.php/c-is-still-lazy

```
    std::ranges::iterator_t<Range> end_{ std::end(range_) };

public:
    ContainerView() = default;

    constexpr ContainerView(Range r): range_(std::move(r)) ,
                                      begin_(std::begin(r)), end_(std::en\
d(r)) {}

    constexpr auto begin() const {
        return begin_;
    }
    constexpr auto end() const {
        return end_;
    }
};

template<typename Range>
ContainerView(Range&& range) -> ContainerView<std::ranges::views::all_t\
<Range>>;

int main() {

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

    auto myContainerView = ContainerView(myVec);
    for (auto c : myContainerView) std::cout << c << " ";
    std::cout << '\n';

    for (auto i : std::views::reverse(ContainerView(myVec))) std::cout  \
<< i <<  ' ';
    std::cout << '\n';

    for (auto i : ContainerView(myVec) | std::views::reverse) std::cout \
 << i <<  ' ';
    std::cout << '\n';
```

```
51
52    std::cout << std::endl;
53
54    std::string myStr = "Only for testing purpose.";
55
56    auto myContainerView2 = ContainerView(myStr);
57    for (auto c: myContainerView2) std::cout << c << " ";
58    std::cout << '\n';
59
60    for (auto i : std::views::reverse(ContainerView(myStr))) std::cout <\
61 < i << ' ';
62    std::cout << '\n';
63
64    for (auto i : ContainerView(myStr) | std::views::reverse) std::cout \
65  << i <<  ' ';
66    std::cout << '\n';
67
68 }
```

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), and the two required member functions `begin()` (line 22) and `end()` (line 25). 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[6] provides more information to the user-defined deduction guide for class templates.

In the next section on the ranges library, I want to perform a small experiment. Can

[6] https://en.cppreference.com/w/cpp/language/class_template_argument_deduction

I add a flavor of Python into C++?

5.1.6 A Flavor of Python

The programming language Python[7] 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

Here is my challenge: I want to implement Python-like functions `filter`, `map`, and list comprehension in C++20 using the ranges library.

5.1.6.1 `filter`

Python's `filter` function can be directly mapped to the corresponding ranges function.

[7] https://www.python.org/

Python's filter function in C++

```
// 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", "purpo\
ses"};
    auto res2 = filter([](const std::string& s){ return std::isupper(s[\
0]); },
                          myStrings);
```

```
36
37     std::cout << "\n\n";
38
39     for (auto word: res2) std::cout << word << '\n';
40
41     std::cout << '\n';
42
43 }
```

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 just 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.

5.1.6.2 `map`

`map` applies a callable to each element of the input sequence.

Python's map function in C++

```
// mapRanges.cpp

#include <iostream>
#include <list>
#include <ranges>
#include <string>
#include <vector>
#include <utility>


template <typename Func, typename Seq>
auto map(Func func, const Seq& seq) {

    typedef typename Seq::value_type value_type;
    using return_type = decltype(func(std::declval<value_type>()));

    std::vector<return_type> result{};
    for (auto i :seq | std::views::transform(func)) result.push_back(i);

    return result;
}

int main() {

    std::cout << '\n';

    std::list<int> myInts{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto res = map([](int i){ return i * i; }, myInts);

    for (auto v: res) std::cout << v << " ";

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

    std::vector<std::string> myStrings{"Only", "for", "testing", "purpo\
ses"};
```

```
36     auto res2 = map([](const std::string& s){ return std::make_pair(s.s\
37 ize(), s); },
38                                                                     myS\
39 trings);
40 
41     for (auto p: res2) std::cout << "(" <<  p.first << ", " << p.second\
42  << ") " ;
43 
44     std::cout << "\n\n";
45                                                                        \
46 
47 }
```

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 elements of the input sequence 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. This 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

5.1.6.3 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++

```
// 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::val\
ue_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; });
    for (auto v: res) std::cout << v << " ";

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

    std::vector<std::string> myStrings{"Only", "for", "testing", "purpo\
ses"};
    auto res2 = mapFilter([](const std::string& s){
                              return std::make_pair(s.size(), s);
                          }, myStrings);
    for (auto p: res2) std::cout << "(" <<  p.first << ", " << p.second\
 << ") " ;

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

    myStrings = {"Only", "for", "testing", "purposes"};
    res2 = mapFilter([](const std::string& s){
                         return std::make_pair(s.size(), s);
                     }, myStrings,
                     [](const std::string& word){ return std::isupper(w\
ord[0]); });

    for (auto p: res2) std::cout << "(" <<  p.first << ", " << p.second\
 << ") " ;

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

```

```
72 }
```

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. Line 42 and 55 apply both functions `map` and `filter` in one call.

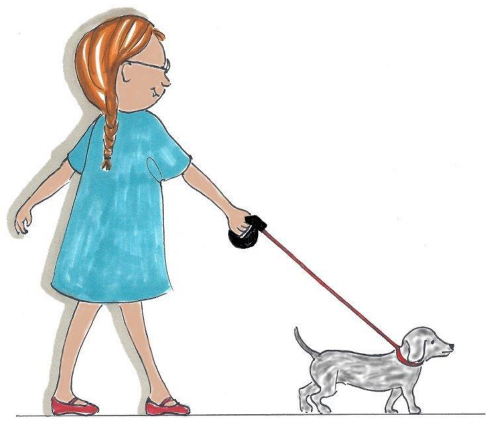
```
1 4 9 16 25 36 49 64 81 100

1 9 25 49 81

(4, Only) (3, for) (7, testing) (8, purposes)

(4, Only)
```

Both functions `map` and `filter` applied

Distilled Information

- The ranges library provides us with an additional version of the STL algorithms. The ranges library algorithms are lazy, can work directly on containers and can be composed.
- The algorithm of the ranges library
  - are lazy and can, therefore, be invoked on infinite data streams.
  - can operate directly on the container and don't need a range defined by two iterators.
  - can be composed using the pipe (`|`) symbol.

5.2 `std::span`



Cippi walks the dog

A `std::span` stands for 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 type: `std::span<T>`.

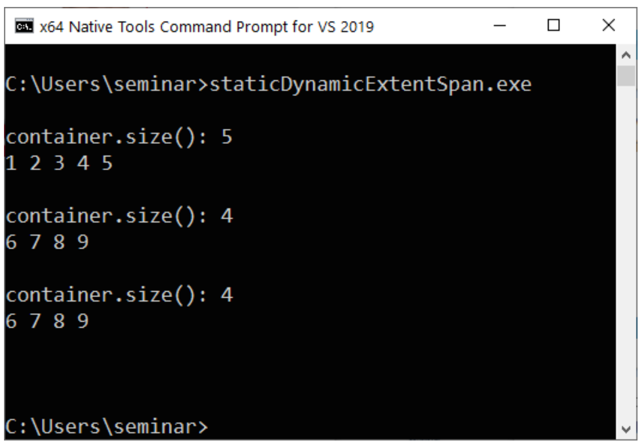
The next example `staticDynamicExtentSpan.cpp` emphasizes the differences between both kinds of views.

`std::span`s with static and dynamic extent

```cpp
// staticDynamicExtentSpan.cpp

#include <iostream>
#include <span>
#include <vector>

void printMe(std::span<int> container) {

    std::cout << "container.size(): " << container.size() << '\n';
    for (auto e : container) std::cout << e << ' ';
    std::cout << "\n\n";
}

int main() {

    std::cout << '\n';

    std::vector myVec1{1, 2, 3, 4, 5};
    std::vector myVec2{6, 7, 8, 9};

    std::span<int> dynamicSpan(myVec1);
    std::span<int, 4> staticSpan(myVec2);

    printMe(dynamicSpan);
    printMe(staticSpan);  // implicitly converted into a dynamic span

    // staticSpan = dynamicSpan; ERROR
    dynamicSpan = staticSpan;

```

```
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 the other way around. Line 27 would cause an error, but lines 7, 25, and 28 are valid.



`std::spans` with static and dynamic extent

One important reason for having a `std::span<T>` is that a plain C-array decays[8] to a pointer if passed to a function; therefore, the size is lost. This decay is a typical reason for errors in C/C++.

5.2.2 Automatically Deduces the Size of a Contiguous Sequence of Objects

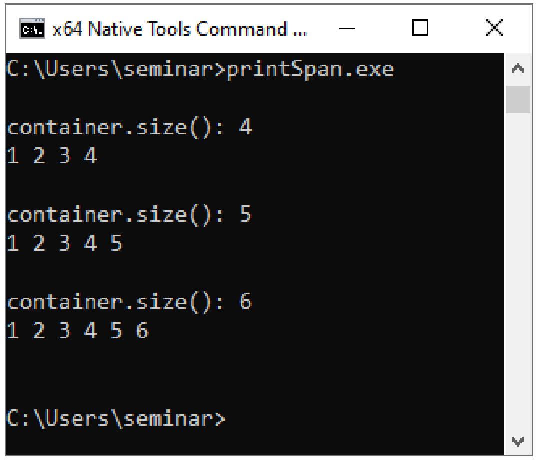
In contrast to a C-array, `std::span<T>` automatically deduces the size of contiguous sequences of objects.

[8]https://en.cppreference.com/w/cpp/types/decay

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 
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

There are more ways to create a `std::span`.

5.2.3 Create a `std::span` from a Pointer and a Size

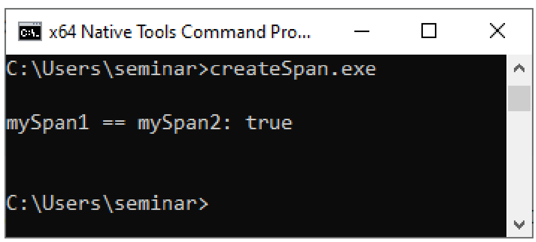
You can create a `std::span` from a pointer and a size.

Create a std::span

```
// createSpan.cpp

#include <algorithm>
#include <iostream>
#include <span>
#include <vector>

int main() {

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

    std::vector myVec{1, 2, 3, 4, 5};

    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

A `std::span` is neither a `std::string_view` nor a view

You may remember that a `std::span` is sometimes called a view. Don't confuse a `std::span` with a view from the ranges library or a std::string_view[9].

A view from the ranges library is something that you can apply on a range and performs some operation. A view does not own data, and its time for each copy, move, and assignment is constant.

A `std::span` and a `std::string_view` 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.

[9] https://www.modernescpp.com/index.php/c-17-what-s-new-in-the-library

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.

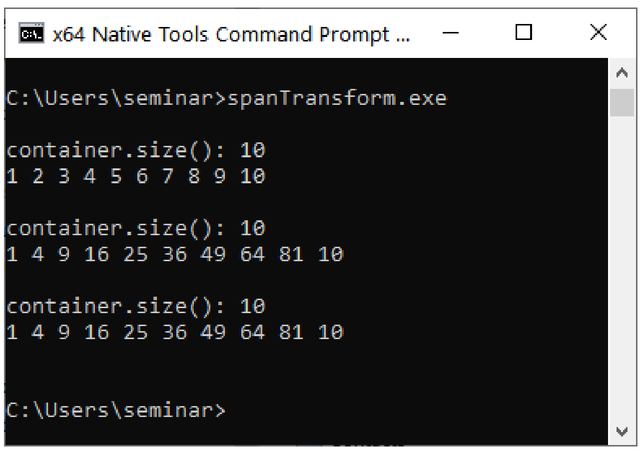
The following program shows how a subspan can be used to modify the referenced objects from a `std::vector`.

Modify the objects referenced by a `std::span`

```
// spanTransform.cpp

#include <algorithm>
#include <iostream>
#include <vector>
#include <span>

void printMe(std::span<int> container) {

    std::cout << "container.size(): " << container.size() << '\n';
    for (auto e : container) std::cout << e << ' ';
    std::cout << "\n\n";
}

int main() {

    std::cout << '\n';

    std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    printMe(vec);

    std::span span1(vec);
    std::span span2{span1.subspan(1, span1.size() - 2)};


    std::transform(span2.begin(), span2.end(),
                   span2.begin(),
                   [](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 elements of the underlying `vec` excluding the first and the last element (line 23). Consequently, the mapping of each element to its square (line 26) only addresses these elements.



Modify the objects referend by a `std::span`

There are various convenience functions to address the elements of the `std::span`.

5.2.5 Addressing `std::span` Elements

The following table presents the functions to refer to the elements of a `std::span`.

Interface of a std::span sp

Function	Description
sp.front()	Access the first element.
sp.back()	Access the last element.
sp[i]	Access the i-th element.
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 size of the sequence in bytes.
sp.empty()	Returns true if the sequence is empty.
sp.first<count>() sp.first(count)	Returns a subspan consisting of the first count elements of the sequence.
sp.last<count>() sp.last(count)	Returns a subspan consisting of the last count elements of the sequence.
sp.subspan<first, count>() sp.subspan(first, count)	Returns a subspan consisting of count elements starting at first.

The program subspan.cpp shows the usage of the member function subspan.

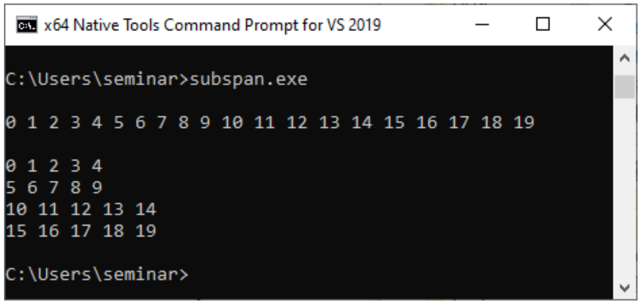
Use of the member function `subspan`

```
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 += cou\
23 nt ) {
24         for (auto ele: mySpan.subspan(first, count)) std::cout << ele <\
25 < " ";
26         std::cout << '\n';
27     }
28 
29 }
```

Line 13 fills the vector with all numbers from 0 to 19 (line 13) using the algorithm `std::iota`[10]. This vector is further used to initialize a `std::span` (line 18). Finally, the

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

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 special use case of `std::span`. 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` a 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`, the range models a constant range of constant objects: `const std::vector<T>`. You cannot model a constant range of modifiable elements. Here comes `std::span` 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>`

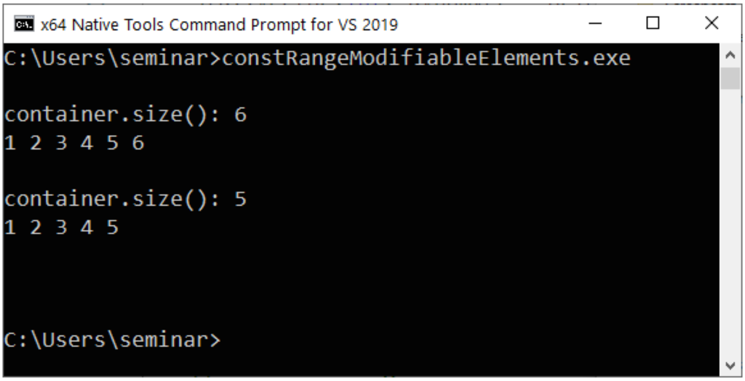
The program `constRangeModifiableElements.cpp` exemplifies each combination.

(Constant/modifiable) ranges of (constant/modifiable) elements

```cpp
// constRangeModifiableElements.cpp

#include <iostream>
#include <span>
#include <vector>

void printMe(std::span<int> container) {

    std::cout << "container.size(): " << container.size() << '\n';
    for (auto e : container) std::cout << e << ' ';
    std::cout << "\n\n";
}

int main() {

    std::cout << '\n';

    std::vector<int> origVec{1, 2, 2, 4, 5};

    // Modifiable range of modifiable elements
    std::vector<int> dynamVec = origVec;
    dynamVec[2] = 3;
    dynamVec.push_back(6);
    printMe(dynamVec);

    // Constant range of constant elements
    const std::vector<int> constVec = origVec;
    // constVec[2] = 3;         ERROR
    // constVec.push_back(6);   ERROR
    std::span<const int> constSpan(origVec);
    // constSpan[2] = 3;        ERROR

    // Constant range of modifiable elements
    std::span<int> dynamSpan{origVec};
    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 an element 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

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`.
- 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.

5.3 Container Improvements

Cippi inspects the container

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

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[11] of the Standard Template Library are declared as `constexpr`.

Consequently, you can sort a `std::vector` of ints at compile time.

[11]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

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[12].

Create a `std::array` from various one-dimensional arrays

```
// toArray.cpp

#include <iostream>
#include <utility>
#include <array>
#include <memory>

int main() {

    std::cout << '\n';

    auto arr1 = std::to_array("A simple test");
```

[12] https://en.cppreference.com/w/cpp/container/array/to_array

```
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() <\
24  < '\n';
25      std::cout << '\n';
26
27      auto arr4 = std::to_array<std::pair<int, double>>({ {1, 0.0}, {2, 5\
28  .1},
29                                                          {3, 5.1} });
30      for (auto p: arr4) {
31          std::cout << "(" << p.first << ", " << p.second << ")" << '\n';
32      }
33
34      std::cout << "\n\n";
35
36  }
```

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).

```
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`[13]. 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

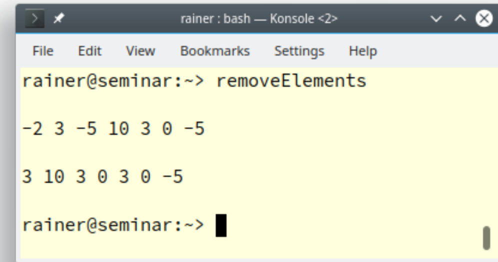
Removing an element from a container seems to be quite easy. In the case of a `std::vector`, you can use the function `std::remove_if`.

[13] https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared

Using `std::remove_if` to remove elements from a container

```
// removeElements.cpp

#include <algorithm>
#include <iostream>
#include <vector>

int main() {

    std::cout << '\n';

    std::vector myVec{-2, 3, -5, 10, 3, 0, -5 };

    for (auto ele: myVec) std::cout << ele << " ";
    std::cout << "\n\n";

    std::remove_if(myVec.begin(), myVec.end(), [](int ele){ return ele \
< 0; });
    for (auto ele: myVec) std::cout << ele << " ";

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

}
```

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.

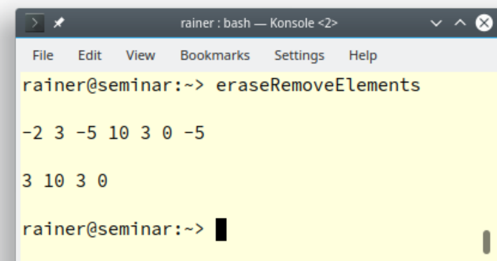
To modify a container, you have to apply the new logical end to the container.

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.

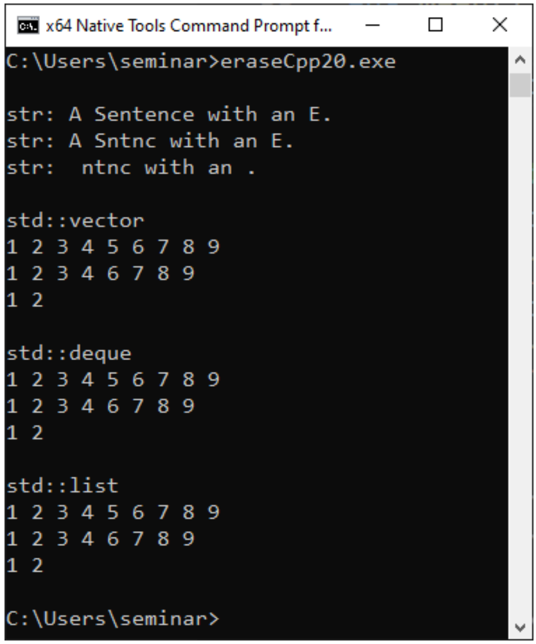
Erase elements from a container

```
// eraseCpp20.cpp

#include <iostream>
#include <numeric>
#include <deque>
#include <list>
#include <string>
#include <vector>

template <typename Cont>
void eraseVal(Cont& cont, int val) {
    std::erase(cont, val);
}

template <typename Cont, typename Pred>
void erasePredicate(Cont& cont, Pred pred) {
    std::erase_if(cont, pred);
}

template <typename Cont>
void printContainer(Cont& cont) {
    for (auto c: cont) std::cout << c << " ";
    std::cout << '\n';
}

template <typename Cont>
void doAll(Cont& cont) {
    printContainer(cont);
    eraseVal(cont, 5);
    printContainer(cont);
    erasePredicate(cont, [](auto i) { return i >= 3; } );
    printContainer(cont);
}

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 and erases all the upper case 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

```cpp
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.

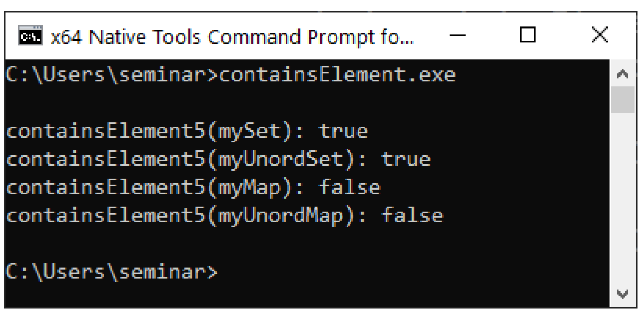
Unlike `find` and `count`, the `contains` member function in C++20 is quite convenient to use.

`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(m\
yUnordSet);

    std::cout << '\n';

```

```
    std::map<int, std::string> myMap{ {1, "red"}, {2, "blue"}, {3, "gre\
en"} };
    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(m\
yUnordMap);

    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 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

```cpp
// stringStartsWithEndsWith.cpp

#include <iostream>
#include <string_view>
#include <string>

template <typename PrefixType>
void startsWith(const std::string& str, PrefixType prefix) {
    std::cout << "            starts with " << prefix << ": "
              << str.starts_with(prefix) << '\n';
}

template <typename SuffixType>
void endsWith(const std::string& str, SuffixType suffix) {
    std::cout << "            ends with " << suffix << ": "
              << str.ends_with(suffix) << '\n';
}

int main() {

    std::cout << '\n';

    std::cout << std::boolalpha;

    std::string helloWorld("Hello World");

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

    startsWith(helloWorld, helloWorld);

    startsWith(helloWorld, std::string_view("Hello"));

    startsWith(helloWorld, 'H');

    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

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.
- `std::string` supports the new member function `start_with` and `end_with` to check if the container has a specific prefix or suffix.

5.4 Arithmetic Utilities

Cippi studies arithmetic

The comparison of signed and unsigned integers is a subtle cause for unexpected behavior and, therefore, of 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. To motivate 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.

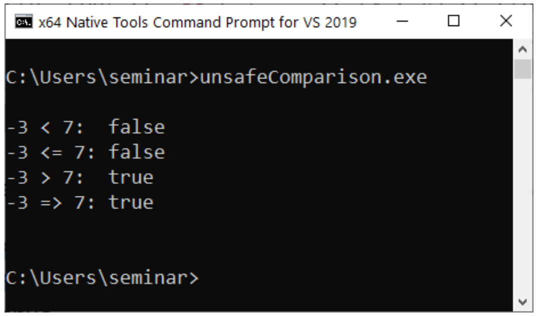
5.4.1.1 Unsafe Comparison

Of course, there is a reason for the name `unsafeComparison.cpp` of the following program.

Unsafe comparison of integers

```cpp
// unsafeComparison.cpp

#include <iostream>

int main() {

    std::cout << '\n';

    std::cout << std::boolalpha;

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

    std::cout << "-3 < 7:  " << (x < y) << '\n';
    std::cout << "-3 <= 7: " << (x <= y) << '\n';
    std::cout << "-3 > 7:  " << (x > y) << '\n';
    std::cout << "-3 => 7: " << (x >= y) << '\n';

    std::cout << '\n';

}
```

When I execute the program, the output may not meet your expectations.



Surprises with unsafe comparisons of integers

When you read the output of the program, you recognize that `-3` is bigger 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

```
// unsafeComparison2.cpp

int main() {
    int x = -3;
    unsigned int y = 7;

    bool val = x < y;
    static_assert(static_cast<unsigned int>(-3) == 4'294'967'293);
}
```

In the example, I'm focusing on the less-than operator. C++ Insights[14] gives me the following output:

[14] https://cppinsights.io/s/62732a01

```
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:

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[15] to `double`. C++ Insights[16] shows the truth:

```
int main()
{
  double x = -3.5;
  unsigned int y = 7;
  bool res = x < static_cast<double>(y);
}
```

Floating point promotion to `double`

5.4.2 Mathematical Constants

First of all, the constants require the header `<numbers>` and the namespace `std::numbers`. The following table gives you an overview.

[15] https://en.cppreference.com/w/cpp/language/implicit_conversion
[16] https://cppinsights.io/s/44216566

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`	$\ln 2$
`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[17]
`std::numbers::phi`	ϕ

The program `mathematicConstants.cpp` applies the mathematical constants.

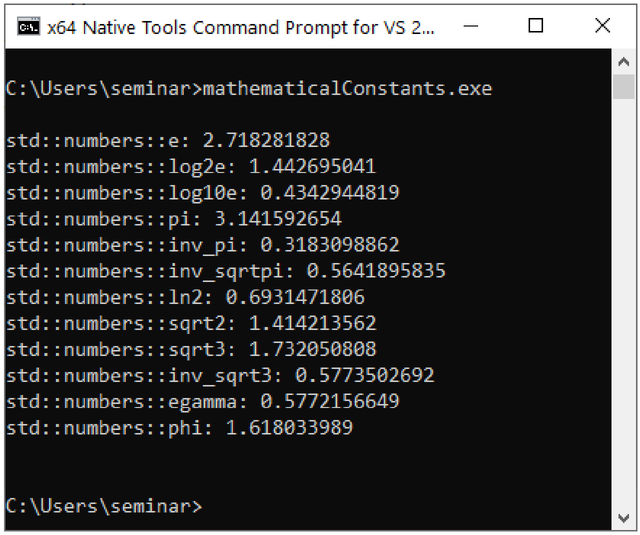
[17] https://en.wikipedia.org/wiki/Euler%E2%80%93Mascheroni_constant

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_sqr\
tpi << '\n';
    std::cout << "std::numbers::ln2: " <<  std::numbers::ln2 << '\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_sqrt\
3 << '\n';
    std::cout << "std::numbers::egamma: " <<  std::numbers::egamma << '\
\n';
    std::cout << "std::numbers::phi: " <<  std::numbers::phi << '\n';

    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

```cpp
// 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

5.4.4 Bit Manipulation

The header `<bit>` supports 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 conditionally compile source code. This means that the 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

Bit manipulation

Function	Description
`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 integer type (`unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`).

The program `bit.cpp` shows the application of the functions.

Bit manipulation

```
// bit.cpp

#include <bit>
#include <bitset>
#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::b\
it_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::ro\
tl(num, 2))
              << '\n';
    std::cout << "std::rotr(0b00110010, 2): " << std::bitset<8>(std::ro\
tr(num, 2))
              << '\n';

    std::cout << "std::countl_zero(0b00110010): " << std::countl_zero(n\
um) << '\n';
    std::cout << "std::countl_one(0b00110010): " << std::countl_one(num\
) << '\n';
    std::cout << "std::countr_zero(0b00110010): " << std::countr_zero(n\
um) << '\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::bit_popcount` for the numbers 2 to 7.

Displaying `std::bit_floor`, `std::bit_ceil`, `std::bit_width`, and `std::popcount` for a few numbers

```cpp
// 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 Calendar and Time Zones

Cippi studies the calendar

Lack of Compiler Support

At the end of 2020, no C++ compiler supports the chrono extensions so far. Thanks to the prototype library date[18] from Howard Hinnant, which is essentially a superset of the extended time functionality in C++20, I can experiment with it. The library is hosted on GitHub. There are various ways to use the date prototype:

- You can try it out on Wandbox. Howard has uploaded the `date.h` header, which is sufficient to play with the new type `std::time_of_day` and the calendar. Here is Howard's link: Try it out on Wandbox![19].
- Copy the header `date.h` into the search path of your C++ compiler.
- Download the project and build it. The already mentioned GitHub page date[20] gives you more information. This step is required when you want to try out the new time zone features.

The examples in this chapter use Howard Hinnant's library. My explanations, though, are based on the C++20 terminology. When a C++ compiler supports the extended chrono functionality, I will adapt the examples to the C++20 syntax.

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, a month, a weekday, or the n-th day of a week.
- A **time zone** represents time specific to a geographic area.

Essentially, the time-zone functionality (C++20) is based on the calendar functionality (C++20), and the calendar functionality (C++20) is based on the chrono functionality (C++11).

[18]https://github.com/HowardHinnant/date
[19]https://wandbox.org/permlink/L8MwjzSSC3fXXrMd
[20]https://github.com/HowardHinnant/date

The Time Library in C++11

To get the most out of this section, a basic understanding of the chrono library is essential. C++11 introduced three main components to deal with time:

- A **time point** is defined by a starting point, the so-called epoch, and additional time duration.
- A **time duration** is the difference between two time points. It is given by the number of ticks.
- A **clock** consists of a starting point (epoch) and a tick, so that the current time point can be calculated.

Honestly, time, for me, is a mystery. On one hand, each of us has an intuitive idea of time, on the other hand, defining it formally is extremely challenging. For example, the three components time point, time duration, and clock depend on each other. If you want to know more about the time functionality in C++11, read my posts about time from time[21].

This is not all. The C++20 extension includes new clocks. Thanks to the formatting library in C++20, time durations can comfortably be read or written.

5.5.1 Time of day

`std::chrono::hh_mm_ss` is the 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 you a concise overview of `std::chrono::hh_mm_ss` instance `tOfDay`.

[21] https://www.modernescpp.com/index.php/tag/time

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
`tOfDay.to_duration()`	Returns the time duration since midnight
`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.

The use of the functions is straightforward.

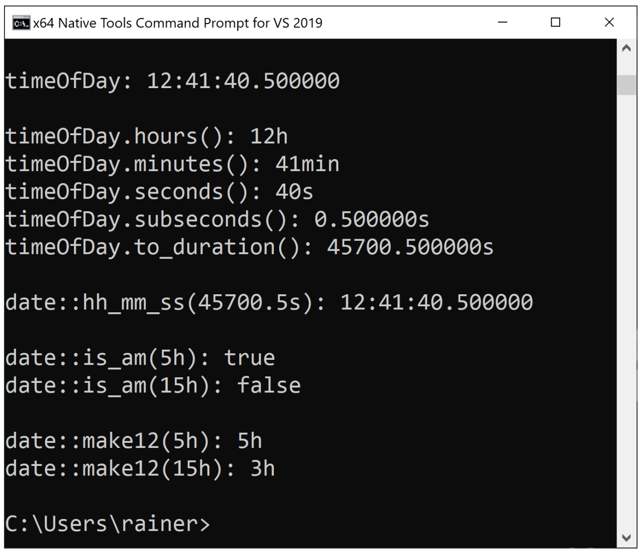
Time of day

```
// timeOfDay.cpp

#include "date.h"
#include <iostream>

int main() {
    using namespace date;                                        \

    using namespace std::chrono_literals;

    std::cout << std::boolalpha << '\n';
    auto timeOfDay = date::hh_mm_ss(10.5h + 98min + 2020s + 0.5s);

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

```

```
16      std::cout << '\n';
17
18      std::cout << "timeOfDay.hours(): " << timeOfDay.hours() << '\n';   \
19
20      std::cout << "timeOfDay.minutes(): " << timeOfDay.minutes() << '\n\
21  ';
22      std::cout << "timeOfDay.seconds(): " << timeOfDay.seconds() << '\n\
23  ';
24      std::cout << "timeOfDay.subseconds(): " << timeOfDay.subseconds() \
25  << '\n';
26      std::cout << "timeOfDay.to_duration(): " << timeOfDay.to_duration(\
27  ) << '\n';
28
29      std::cout << '\n';
30
31      std::cout << "date::hh_mm_ss(45700.5s): " << date::hh_mm_ss(45700.\
32  5s) << '\n';
33
34      std::cout << '\n';
35
36      std::cout << "date::is_am(5h): " << date::is_am(5h) << '\n';       \
37
38      std::cout << "date::is_am(15h): " << date::is_am(15h) << '\n';
39
40      std::cout << '\n';
41
42      std::cout << "date::make12(5h): " << date::make12(5h) << '\n';
43      std::cout << "date::make12(15h): " << date::make12(15h) << '\n';
44
45  }
```

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). This is the reason I have to introduce the namespace `date` in line 7. 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 30 and 32 return if the given hour is a.m. Line 35 and 36 return the 12-hour equivalent of the given hour.

Here is the output of the program:



Time of day

5.5.2 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 first overview of calendar-date types before I show you various use-cases.

Various calendar-date types

Type	Description
`std::chrono::last_spec`	Indicates the last day or weekday of a month
`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

Various calendar-date types

Type	Description
`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 years and month
`std::chrono::operator /`	Creates a date of the Gregorian calendar

Let me start simple and create a few calendar dates.

5.5.2.1 Create Calendar Dates

The program `createCalendar.cpp` shows various ways to create calendar-related dates.

Create calendar dates

```
// createCalendar.cpp

#include <iostream>
#include "date.h"

int main() {

    std::cout << '\n';

    using namespace date;

    constexpr auto yearMonthDay{year(1940)/month(6)/day(26)};           \

    std::cout << yearMonthDay << " ";
    std::cout << date::year_month_day(1940_y, June, 26_d) << '\n';

    std::cout << '\n';

```

```
    constexpr auto yearMonthDayLast{year(2010)/March/last};
    std::cout << yearMonthDayLast << " ";
    std::cout << date::year_month_day_last(2010_y, month_day_last(month\
(3))) << '\n';

    constexpr auto yearMonthWeekday{year(2020)/March/Thursday[2]};     \

    std::cout << yearMonthWeekday << " ";
    std::cout << date::year_month_weekday(2020_y, month(March), Thursda\
y[2]) << '\n';

    constexpr auto yearMonthWeekdayLast{year(2010)/March/Monday[last]};\

    std::cout << yearMonthWeekdayLast << " ";
    std::cout << date::year_month_weekday_last(2010_y, month(March),
                                               weekday_last(Monday)) <<\
  '\n';

    std::cout << '\n';

    constexpr auto day_{day(19)};
    std::cout << day_  << " ";
    std::cout << date::day(19) << '\n';

    constexpr auto month_{month(1)};
    std::cout << month_  << " ";
    std::cout << date::month(1) << '\n';

    constexpr auto year_{year(1988)};
    std::cout << year_  << " ";
    std::cout << date::year(1988) << '\n';

    constexpr auto weekday_{weekday(5)};
    std::cout << weekday_  << " ";
    std::cout << date::weekday(5) << '\n';

```

```
55     constexpr auto yearMonth{year(1988)/1};
56     std::cout << yearMonth  << " ";
57     std::cout << date::year_month(year(1988), January) << '\n';
58 
59     constexpr auto monthDay{10/day(22)};
60     std::cout << monthDay <<  " ";
61     std::cout << date::month_day(October, day(22)) << '\n';
62 
63     constexpr auto monthDayLast{June/last};
64     std::cout << monthDayLast << " ";
65     std::cout << date::month_day_last(month(6)) << '\n';
66 
67     constexpr auto monthWeekday{2/Monday[3]};
68     std::cout << monthWeekday << " ";
69     std::cout << date::month_weekday(February, Monday[3]) << '\n';
70 
71     constexpr auto monthWeekDayLast{June/Sunday[last]};
72     std::cout << monthWeekDayLast << " ";
73     std::cout << date::month_weekday_last(June, weekday_last(Sunday)) <\
74 < '\n';
75 
76     std::cout << '\n';
77 
78 }
```

There are essentially two ways to create a calendar date. You can use the so-called cute syntax `yearMonthDay{year(1940)/month(6)/day(26)}` (line 12), or you can use the explicit type `date::year_month_day(1940y, June, 26d)` (line 14). In order not to overwhelm you, I will delay my explanation of the cute syntax to the next section. The explicit type is quite interesting, because it uses the date-time literals `1940y`, `26d`, and the predefined constant `June`. This was the obvious part of the program.

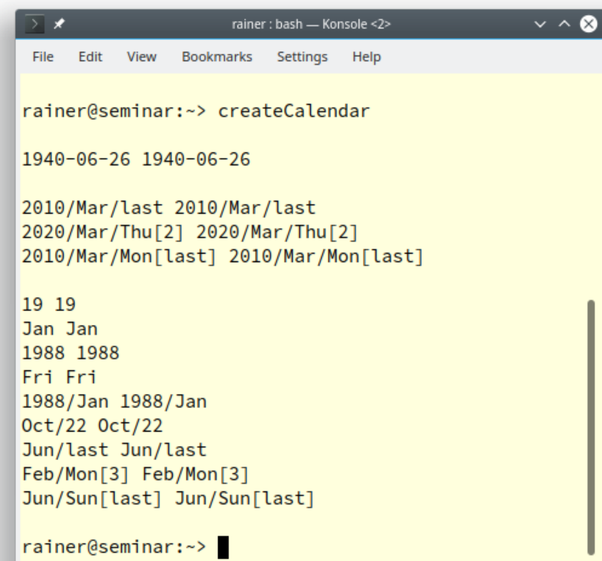
Line 18, line 22, and line 26 offer further ways to create calendar dates.

- Line 18: the last day of March 2010: `{year(2010)/March/last}` or `year_month_day_last(2010y, month_day_last(month(3)))`

- Line 22: the second Thursday of March 2020: {year(2020)/March/Thursday[2]} or year_month_weekday(2020y, month(March), Thursday[2])
- Line 26: 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 33), a month (line 37), or a year (line 41). You can combine and use them as basic building blocks for fully specified calendar dates, such as in lines 18, 22, or 26.

This is the output of the program:



Various calendar days

As promised, let me write about the cute syntax.

5.5.2.2 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 constants

(`January`, `February`, `March`, `April`, `May`, `June`, `July`, `August`, `September`, `October`, `November`, `December`).

The following three combinations of year, month, and day are possible when you use the cute syntax.

Cute syntax

`year/month/day`
`day/month/year`
`month/day/year`

These combinations are not arbitrarily chosen. They are the ones used worldwide. Any other combination is not allowed.

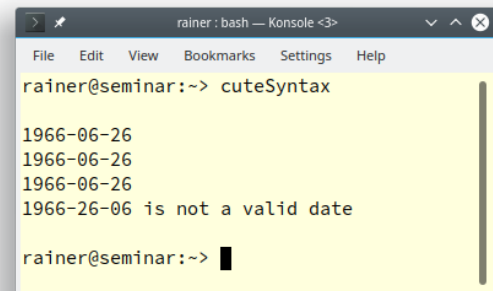
Consequently, when you choose the type `year`, `month`, or `day` for the first argument, the type for the remaining two arguments is no longer necessary anymore, and a number would do the job.

Cute syntax

```cpp
// cuteSyntax.cpp

#include <iostream>
#include "date.h"

int main() {

    std::cout << '\n';

    using namespace date;

    constexpr auto yearMonthDay{year(1966)/6/26};
    std::cout << yearMonthDay << '\n';

    constexpr auto dayMonthYear{day(26)/6/1966};
    std::cout << dayMonthYear << '\n';

    constexpr auto monthDayYear{month(6)/26/1966};
```

```
19      std::cout << monthDayYear << '\n';
20
21      constexpr auto yearDayMonth{year(1966)/month(26)/6};
22      std::cout << yearDayMonth << '\n';
23
24      std::cout << '\n';
25
26  }
```

The combination year/day/month (line 21) 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.

5.5.2.3 Displaying Calendar Dates

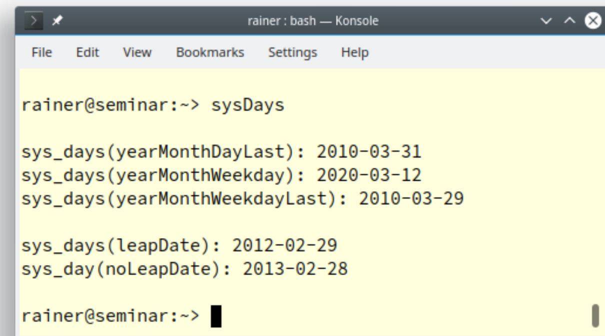
Thanks to `std::chrono::local_days` or `std::chrono::sys_days`, you can convert calendar dates to a `std::chrono::time_point`. I use `std::chrono::sys_days` in my example. `std::chrono::sys_days` is based on `std::chrono::system_clock`[22]. Let me convert the calendar dates (lines 18, 22, and 26) from the previous program `createCalendar.cpp`.

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

Displaying calendar dates

```
// sysDays.cpp

#include <iostream>
#include "date.h"

int main() {

    std::cout << '\n';

    using namespace date;

    constexpr auto yearMonthDayLast{year(2010)/March/last};
    std::cout << "sys_days(yearMonthDayLast): "
              << sys_days(yearMonthDayLast)  <<  '\n';

    constexpr auto yearMonthWeekday{year(2020)/March/Thursday[2]};
    std::cout << "sys_days(yearMonthWeekday): "
              <<  sys_days(yearMonthWeekday) << '\n';

    constexpr auto yearMonthWeekdayLast{year(2010)/March/Monday[last]};
    std::cout << "sys_days(yearMonthWeekdayLast): "
              << sys_days(yearMonthWeekdayLast) << '\n';

    std::cout << '\n';

    constexpr auto leapDate{year(2012)/February/last};
    std::cout << "sys_days(leapDate): " << sys_days(leapDate) << '\n';

    constexpr auto noLeapDate{year(2013)/February/last};
    std::cout << "sys_day(noLeapDate): " << sys_days(noLeapDate) << '\n\
';

    std::cout << '\n';

}
```

The `std::chrono::last` constant (line 11) lets me easily determine how many days a month has. The output shows that 2012 is a leap year (line 26), but not 2013 (line 29).



Displaying calendar dates

Assume you have a calendar date such as `year(2100)/2/29`. Your first question may be: Is this date valid?

5.5.2.4 Check if a Date is valid

The various calendar types in C++20 have a function `ok`. This function returns true if the date is valid.

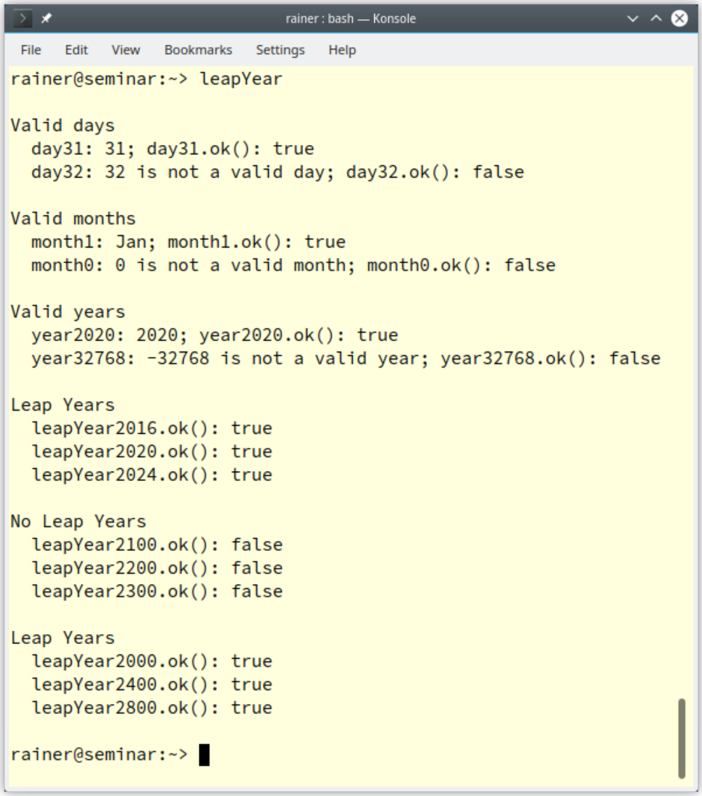
Checking if a date is valid

```
//  leapYear.cpp

#include <iostream>
#include "date.h"

int main() {

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

    using namespace date;

    std::cout << "Valid days" << '\n';
    day day31(31);
    day day32 = day31 + 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';
    month month1(1);
    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';
    year year2020(2020);
    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';

    std::cout << "Leap Years"  << '\n';

    constexpr auto leapYear2016{year(2016)/2/29};
    constexpr auto leapYear2020{year(2020)/2/29};
    constexpr auto leapYear2024{year(2024)/2/29};

    std::cout << "  leapYear2016.ok(): " << leapYear2016.ok() << '\n';
    std::cout << "  leapYear2020.ok(): " << leapYear2020.ok() << '\n';
    std::cout << "  leapYear2024.ok(): " << leapYear2024.ok() << '\n';

    std::cout << '\n';

     std::cout << "No Leap Years"  << '\n';

    constexpr auto leapYear2100{year(2100)/2/29};
    constexpr auto leapYear2200{year(2200)/2/29};
    constexpr auto leapYear2300{year(2300)/2/29};

    std::cout << "  leapYear2100.ok(): " << leapYear2100.ok() << '\n';
    std::cout << "  leapYear2200.ok(): " << leapYear2200.ok() << '\n';
    std::cout << "  leapYear2300.ok(): " << leapYear2300.ok() << '\n';

    std::cout << '\n';

    std::cout << "Leap Years"  << '\n';

    constexpr auto leapYear2000{year(2000)/2/29};
    constexpr auto leapYear2400{year(2400)/2/29};
    constexpr auto leapYear2800{year(2800)/2/29};
```

```
72
73     std::cout << "  leapYear2000.ok(): " << leapYear2000.ok() << '\n';
74     std::cout << "  leapYear2400.ok(): " << leapYear2400.ok() << '\n';
75     std::cout << "  leapYear2800.ok(): " << leapYear2800.ok() << '\n';
76
77     std::cout << '\n';
78
79 }
```

I check in the program if a given day (line 12), a given month (line 23), or a given year (line 33) is valid. The range of a day is [1, 31], of a month [1, 12], and of a year [-32767, 32767]. Consequently, the `ok()` calls on the corresponding values returns false. Two facts are interesting when I display various values. First, if the value is not valid, the output displays: "is not a valid day", "is not a valid month", "is not a valid year". Second, month values are displayed in string representation.



```
rainer@seminar:~> leapYear

Valid days
  day31: 31; day31.ok(): true
  day32: 32 is not a valid day; day32.ok(): false

Valid months
  month1: Jan; month1.ok(): true
  month0: 0 is not a valid month; month0.ok(): false

Valid years
  year2020: 2020; year2020.ok(): true
  year32768: -32768 is not a valid year; year32768.ok(): false

Leap Years
  leapYear2016.ok(): true
  leapYear2020.ok(): true
  leapYear2024.ok(): true

No Leap Years
  leapYear2100.ok(): false
  leapYear2200.ok(): false
  leapYear2300.ok(): false

Leap Years
  leapYear2000.ok(): true
  leapYear2400.ok(): true
  leapYear2800.ok(): true

rainer@seminar:~>
```

Check if a data is valid

You can apply the `ok`-call on a calendar date. Now it's quite 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[23], the following rules apply:

Each year that is exactly divisible by **4** is a **leap year**.

- Except for years which are exactly divisible by **100**. They are **not leap years**.
  - Except for years which are exactly divisible by **400**. They are **leap years**.

Too complicated? The program `leapYears.cpp` exemplifies this rule.

The extended chrono library makes it quite easy to ask for the time duration between calendar dates.

[23] https://en.wikipedia.org/wiki/Gregorian_calendar

5.5.2.5 Query Calendar Dates

Without further ado. The following program `queryCalendarDates.cpp` queries a few calendar dates.

Query calendar dates

```
// queryCalendarDates.cpp

#include "date.h"
#include <iostream>

int main() {

    using namespace date;

    std::cout << '\n';

    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<da\
ys>(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();
```

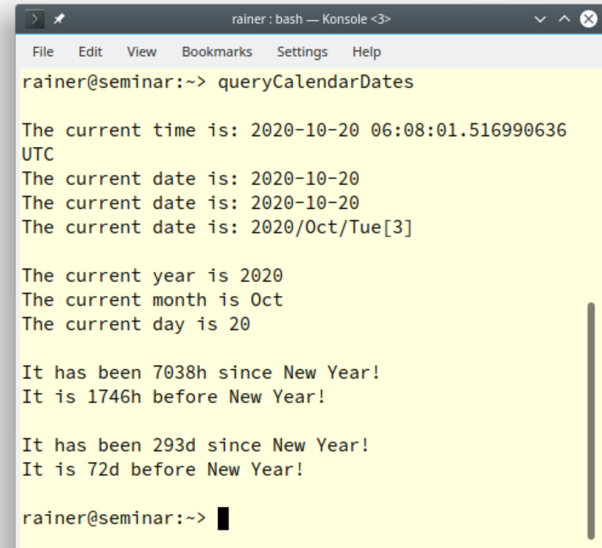
```
31      std::cout << "The current day is " << currentDay << '\n';
32
33      std::cout << '\n';
34
35      auto hAfter = floor<std::chrono::hours>(now) - sys_days(January/1/c\
36  urrentYear);
37      std::cout << "It has been " << hAfter << " since New Year!\n";
38      auto nextYear = currentDate.year() + years(1);
39      auto nextNewYear = sys_days(January/1/nextYear);
40      auto hBefore =  sys_days(January/1/nextYear) - floor<std::chrono::h\
41  ours>(now);
42      std::cout << "It is " << hBefore << " before New Year!\n";
43
44      std::cout << '\n';
45
46      std::cout << "It has been " << floor<days>(hAfter) << " since New Y\
47  ear!\n";
48      std::cout << "It is " << floor<days>(hBefore) << " before New Year!\
49  \n";
50
51      std::cout << '\n';
52
53  }
```

With the C++20 extension, you can directly display a time point, such as now (line 12). `std::chrono::floor` converts the time point 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 23).

Line 33 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 37) the current date, using hour resolution, I get the hours to the new year.

Maybe you don't like hour resolution. Line 42 and 43 display the values using day resolution.



Query calendar days

Now, I want to know the day of the week of my birthday.

5.5.2.6 Query Weekdays

Thanks to the extended chrono library, it is quite easy to get the weekday of a given calendar date.

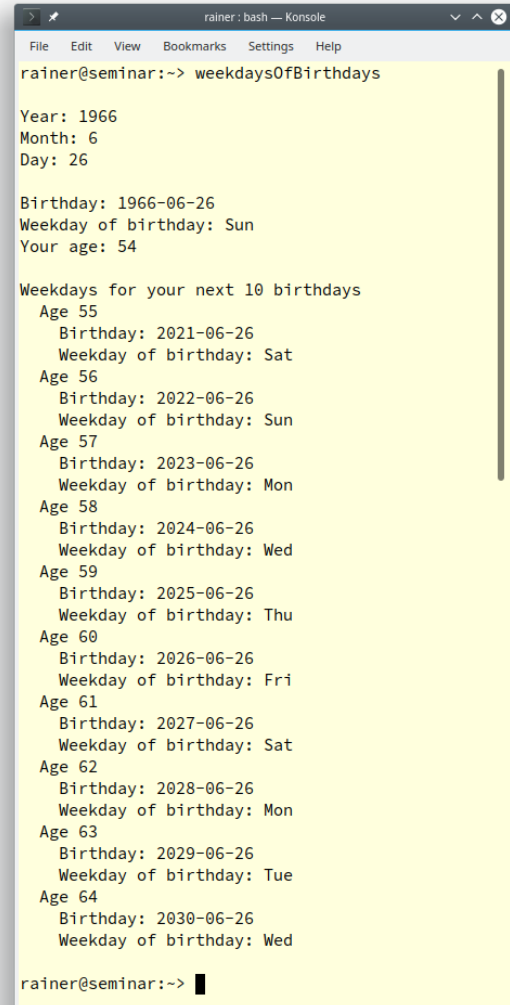
Weekdays of given calendar dates

```
// weekdaysOfBirthdays.cpp

#include <cstdlib>
#include <iostream>
#include "date.h"

int main() {

    std::cout << '\n';

    using namespace date;

    int y;
    int m;
    int d;

    std::cout << "Year: ";
    std::cin >> y;
    std::cout << "Month: ";
    std::cin >> m;
    std::cout << "Day: ";
    std::cin >> d;

    std::cout << '\n';

    auto birthday = year(y)/month(m)/day(d);

    if (not birthday.ok()) {
        std::cout << birthday << '\n';
        std::exit(EXIT_FAILURE);
    }

    std::cout << "Birthday: " << birthday << '\n';
    auto birthdayWeekday = year_month_weekday(birthday);
    std::cout << "Weekday of birthday: " << birthdayWeekday.weekday() <\
```

```
36 < '\n';
37
38     auto currentDate = year_month_day(floor<days>(
39                                 std::chrono::system_clock::now())\
40 );
41     auto currentYear = currentDate.year();
42
43     auto age = (int)currentDate.year() - (int)birthday.year();
44     std::cout << "Your age: " << age << '\n';
45
46     std::cout << '\n';
47
48     std::cout << "Weekdays for your next 10 birthdays" << '\n';
49
50     for (int i = 1, newYear = (int)currentYear; i <= 10;  ++i ) {
51         std::cout << "  Age " <<  ++age << '\n';
52         auto newBirthday = year(++newYear)/month(m)/day(d);
53         std::cout << "    Birthday: " << newBirthday << '\n';
54         std::cout << "    Weekday of birthday: "
55                   << year_month_weekday(newBirthday).weekday() << '\n';
56     }
57
58     std::cout << '\n';
59
60 }
```

First, the program asks you for the year, month, and day of your birthday (line 17). Based on the input, a calendar date is created (line 26) and checked if it's valid (line 28). Now I display the weekday of your birthday. I use the calendar date to fill the calendar type `std::chrono::year_month_weekday` (line 34). To get the `int` representation of the calendar type `year`, I have to convert it to `int` (line 41). Now I can display your age. Finally, the for loop displays, for each of your next ten birthdays (line 46), the following information: your age, the calendar date, and the weekday. I just have to increment the `age` and `newYear` variable.

Here is a run of the program with my birthday.



Weekdays of birthdays

5.5.2.7 Calculating Ordinal Dates

As a last example of the new calendar facility, I want to present the online resource Examples and Recipes[24] from Howard Hinnant, which has about 40 examples of the

[24] https://github.com/HowardHinnant/date/wiki/Examples-and-Recipes

new chrono functionality. Presumably, the chrono extension in C++20 is not easy to get, therefore it's quite important 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 program by Roland Bock[25] 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.

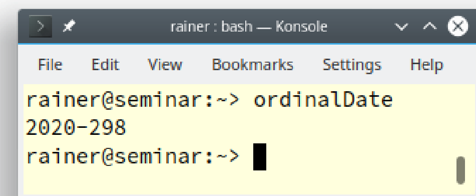
Calculating ordinal dates

```
// ordinalDate.cpp

#include "date.h"
#include <iomanip>
#include <iostream>

int main()
{
   using namespace date;

   const auto time = std::chrono::system_clock::now();
   const auto daypoint = floor<days>(time);
   const auto ymd = year_month_day{daypoint};

   // calculating the year and the day of the year
   const auto year = ymd.year();
   const auto year_day = daypoint - sys_days{year/January/0};

   std::cout << year << '-' << std::setfill('0') << std::setw(3)
```

[25] https://github.com/rbock

```
20                  << year_day.count() << '\n';
21
22     // inverse calculation and check
23     assert(ymd == year_month_day{sys_days{year/January/0} + year_day});
24 }
```

I want to make a few remarks about the program. Line 12 truncates the current time point. The value is used in the following line to initialize a calendar date. Line 17 calculates the time duration between the two time points. Both time points have the resolution day. Finally, `year_day.count()` in line 19 returns the time duration in days.



Caculating ordinal dates

5.5.3 Time Zones

First of all, a time zone is a region, and its full history of the date, such as daylight saving time or leap seconds. The time zone library in C++20 is a complete parser of the IANA timezone database[26]. The following table should give you a first idea of the new functionality.

[26] https://www.iana.org/timezones

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
`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

I use in my examples the function `std::chrono::zones_time`, which is essentially a time zone combined with a time point.

Compilation of the examples

Before I show you two examples, I want to make a short remark. To compile a program using the time zone library, you have to compile the `tz.cpp` file from the date[27] library and link it against the curl[28] library. The curl library is necessary to get the current IANA timezone database[29]. The following command line for g++ should give you the idea:

Compilation with the prototype library date

```
g++ localTime.cpp -I <Path to data/tz.h> tz.cpp -std=c++17 -lcurl -o lo\
calTime
```

My first example is straightforward. It displays the UTC time and the local time.

5.5.3.1 UTC Time and Local Time

The UTC time or Coordinated Universal Time[30] is the primary time standard worldwide. A computer uses Unix time[31] which is 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.

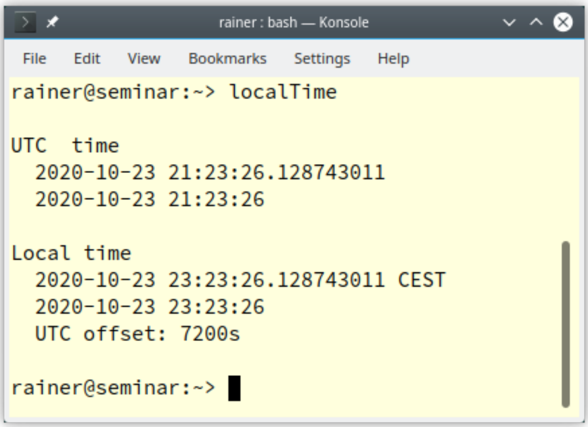
`std::chrono::system_clock::now()` returns in the program `localTime.cpp` the Unix time.

[27] https://github.com/HowardHinnant/date
[28] https://curl.se/
[29] https://www.iana.org/timezones
[30] https://en.wikipedia.org/wiki/Coordinated_Universal_Time
[31] https://en.wikipedia.org/wiki/Unix_time

Getting the UTC time and local time

```
1 // localTime.cpp
2
3 #include "date/tz.h"
4 #include <iostream>
5
6 int main() {
7
8   std::cout << '\n';
9
10   using namespace date;
11
12   std::cout << "UTC  time" << '\n';
13   auto utcTime = std::chrono::system_clock::now();
14   std::cout << "  " << utcTime << '\n';
15   std::cout << "  " << date::floor<std::chrono::seconds>(utcTime) << '\
16 n';
17
18   std::cout << '\n';
19
20   std::cout << "Local time" << '\n';
21   auto localTime = date::make_zoned(date::current_zone(), utcTime);
22   std::cout << "  " << localTime << '\n';
23   std::cout << "  " << date::floor<std::chrono::seconds>(localTime.get_\
24 local_time())
25             << '\n';
26
27   auto offset = localTime.get_info().offset;
28   std::cout << "  UTC offset: "  << offset << '\n';
29
30   std::cout << '\n';
31
32 }
```

The code block beginning with line 12 gets the current time point, truncates it to seconds, and displays it. The call `make_zoned` (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 25) 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?

5.5.3.2 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 and should take four hours. Because of daylight saving time, the calendar date is essential to get the correct answer.

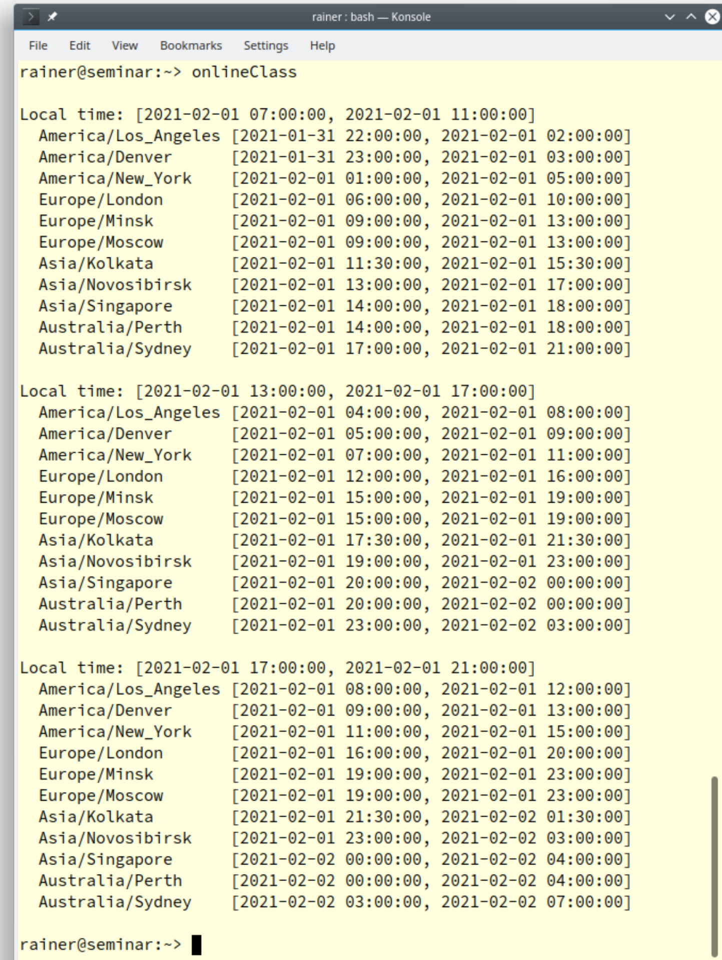
Calculating the time in different time zones

```
// onlineClass.cpp

#include "date/tz.h"
#include <algorithm>
#include <iomanip>
#include <iostream>

template <typename ZonedTime>
auto getMinutes(const ZonedTime& zonedTime) {
    return date::floor<std::chrono::minutes>(zonedTime.get_local_time()\
);
}

void printStartEndTimes(const date::local_days& localDay,
                        const std::chrono::hours& h,
                        const std::chrono::hours& durationClass,
                        const std::initializer_list<std::string>& timeZ\
ones ){

    date::zoned_time startDate{date::current_zone(), localDay + h};
    date::zoned_time endDate{date::current_zone(), localDay + h + durat\
ionClass};
    std::cout << "Local time: ["  << getMinutes(startDate) << ", "
                                  << getMinutes(endDate) << "]" << '\n';

   longestStringSize = std::max(timeZones, [](const std::string& a,
                       const std::string& b) { return a.size() < b.size\
(); }).size();
    for (auto timeZone: timeZones) {
        std::cout << "  " << std::setw(longestStringSize + 1) << std::l\
eft
                  << timeZone
                  << "[" << getMinutes(date::zoned_time(timeZone, start\
Date))
                  << ", " << getMinutes(date::zoned_time(timeZone, endD\
```

```
36 ate))
37                   << "]" << '\n';
38
39     }
40  }
41
42 int main() {
43
44     using namespace std::string_literals;
45     using namespace std::chrono;
46
47     std::cout << '\n';
48
49     constexpr auto classDay{date::year(2021)/2/1};
50     constexpr auto durationClass = 4h;
51     auto timeZones = {"America/Los_Angeles"s, "America/Denver"s,
52                       "America/New_York"s, "Europe/London"s,
53                       "Europe/Minsk"s, "Europe/Moscow"s,
54                       "Asia/Kolkata"s, "Asia/Novosibirsk"s,
55                       "Asia/Singapore"s, "Australia/Perth"s,
56                       "Australia/Sydney"s};
57
58     for (auto startTime: {7h, 13h, 17h}) {
59         printStartEndTimes(date::local_days{classDay}, startTime,
60                            durationClass, timeZones);
61         std::cout << '\n';
62     }
63
64 }
```

Before I dive into the functions `getMinutes` (line 8) and `printStartEndTimes` (line 13), 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 51) iterates through all potential starting points for an online class. Thanks to the function `printStartEndTimes` (line 13), all necessary information is displayed.

The few lines beginning with line 18 calculate the `startDate` and `endDate` of my training by adding the start time and the duration of the class to the calendar date. Both values are displayed with the help of the function `getMinutes` (line 8). `floor<std::chrono::minutes>(zonedTime.get_local_time())` gets the stored timepoint out of the `std::chrono::zoned_time` and truncates the value to the minute resolution. To properly align the output of the program, line 23 determines the size of the longest of all timezone names. Line 25 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.



```
rainer@seminar:~> onlineClass

Local time: [2021-02-01 07:00:00, 2021-02-01 11:00:00]
  America/Los_Angeles [2021-01-31 22:00:00, 2021-02-01 02:00:00]
  America/Denver      [2021-01-31 23:00:00, 2021-02-01 03:00:00]
  America/New_York    [2021-02-01 01:00:00, 2021-02-01 05:00:00]
  Europe/London       [2021-02-01 06:00:00, 2021-02-01 10:00:00]
  Europe/Minsk        [2021-02-01 09:00:00, 2021-02-01 13:00:00]
  Europe/Moscow       [2021-02-01 09:00:00, 2021-02-01 13:00:00]
  Asia/Kolkata        [2021-02-01 11:30:00, 2021-02-01 15:30:00]
  Asia/Novosibirsk    [2021-02-01 13:00:00, 2021-02-01 17:00:00]
  Asia/Singapore      [2021-02-01 14:00:00, 2021-02-01 18:00:00]
  Australia/Perth     [2021-02-01 14:00:00, 2021-02-01 18:00:00]
  Australia/Sydney    [2021-02-01 17:00:00, 2021-02-01 21:00:00]

Local time: [2021-02-01 13:00:00, 2021-02-01 17:00:00]
  America/Los_Angeles [2021-02-01 04:00:00, 2021-02-01 08:00:00]
  America/Denver      [2021-02-01 05:00:00, 2021-02-01 09:00:00]
  America/New_York    [2021-02-01 07:00:00, 2021-02-01 11:00:00]
  Europe/London       [2021-02-01 12:00:00, 2021-02-01 16:00:00]
  Europe/Minsk        [2021-02-01 15:00:00, 2021-02-01 19:00:00]
  Europe/Moscow       [2021-02-01 15:00:00, 2021-02-01 19:00:00]
  Asia/Kolkata        [2021-02-01 17:30:00, 2021-02-01 21:30:00]
  Asia/Novosibirsk    [2021-02-01 19:00:00, 2021-02-01 23:00:00]
  Asia/Singapore      [2021-02-01 20:00:00, 2021-02-02 00:00:00]
  Australia/Perth     [2021-02-01 20:00:00, 2021-02-02 00:00:00]
  Australia/Sydney    [2021-02-01 23:00:00, 2021-02-02 03:00:00]

Local time: [2021-02-01 17:00:00, 2021-02-01 21:00:00]
  America/Los_Angeles [2021-02-01 08:00:00, 2021-02-01 12:00:00]
  America/Denver      [2021-02-01 09:00:00, 2021-02-01 13:00:00]
  America/New_York    [2021-02-01 11:00:00, 2021-02-01 15:00:00]
  Europe/London       [2021-02-01 16:00:00, 2021-02-01 20:00:00]
  Europe/Minsk        [2021-02-01 19:00:00, 2021-02-01 23:00:00]
  Europe/Moscow       [2021-02-01 19:00:00, 2021-02-01 23:00:00]
  Asia/Kolkata        [2021-02-01 21:30:00, 2021-02-02 01:30:00]
  Asia/Novosibirsk    [2021-02-01 23:00:00, 2021-02-02 03:00:00]
  Asia/Singapore      [2021-02-02 00:00:00, 2021-02-02 04:00:00]
  Australia/Perth     [2021-02-02 00:00:00, 2021-02-02 04:00:00]
  Australia/Sydney    [2021-02-02 03:00:00, 2021-02-02 07:00:00]

rainer@seminar:~>
```

Displaying start and end times in various time zones

5.5.3.3 New Clocks

Beside the wall clock `std::system_clock`[32], the monotonic clock `std::steady_clock`[33], and the most precise clock `std::high_resolution_clock`[34] in C++11, C++20 supports five additional clocks.

- **std::utc_clock**: Clock for the coordinated Universal Time (UTC). Measures the time since 00:00:00 UTC, 1 January 1970, including leap seconds.
- **std::tai_clock**: Clock for International Atomic Time[35] (TAI). Measure time since 00:00:00, 1 January 1958, and is offset 10 seconds ahead of UTC at that date. Leap seconds are not inserted.
- **std::gps_clock**: Clock for GPS time. It represents Global Positioning System[36] (GPS) time. It measures the time since 00:00:00, 6 January 1980 UTC. Leap seconds are not inserted.
- **std::file_clock**: Clock for file time. It's an alias for `std::filesystem::file_time_type`[37].
- **std::local_t**: Pseudo clock to represent local time.

5.5.3.4 Chrono I/O

Thanks to the function `std::chrono::parse` and the `std::formatter` from the formatting library, you can read and write chrono objects.

- **std::chrono::parse**: Parses a chrono object from a stream. cppreference.com/parse[38] gives you detailed infomation about the format string.
- **std::formatter**: Defines specializations for the various chrono types. Read the details on the format specification on `std::formatter` here cppreference.com/formatter[39].

[32]https://www.modernescpp.com/index.php/the-three-clocks
[33]https://www.modernescpp.com/index.php/the-three-clocks
[34]https://www.modernescpp.com/index.php/the-three-clocks
[35]https://en.wikipedia.org/wiki/International_Atomic_Time
[36]https://en.wikipedia.org/wiki/Global_Positioning_System
[37]https://en.cppreference.com/w/cpp/filesystem/file_time_type
[38]https://en.cppreference.com/w/cpp/chrono/parse
[39]https://en.cppreference.com/w/cpp/chrono/system_clock/formatter#Format_specification

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.

5.6 Formatting Library

Cippi forms a cup

Lack of Compiler Support

At the end of 2020, no C++ compiler supports the formatting library. Thanks to the prototype library fmt[40] by Victor Zverovich, I can experiment with it. The library is hosted on the Compiler Explorer[41]. Once one of the big three compilers GCC, Clang, or MSVC supports the C++20 formatting library, I will replace the examples in this chapter.

The formatting library offers a secure and expandable alternative to the `printf`[42] family and extends the I/O streams. The library requires the header <format>. The format specification follows Python syntax[43] and 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.

5.6.0.1 Formatting Functions

C++20 supports three formatting functions:

---

[40] https://github.com/fmtlib/fmt
[41] https://godbolt.org/z/Eq5763
[42] https://en.cppreference.com/w/cpp/io/c/fprintf
[43] https://docs.python.org/3/library/stdtypes.html#str.format

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

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`.

Calculating the time in different time zones

```
// format.cpp

#include <fmt/core.h>
#include <fmt/format.h>
#include <iostream>
#include <iterator>
#include <string>

int main() {

    std::cout << '\n';

    std::cout << fmt::format("Hello, C++{}!\n", "20") << '\n';

    std::string buffer;

    fmt::format_to(
        std::back_inserter(buffer),
        "Hello, C++{}!\n",
        "20");

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

```
23
24      buffer.clear();
25
26      fmt::format_to_n(
27          std::back_inserter(buffer), 5,
28          "Hello, C++{}!\n",
29          "20");
30
31      std::cout << buffer << '\n';
32
33
34      std::cout << '\n';
35
36  }
```

The program on line 13 directly displays the formatted string. The calls on line 17 and 26, though, use a string as a buffer. Additionally, `std::format_to_n` pushes only five characters onto the buffer.

```
Hello, C++20!

Hello, C++20!

Hello
```

Formatted output

Presumably, the most interesting part of the three formatting functions is the format string (`"Hello, C++{}!\n"`).

5.6.1 Format String

The formatting string syntax is identical for the formatting functions `std::format`, `std::format_to`, and `std::format_to_n`. 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 inside the replacement field an argument id and a colon 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[44] based on Python's format specification[45]
- Chrono types: Chrono format specification[46]
- Other formattable types: User-defined `std::formatter` specialization

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.

5.6.1.1 Argument ID

Thanks to the argument id, you can reorder the arguments or address particular arguments.

[44]https://en.cppreference.com/w/cpp/utility/format/formatter#Standard_format_specification
[45]https://docs.python.org/3/library/stdtypes.html#str.format
[46]https://en.cppreference.com/w/cpp/chrono/system_clock/formatter#Format_specification

Using the argument id

```
1  // formatArgumentID.cpp
2  
3  #include <fmt/core.h>
4  #include <iostream>
5  #include <string>
6  
7  int main() {
8  
9      std::cout << '\n';
10 
11     std::cout << fmt::format("{} {}: {}!\n", "Hello", "World", 2020);
12 
13     std::cout << fmt::format("{1} {0}: {2}!\n", "World", "Hello", 2020);
14 
15     std::cout << fmt::format("{0} {0} {1}: {2}!\n", "Hello", "World", 2\
16 020);
17 
18     std::cout << fmt::format("{0}: {2}!\n", "Hello", "World", 2020);
19 
20     std::cout << '\n';
21 
22 }
```

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:

```
Hello World: 2020!
Hello World: 2020!
Hello Hello World: 2020!
Hello: 2020!
```

Applying the argument id

Applying the argument id with the format specification makes formatting of text in C++20 very powerful.

5.6.1.2 Format Specification

I'm not going to 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[47]. Accordingly, you can find the details of chrono types here: chrono format specification[48].

Rather, I present the simplified format specification for basic types and string types.

Simplified format specification for basic types and string types

```
fill_align(opt) sign(opt) #(opt) 0(opt) width(opt) precision(opt) type(\
opt)
```

All parts are optional (opt). The next few sections present the parts of this format specification.

5.6.1.2.1 Fill Character and Alignment

The fill character is optional (any character except `{` or `}`) and is followed by an alignment specification.

- Fill character: by default, space is used
- Alignment:
  - `<`: left (default for non-numbers)
  - `>`: right (default for numbers)
  - `^`: center

[47]https://en.cppreference.com/w/cpp/utility/format/formatter#Standard_format_specification
[48]https://en.cppreference.com/w/cpp/chrono/system_clock/formatter#Format_specification

Applying the fill character and alignment

```cpp
// formatFillAlign.cpp

#include <fmt/core.h>
#include <iostream>

int main() {

    std::cout << '\n';

    int num = 2020;

    std::cout << fmt::format("{:6}", num) << '\n';
    std::cout << fmt::format("{:6}", 'x') << '\n';
    std::cout << fmt::format("{:*<6}", 'x') << '\n';
    std::cout << fmt::format("{:*>6}", 'x') << '\n';
    std::cout << fmt::format("{:*^6}", 'x') << '\n';
    std::cout << fmt::format("{:6d}", num) << '\n';
    std::cout << fmt::format("{:6}", true) << '\n';

    std::cout << '\n';

}
```

```
  2020
x
x*****
*****x
**x***
  2020
true
```

Applying the fill character and alignment

5.6.1.2.2 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 <fmt/core.h>
#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';

}
```

```
0,+0,0, 0
0,+0,0, 0
1,+1,1, 1
-1,-1,-1,-1
```

Applying the sign character

The `#` causes the alternate 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 <fmt/core.h>
#include <iostream>

int main() {

    std::cout << '\n';

    std::cout << fmt::format("{:#015}", 0x78) << '\n';
    std::cout << fmt::format("{:#015b}", 0x78) << '\n';
    std::cout << fmt::format("{:#015x}", 0x78) << '\n';

    std::cout << '\n';

    std::cout << fmt::format("{:g}", 120.0) << '\n';
    std::cout << fmt::format("{:#g}", 120.0) << '\n';


    std::cout << '\n';

}
```

```
000000000000120
0b000001111000
0x000000000078

120
120.000
```

Applying the `#` and the `0` characters

5.6.1.2.3 Width and Precision

You can specify the width and the precision of your type. 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 either a non-negative decimal number or a replacement field.

A few examples should help you grasp the basics:

Applying the width and precision specifier

```cpp
// formatWidthPrecision.cpp

#include <fmt/core.h>
#include <iostream>
#include <string>

int main() {

    int i = 123456789;
```

```
    double d = 123.456789;

    std::cout << "---" << fmt::format("{}", i) << "---\n";
    std::cout << "---" << fmt::format("{:15}", i) << "---\n";   // (w =\
 15)
    std::cout << "---" << fmt::format("{:}", i, 15) << "---\n"; // (w =\
 15)

    std::cout << '\n';

    std::cout << "---" << fmt::format("{}", d) << "---\n";
    std::cout << "---" << fmt::format("{:15}", d) << "---\n";    // (w \
= 15)
    std::cout << "---" << fmt::format("{:}", d, 15) << "---\n";  // (w \
= 15)

    std::cout << '\n';

    std::string s= "Only a test";

    std::cout << "---" << fmt::format("{:10.50}", d) << "---\n"; // (w \
= 50, p = 50)
    std::cout << "---" << fmt::format("{:{}.{}}", d, 10, 50) << "---\n"\
;  // (w = 50,
                                                                        \
   //  p = 50)
    std::cout << "---" << fmt::format("{:10.5}", d) << "---\n";  // (w \
= 10, p = 5)
    std::cout << "---" << fmt::format("{:{}.{}}", d, 10, 5) << "---\n";\
  // (w = 10,
                                                                        \
  //  p = 5)

    std::cout << '\n';

    std::cout << "---" << fmt::format("{:.500}", s) << "---\n";      //\
```

```
46  (p = 500)
47     std::cout << "---" << fmt::format("{:.{}}", s, 500) << "---\n";  //\
48  (p = 500)
49     std::cout << "---" << fmt::format("{:.5}", s) << "---\n";        //\
50  (p = 5)
51
52 }
```

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. When you specify the width with a replacement field (line 14), no extra spaces are added. 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).

```
---123456789---
---      123456789---
---123456789---

---123.456789---
---     123.456789---
---123.456789---

---123.4567890000000055570126278325915336608886718750---
---123.4567890000000055570126278325915336608886718750---
---    123.46---
---    123.46---

---Only a test---
---Only a test---
---Only ---
```

Applying the width and precision specifiers

5.6.1.2.4 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`

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`[49].

Thanks to the type specifiers, you can easily display an `int` in a different number system.

[49] https://en.cppreference.com/w/cpp/utility/to_chars

Applying the type specifier

```
// formatType.cpp

#include <fmt/core.h>
#include <iostream>

int main() {

    int num{2020};

    std::cout << "default:     " << fmt::format("{:}", num) << '\n';
    std::cout << "decimal:     " << fmt::format("{:d}", num) << '\n';
    std::cout << "binary:      " << fmt::format("{:b}", num) << '\n';
    std::cout << "octal:       " << fmt::format("{:o}", num) << '\n';
    std::cout << "hexadecimal: " << fmt::format("{:x}", num) << '\n';

}
```

```
default:     2020
decimal:     2020
binary:      11111100100
octal:       3744
hexadecimal: 7e4
```

Applying the type specifier

So far, I've formatted basics types and strings. Additionally, you can format user-defined types.

5.6.2 User-Defined Types

To format a user-defined type, I have to specialize the class `std::formatter`[50] for my user-defined type. This means, in particular, I have to implement the member functions `parse` and `format`.

[50] https://en.cppreference.com/w/cpp/utility/format/formatter

- **parse**:
  - Accepts the parse context
  - Parses the parse context
  - Returns an iterator to the end of the format specification
  - Throws a `std::format_error` in case of an error
- **format**:
  - Gets the value `t`, which should be formatted, and the format context `fc`
  - Formats `t` according to the format context
  - Writes the output to `fc.out()`
  - Returns an iterator that represents the end of the output

Let me put the theory into practice and format a `std::vector`.

5.6.2.1 Formatting a `std::vector`

My first specialization of the class `std::formatter` is as easy as possible. I specify a format specification used for each element of the container.

Applying the format specification to the elements of a `std::vector`

```
// formatVector.cpp

#include <iostream>
#include <fmt/format.h>
#include <string>
#include <vector>

template <typename T>
struct fmt::formatter<std::vector<T>> {

  std::string formatString;

  auto constexpr parse(format_parse_context& ctx) {
    formatString = "{:";
    std::string parseContext(std::begin(ctx), std::end(ctx));
    formatString += parseContext;
```

```
17     return std::end(ctx) - 1;
18   }
19 
20   template <typename FormatContext>
21   auto format(const std::vector<T>& v, FormatContext& ctx) {
22     auto out= ctx.out();
23     fmt::format_to(out, "[");
24     if (v.size() > 0) fmt::format_to(out, formatString, v[0]);
25     for (int i= 1; i < v.size(); ++i) fmt::format_to(out, ", " + format\
26 String, v[i]);
27     fmt::format_to(out, "]");
28     return fmt::format_to(out,  "\n" );
29   }
30 
31 };
32 
33 
34 int main() {
35 
36   std::vector<int> myInts{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
37   std::cout << fmt::format("{:}", myInts);
38   std::cout << fmt::format("{:+}", myInts);
39   std::cout << fmt::format("{:03d}", myInts);
40   std::cout << fmt::format("{:b}", myInts);
41 
42   std::cout << '\n';
43 
44   std::vector<std::string> myStrings{"Only", "for", "testing", "purpose\
45 "};
46   std::cout << fmt::format("{:}", myStrings);
47   std::cout << fmt::format("{:.3}", myStrings);
48 
49 }
```

The specialization for `std::vector` (line 8) has the member functions `parse` (line 13) and `format` (line 20). `parse` essentially creates the `formatString` which is applied to each element of the `std::vector` (lines 24 and 25). The parse context `ctx` (line 13) contains the characters between the colon (`:`) and the closing curly brace (`}`). On end, the function returns an iterator to the closing curly brace (`}`). The job of the member function `format` is more interesting. The format context returns the output iterator. Thanks to the output iterator and the function `std::format_to`[51], the elements of a `std::vector` are nicely displayed.

The elements of the `std::vector` (line 35) are formatted in a few ways. Line 36 displays the number, line 37 writes a sign before each number, line 38 aligns them to 3 characters and uses the `0` as a fill character. Line 39 displays them in binary format. The remaining two lines output each string of the `std::vector`. Finally, line 45 truncates each string to three characters.

```
[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`

When the `std::vector` becomes bigger, I want to add a linebreak. For this use case, I extended the syntax of the format specification.

Layouting the elements of a `std::vector`

```
// formatVectorLinebreak.cpp

#include <algorithm>
#include <iostream>
#include <limits>
#include <numeric>
#include <fmt/format.h>
#include <string>
#include <vector>

```

[51] https://en.cppreference.com/w/cpp/utility/format/format_to

```
template <typename T>
struct fmt::formatter<std::vector<T>> {

  std::string systemFormatString;
  std::string userFormatString;
  int lineBreak{std::numeric_limits<int>::max()};

  auto constexpr parse(format_parse_context& ctx) {
    std::string startFormatString = "{:";
    std::string parseContext(std::begin(ctx), std::end(ctx));
    auto posCurly = parseContext.find_last_of("}");
    auto posTab = parseContext.find_last_of("|");
    if (posTab == std::string::npos) {
      systemFormatString = startFormatString + parseContext.substr(0, p\
osCurly + 1);
    }
    else {
      systemFormatString = startFormatString + parseContext.substr(0, p\
osTab) + "}";
      userFormatString = parseContext.substr(posTab + 1, posCurly - pos\
Tab - 1);
      lineBreak = std::stoi(userFormatString);
    }
    return std::begin(ctx) + posCurly;
  }

  template <typename FormatContext>
  auto format(const std::vector<T>& v, FormatContext& ctx) {
    auto out = ctx.out();
    auto vectorSize = v.size();
    if (vectorSize == 0) return fmt::format_to(out, "\n");
    for (int i = 1;  i < vectorSize + 1; ++i) {
      fmt::format_to(out, systemFormatString, v[i-1]);
      if ( (i % lineBreak) == 0 ) fmt::format_to(out, "\n");
    }
    return fmt::format_to(out, "\n" );
```

```
47   }
48
49 };
50
51 int main() {
52
53   std::vector<int> myInts(100);
54   std::iota(myInts.begin(), myInts.end(), 1);
55
56   std::cout << fmt::format("{:|20}", myInts);
57   std::cout << '\n';
58   std::cout << fmt::format("{: |20}", myInts);
59   std::cout << '\n';
60   std::cout << fmt::format("{:4d|20}", myInts);
61   std::cout << '\n';
62   std::cout << fmt::format("{:10b|8}", myInts);
63
64 }
```

Here is how it works. I support an optional `|` followed by a number to the format specification. The number tells if a line break should be introduced. I search for the optional `|` symbol and the closing curly brace `}`. For robustness reasons, I start in lines 21 and 22 from the end. Thanks to the index of the `|` symbol and the index of the `}`, I can create the strings `systemFormatString` and `useFormatString` (lines 24 to 29). The member function `format` uses the `systemFormatString` and applies it to each element of the vector. I make a line break when (`i % lineBreak == 0`) holds (line 41).

Line 53 displays 20 elements in a row and makes a line break. I can do better. The format specification `{: |20}` (line 55) puts a space before each number. Additionally, line 57 aligns each element to four characters. Finally, the last line displays 8 numbers per line, aligns each element to 8 characters, and displays them: `{:10b|8}`.

The screenshot shows the readable formated elements of the `std::vector`.

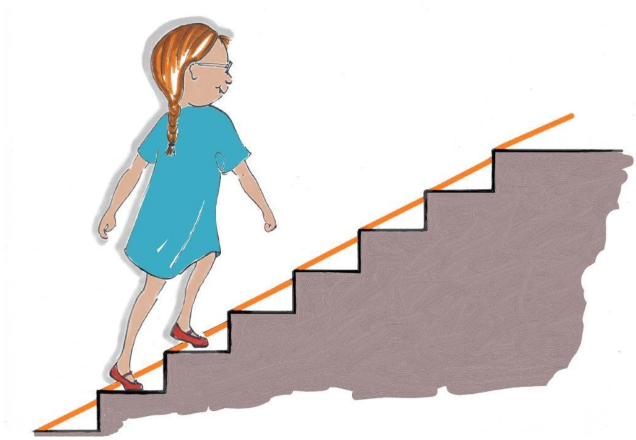
```
1234567891011121314151617181920
2122232425262728293031323334353637383940
4142434445464748495051525354555657585960
6162636465666768697071727374757677787980
81828384858687888990919293949596979899100


 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100


   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40
  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100


          1       10       11      100      101      110      111     1000
       1001     1010     1011     1100     1101     1110     1111    10000
      10001    10010    10011    10100    10101    10110    10111    11000
      11001    11010    11011    11100    11101    11110    11111   100000
     100001   100010   100011   100100   100101   100110   100111   101000
     101001   101010   101011   101100   101101   101110   101111   110000
     110001   110010   110011   110100   110101   110110   110111   111000
     111001   111010   111011   111100   111101   111110   111111  1000000
    1000001  1000010  1000011  1000100  1000101  1000110  1000111  1001000
    1001001  1001010  1001011  1001100  1001101  1001110  1001111  1010000
    1010001  1010010  1010011  1010100  1010101  1010110  1010111  1011000
    1011001  1011010  1011011  1011100  1011101  1011110  1011111  1100000
    1100001  1100010  1100011  1100100
```

Applying the format specification and a line break to the elements of a `std::vector`

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.

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`[52] and lambda expressions[53]. With C++20, we get `std::bind_front`[54]. This may make you wonder. To be pedantic `std::bind` is available since the Technical Report 1[55] (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.

[52] https://en.cppreference.com/w/cpp/utility/functional/bind
[53] https://en.cppreference.com/w/cpp/language/lambda
[54] https://en.cppreference.com/w/cpp/utility/functional/bind_front
[55] https://en.wikipedia.org/wiki/C%2B%2B_Technical_Report_1

The following program shows that you can replace `std::bind_front` with `std::bind` or lambda expressions.

Comparing `std::bind_front`, `std::bind`, and a lambda expression

```
// bindFront.cpp

#include <functional>
#include <iostream>

int plusFunction(int a, int b) {
    return a + b;
}

auto plusLambda = [](int a, int b) {
    return a + b;
};

int main() {

    std::cout << '\n';

    auto twoThousandPlus1 = std::bind_front(plusFunction, 2000);        \

    std::cout << "twoThousandPlus1(20): " << twoThousandPlus1(20) << '\
n';

    auto twoThousandPlus2 = std::bind_front(plusLambda, 2000);          \

    std::cout << "twoThousandPlus2(20): " << twoThousandPlus2(20) << '\
n';

    auto twoThousandPlus3 = std::bind_front(std::plus<int>(), 2000);
    std::cout << "twoThousandPlus3(20): " << twoThousandPlus3(20) << '\
n';

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

```
33
34     using namespace std::placeholders;
35
36     auto twoThousandPlus4 = std::bind(plusFunction, 2000, _1);         \
37
38     std::cout << "twoThousandPlus4(20): " << twoThousandPlus4(20) << '\\
39 n';
40
41     auto twoThousandPlus5 =  [](int b) { return plusLambda(2000, b); };\
42
43     std::cout << "twoThousandPlus5(20): " << twoThousandPlus5(20) << '\\
44 n';
45
46     std::cout << '\n';
47
48 }
```

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_evaluated` determines whether the function is executed at compile time or run time. Why do we need this function from the type-traits library? In C++20, we have roughly spoken of 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 have to 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 is executed at compile time or run time. 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;
}
```

How can we be sure that the function is executed at compile time? Essentially, there are three possibilities.

1. A constexpr function is executed at compile time:
   - The function is used in a so-called constant-evaluated context. A constant-evaluated context could be inside a `constexpr` function or a `static_assert`.
   - 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 `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, which was not declared as `constexpr` : `int a = 2000`.
3. A `constexpr` function can be executed at compile time or run time when neither rule 1 nor rule 2 applies. In this case, both options are valid and the decision is up to the compiler.

Exactly in point 3, the power of `std::is_constant_evaluated` kicks in. You can detect if the program runs at compile time or run time and perform different operations. cppreference.com/is_constant_evaluted[56] shows a smart use case. At compile time, you calculate the power of two numbers manually; 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) {
```

[56] https://en.cppreference.com/w/cpp/types/is_constant_evaluated

```
            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;
    double kilo2 = power(10.0, n);
    std::cout << "kilo2: " << kilo2 << '\n';

    std::cout << '\n';

}
```

There is one interesting observation I want to share. 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. Of course, the result of these calls is always `true` or `false`.

5.7.3 `std::source_location`

`std::source_location` represents information about the source code. This information includes file names, line numbers, and function names. The information is very valuable when you need information about the call site such as for debugging, logging, or testing purposes. The class `std::source_location` is the 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 call `std::source_location::current()` creates a new source location object `src` that represents the information of the call site. At the end of 2020, no C++ compiler supports `std::source_location`. Consequently, the following program `sourceLocation.cpp` is from cppreference.com/source_location[57].

Displaying information about the call site with `std::source_location`

```
// sourceLocation.cpp
// from cppreference.com

#include <iostream>
#include <string_view>
#include <source_location>

void log(std::string_view message,
         const std::source_location& location = std::source_location::c\
urrent())
{
```

[57] https://en.cppreference.com/w/cpp/utility/source_location

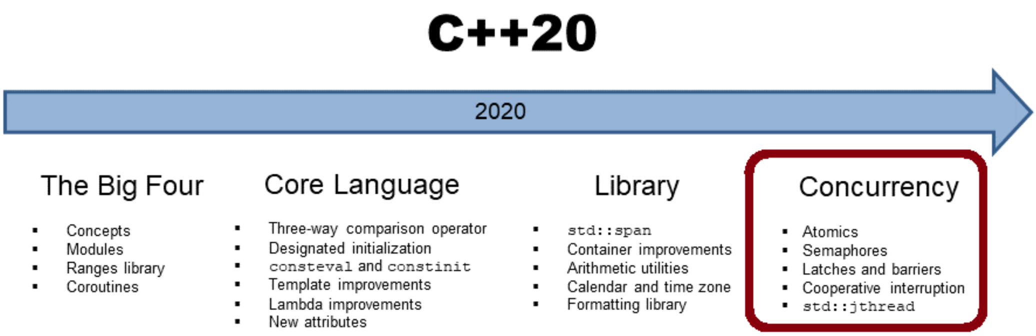
```
    std::cout << "info:"
              << location.file_name() << ':'
              << location.line() << ' '
              << message << '\n';
}

int main()
{
    log("Hello world!");  // info:main.cpp:19 Hello world!
}
```

The output of the program is part of its source code.

Distilled Information

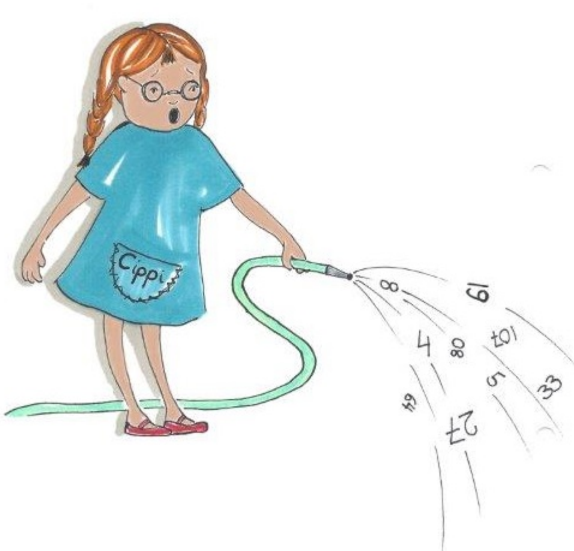
- `std::bind_front` is the easier-to-use variant for `std::bind` (C++11). In constrast to `std::bind`, `std::bind_front` does not enable the rearranging of its arguments.
- The function `std::is_constant_evaluted` determines whether the function is executed at compile time or run time.
- `std::source_location` 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.

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.

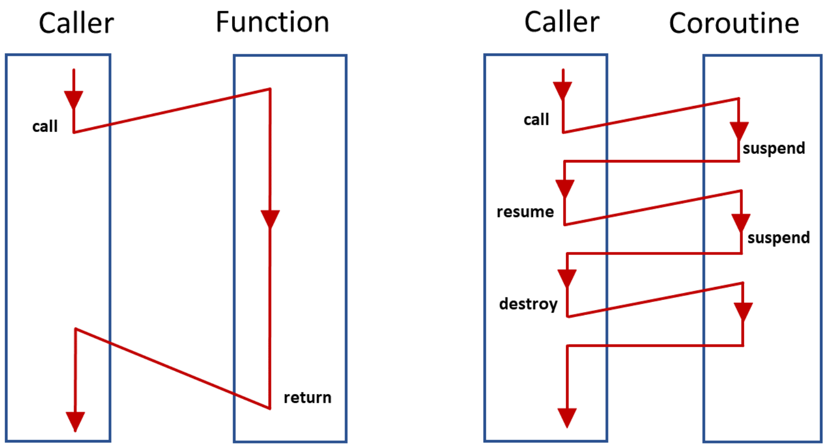
The Challenge of Understanding Coroutines

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", 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 provided examples should give you an initial intuition need for you to actually 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

[1] https://en.wikipedia.org/wiki/Melvin_Conway

construction in 1963. Donald Knuth[2] called procedures a special case of coroutines. Sometimes, it just takes a while to get your ideas accepted.



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 `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 pick 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

[2] https://en.wikipedia.org/wiki/Donald_Knuth

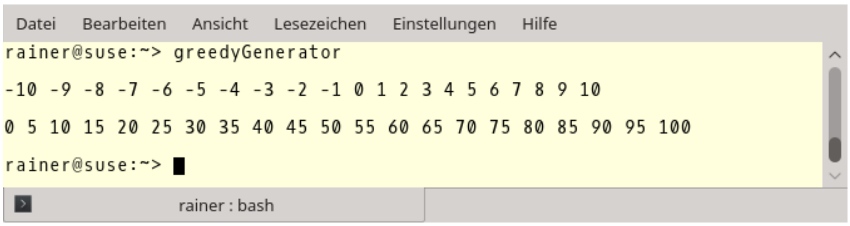
end, and inc has to be positive.

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 am reinventing the wheel with getNumbers, because that job could be done with `std::iota`[3].

For completeness, here is the output.



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 quite 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

```cpp
// lazyGenerator.cpp

#include <iostream>

generator<int> generatorForNumbers(int begin, int inc = 1) {

    for (int i = begin;; i += inc) {
        co_yield i;
    }

}

int main() {

    std::cout << '\n';

```

[3] http://en.cppreference.com/w/cpp/algorithm/iota

```
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 its 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

[4] https://en.wikipedia.org/wiki/Event-driven_programming

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.

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

[5]https://en.wikipedia.org/wiki/Computer_multitasking
[6]https://isocpp.org/files/papers/N4402.pdf

- 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.

[7] https://tour.golang.org/concurrency/1

Distinguish Between the Coroutine Factory and the Coroutine Object

The term coroutine is often used for two different aspects of coroutines: the function invoking `co_return`, `co_await`, or `co_yield`, and the coroutine object. Using one term for two different coroutine aspects 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 a coroutine factory that returns a coroutine object. The coroutine object is a resumable object 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 holds 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.

[8] https://en.cppreference.com/w/cpp/language/variadic_arguments

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.

6.1.3.1 Promise Object

The promise object is manipulated from inside the coroutine, and it delivers its result or exception via the promise object.

The promise object must support the following interface.

Promise object

Member Function	Description
Default constructor	A promise must be default constructible.
`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 exception happens.
`get_return_object()`	Returns the coroutine object (resumable object).
`return_value(val)`	Is invoked by `co_return val`.
`return_void()`	Is invoked by `co_return`.
`yield_value(val)`	Is invoked by `co_yield val`.

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` returns a resumable object that the client uses to interact with the coroutine. A promise needs at least one of the member functions `return_value`, `return_void`, or `yield_value`. You don't need to define the member functions `return_value` or `return_void` if your coroutine never ends.

The three functions `yield_value`, `initial_suspend`, and `final_suspend` return awaitables. An Awaitable is something that you can await on. The awaitable determines if the coroutine pauses or not.

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 code snippet shows a simple `Generator` having a coroutine handle `coro`.

A coroutine handle

```
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();
    }
    T getValue() {
        return coro.promise().current_value;
    }
    bool next() {
        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>`[9]. The member functions `next` (line 16) and `getValue` (line 13) allow a client to resume the promise (`gen.next()`) or ask for its value (`gen.getValue()`) using the coroutine handle.

Invoking a coroutine

```
Generator<int> coroutineFactory(); // function that returns a coroutine\
 object

auto gen = coroutineFactory();
gen.next();
auto result = gen.getValue();
```

Internally, both functions trigger the coroutine handle `coro` (line 8) to

- resume the coroutine: `coro.resume()` (line 17) or `coro()`;
- 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. The call `coro` only returns `true` at its final suspension point.

The resumable object requires an inner type `promise_type`

A resumable object such as `Generator` must have an inner type `promise_type`. Alternatively, you can specialize `std::coroutine_traits`[10] on `Generator` and define a public member `promise_type` in it: `std::coroutine_traits<Generator>`.

[9] https://en.cppreference.com/w/cpp/coroutine/coroutine_handle
[10] https://en.cppreference.com/w/cpp/coroutine/coroutine_traits

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 exceed the current suspension point.

Two requirements are necessary to optimize out the allocation of the coroutine:

1. The lifetime of the coroutine has to be nested inside the lifetime of the caller.
2. The caller of the coroutine knows the size of the coroutine frame.

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. The awaitable determines if the coroutine pauses or not.

Essentially, the compiler generated the three function calls using the promise `prom` and the `co_await` operator.

Compiler-generated function calls

Call	Compiler generated call
`yield value`	`co_await prom.yield_value(value)`
`prom.initial_suspend()`	`co_await prom.initial_suspend()`
`prom.final_suspend()`	`co_await prom.final_suspend()`

The `co_await` operator needs an awaitable as argument. Awaitables have to implement the concept Awaitable.

6.1.4.2 The Concept Awaitable

The concept Awaitable requires three functions.

The concept Awaitable

Function	Description
`await_ready`	Indicates if the result is ready. When it returns `false`, `await_suspend` is called.
`await_suspend`	Schedule the coroutine for resumption or destruction.
`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`.

6.1.4.3 `std::suspend_always` and `std::suspend_never`

As its name suggests, the Awaitable `suspend_always` always suspends. Therefore, the call `await_ready` returns `false`.

The Awaitable `std::suspend_always`

```
struct suspend_always {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(std::coroutine_handle<>) const noexcep\
t {}
    constexpr void await_resume() const noexcept {}
};
```

The opposite holds for `suspend_never`. It never suspends and, hence, the call `await_ready` returns `true`.

The Awaitable `std::suspend_never`

```
struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(std::coroutine_handle<>) const noexcep\
t {}
    constexpr void await_resume() const noexcept {}
};
```

The awaitables `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 exected: `initial_suspend` at the beginning and `final_suspend` at the end end of the coroutine.

6.1.4.4 `initial_suspend`

When the member function `initial_suspend` returns `std::suspend_always`, the coroutine suspends at its beginning. When returning `std::suspend_never`, the coroutine does not pause.

- A lazy coroutine that pauses immediately

A lazy coroutine

```
std::suspend_always initial_suspend() {
    return {};
}
```

- An eager coroutine that runs immediately

A eager coroutine

```
std::suspend_never initial_suspend() {
    return {};
}
```

6.1.4.5 `final_suspend`

When the member function `final_suspend` returns `std::suspend_always`, the coroutine suspends at its end. When returning `std::suspend_never`, the coroutine does not pause.

- A lazy coroutine that pauses at its end

A lazy coroutine that finally pauses

```
std::suspend_always final_suspend noexcept noexcept noexcept noexcept()\
 {
    return {};
}
```

- An eager coroutine that doesn't pause at its end

A eager coroutine that doesn't pause

```
std::suspend_never final_suspend() noexcept {
    return {};
}
```

So far, we have only Awaitables, but we need something to await for. Let me fill the gap and write about Awaiters.

6.1.4.6 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 an Awaitable. Further, an `expression` is a call on the promise object (Awaitable): `prom.yield_-value(value)`, `prom.initial_suspend()`, or `prom.final_suspend()`. For readability, I rename in the following lines promise object `prom` to `awaitable`.

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_wait` operator and returns an Awaiter:
   ```
   awaiter = operator co_await();
   ```
3. If there is no `co_wait` 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_wait>
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_awaits` 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_awaits` its result
- Coroutine is destroyed (by terminating via `co_return` an uncaught exception, or via the coroutine handle)
  - calls the destruction 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_awaits` 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 `awaitable`. For simplicity, I ignore exception handling and describe the workflow with comments.

The generated Awaiter Workflow

```
1 awaitable.await_ready() returns false:
2
3     suspend coroutine
4
5     awaitable.await_suspend(coroutineHandle) returns:
6
7         void:
8             awaitable.await_suspend(coroutineHandle);
9             coroutine keeps suspended
10            return to caller
11
12        bool:
13            bool result = awaitable.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 = awaitable.await_suspend(corou\
22 tineHandle);
23            anotherCoroutineHandle.resume();
24            return to caller
25
26 resumptionPoint:
27
28 return awaitable.await_resume();
```

The workflow is only executed if `awaitable.await_ready()` returns `false` (line 1). In case it returns `true`, the coroutine is ready and returns with the result of the call `awaitable.await_resume()` (line 27).

Let me assume that `awaitable.await_ready()` returns `false`. First, the coroutine is suspended (line 3), and immediately the return value of `awaitable.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 `awaitable.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, nor are the calls `await_suspend` or `await_resume` 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 that still does something meaningful: it automatically stores the result of its invocation.

An eager future

```
// 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);
        }
    };
};
```

```
36
37 MyFuture<int> createFuture() {
38     co_return 2021;
39 }
40
41 int main() {
42
43     std::cout << '\n';
44
45     auto fut = createFuture();
46     std::cout << "fut.get(): " << fut.get() << '\n';
47
48     std::cout << '\n';
49
50 }
```

`MyFuture` behaves as a future[11], 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

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

You may think that 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 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 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`.

An infinite data stream

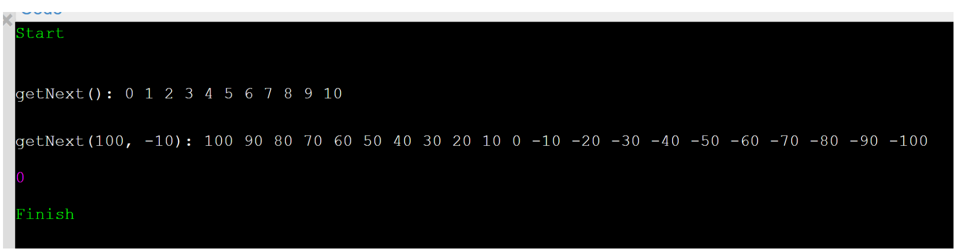
```
// infiniteDataStream.cpp

#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 {
            return std::suspend_always{};
        }
        auto get_return_object() {                                  // (2)
            return Generator{handle_type::from_promise(*this)};
        }
        auto return_void() {
```

```
            return std::suspend_never{};
        }

        auto yield_value(const T value) {                                   // (6)
            current_value = value;
            return std::suspend_always{};
        }
        void unhandled_exception() {
            std::exit(1);
        }
        T current_value;
    };

};

Generator<int> getNext(int start = 0, int step = 1) {
    auto value = start;
    while (true) {
        co_yield value;
        value += step;
    }
}

int main() {

    std::cout << '\n';

    std::cout << "getNext():";
    auto gen = getNext();
    for (int i = 0; i <= 10; ++i) {
        gen.next();
        std::cout << " " << gen.getValue();                                // (7)
    }

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

```

```
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[12], 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.

[12] https://wandbox.org/

7. gets the next value

In additional 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.

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

```
Acceptor acceptor{443};
while (true) {
    Socket socket = co_await acceptor.accept();
    auto request = co_await socket.read();
    auto response = handleRequest(request);
    co_await socket.write(response);
}
```

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

```
// startJob.cpp

#include <coroutine>
#include <iostream>

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();
    }

```

```
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 always 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[13], promises and futures[14], and also an atomic boolean[15] 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.

[13] https://en.cppreference.com/w/cpp/thread/condition_variable
[14] https://en.cppreference.com/w/cpp/thread
[15] https://en.cppreference.com/w/cpp/atomic/atomic

Thread Synchronization

```
// senderReceiver.cpp

#include <coroutine>
#include <chrono>
#include <iostream>
#include <functional>
#include <string>
#include <stdexcept>
#include <atomic>
#include <thread>

class Event {
 public:

    Event() = default;

    Event(const Event&) = delete;
    Event(Event&&) = delete;
    Event& operator=(const Event&) = delete;
    Event& operator=(Event&&) = delete;

    class Awaiter;
    Awaiter operator co_await() const noexcept;

    void notify() noexcept;

 private:

    friend class Awaiter;

    mutable std::atomic<void*> suspendedWaiter{nullptr};
    mutable std::atomic<bool> notified{false};

};

```

```
class Event::Awaiter {
 public:
  Awaiter(const Event& eve): event(eve) {}

  bool await_ready() const;
  bool await_suspend(std::coroutine_handle<> corHandle) noexcept;
  void await_resume() noexcept {}

 private:
    friend class Event;

    const Event& event;
    std::coroutine_handle<> coroutineHandle;
};

bool Event::Awaiter::await_ready() const {

    // allow at most one waiter
    if (event.suspendedWaiter.load() != nullptr){
        throw std::runtime_error("More than one waiter is not valid");
    }

    // event.notified == false; suspends the coroutine
    // event.notified == true; the coroutine is executed like a normal \
function
    return event.notified;
}

bool Event::Awaiter::await_suspend(std::coroutine_handle<> corHandle) n\
oexcept {

    coroutineHandle = corHandle;

    if (event.notified) return false;

    // store the waiter for later notification
```

```
    event.suspendedWaiter.store(this);

    return true;
}

void Event::notify() noexcept {
    notified = true;

    // try to load the waiter
    auto* waiter = static_cast<Awaiter*>(suspendedWaiter.load());

    // check if a waiter is available
    if (waiter != nullptr) {
        // resume the coroutine => await_resume
        waiter->coroutineHandle.resume();
    }
}

Event::Awaiter Event::operator co_await() const noexcept {
    return Awaiter{ *this };
}

struct Task {
    struct promise_type {
        Task get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() {}
    };
};

Task receiver(Event& event) {
    auto start = std::chrono::high_resolution_clock::now();
    co_await event;
    std::cout << "Got the notification! " << '\n';
```

```
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Waited " << elapsed.count() << " seconds." << '\n';
}

using namespace std::chrono_literals;

int main() {

    std::cout << '\n';

    std::cout << "Notification before waiting" << '\n';
    Event event1{};
    auto senderThread1 = std::thread([&event1]{ event1.notify(); });  /\
/ Notification
    auto receiverThread1 = std::thread(receiver, std::ref(event1));

    receiverThread1.join();
    senderThread1.join();

    std::cout << '\n';

    std::cout << "Notification after 2 seconds waiting" << '\n';
    Event event2{};
    auto receiverThread2 = std::thread(receiver, std::ref(event2));
    auto senderThread2 = std::thread([&event2]{
      std::this_thread::sleep_for(2s);
      event2.notify();                                              //\
 Notification
    });

    receiverThread2.join();
    senderThread2.join();

    std::cout << '\n';

```

```
144 }
```

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 119) and `senderThread2` (line 130) each uses an event to send its notification,respectively, in lines 119 and 132. The function `receiver` in lines 102 - 109 is the coroutine, which is executed in threads `receiverThread1` (line 122) and `receiverThread2` (line 135). 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.

```
Start


Notification before waiting
Got the notification!
Waited 1.5738e-05 seconds.

Notification after 2 seconds waiting
Got the notification!
Waited 2.00019 seconds.


0

Finish
```

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 119) before the coroutine is suspended, but the `event2` sends its notification after a time duration of 2 seconds (line 132).

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.

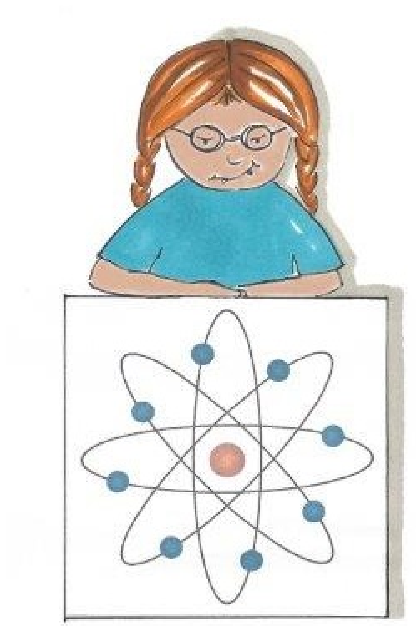
Let's first look at `event1` and the first workflow. Here, `event1` sends its notification before `receiverThread1` is started. The invocation `event1` (line 118) triggers the method `notify` (lines 75 to 86). First the notification flag is set and then, the call `static_cast<Awaiter*>(suspendedWaiter.load());` 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 84 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_wait 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 - 73) is executed. `await_suspend` gets the coroutine handle `corHandle` and stores it for later invocation in the variable `coroutineHandle` (line 65). 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 75) 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 ensures that there is no data race. The lifetime of the referenced object must exceed the lifetime of the `atomic_ref`. When any `atomic_ref` is accessing 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`. The `counter` is concurrently incremented by a few threads. Consequently, `counter` has to be protected.

Using an atomic reference

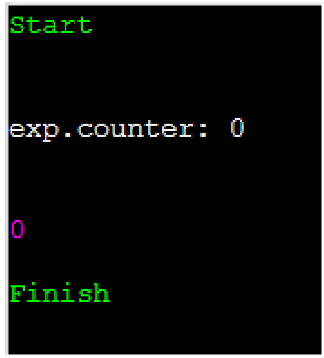
```
// atomicReference.cpp

#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) {
```

```
28         v.emplace_back([&counter] {
29             auto randomNumber = getRandom(100, 200);
30             for (int i = 0; i < randomNumber; ++i) { ++counter; }
31         });
32     }
33 
34     for (auto& t : v) t.join();
35 
36 }
37 
38 int main() {
39 
40     std::cout << '\n';
41 
42     ExpensiveToCopy exp;
43     count(exp);
44     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 10 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[16] 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[17] gives me a surprising result.

[16] https://en.wikipedia.org/wiki/Mersenne_Twister
[17] https://wandbox.org/



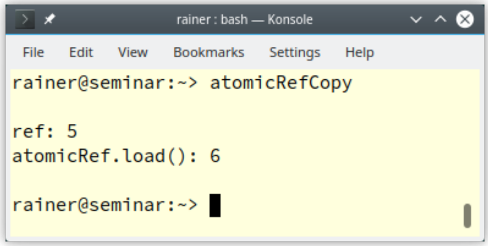
Surprise with an atomic reference

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.

Copying the reference

```
// atomicRefCopy.cpp

#include <atomic>
#include <iostream>

int main() {

    std::cout << '\n';

    int val{5};
    int& ref = val;
    std::atomic<int> atomicRef(ref);
    ++atomicRef;
    std::cout << "ref: " << ref << '\n';
    std::cout << "atomicRef.load(): " << atomicRef.load() << '\n';

    std::cout << '\n';

}
```

The increment operation in line 13 does not address the reference `ref` (line 11). The value of `ref` is not changed.



Copying the reference

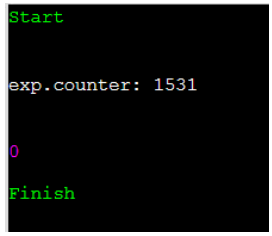
Replacing the `std::atomic<int>` with `std::atomic_ref<int>` solves the issue.

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`[18], 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[19] 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*>`.

[18] https://en.cppreference.com/w/cpp/atomic/atomic
[19] 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>`[20]:
  - `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`.

[20] http://en.cppreference.com/w/cpp/header/cstdint

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()`.
`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`	Unblocks one atomic wait operation.
`notify_all`	Unblocks all atomic wait operations.
`wait`	Blocks until it is notified.

All operations on `atomic_ref`

Function	Description
	Compares itself with the `old` value to protect against spurious wakeups and lost 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, 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, otherwise `false`. 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[21].

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`.

[21] https://en.cppreference.com/w/cpp/atomic/memory_order

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`[22] 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

[22]https://en.cppreference.com/w/cpp/memory/shared_ptr

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.

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, use of 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[23] for atomic smart pointers directly addresses the deficiencies of the current implementation. The deficiencies 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**: the use of the 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 big advantage compared to the free `atomic_*` functions. The atomic versions are designed for the special use case and can internally have a `std::atomic_flag` as a kind of cheap spinlock[24]. Designing the non-atomic versions of the pointer functions to be thread-safe would be overkill where 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

[23] http://wg21.link/n4162
[24] https://en.wikipedia.org/wiki/Spinlock

insertion, deletion, and searching of elements. This singly-linked list is implemented in a lock-free way.

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 a lot 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[25] proposed the new types `std::atomic_shared_ptr` and `std::atomic_-weak_ptr` as atomic smart pointers. By merging them in the mainline ISO C++ standard, they became partial template specialization of `std::atomic`, namely `std::atomic<std::sha` `ptr<T>>`, and `std::atomic<std::weak_ptr<T>>`.

The following program shows five thread modifying a `std::atomic<std::shared_-ptr<std::string>>` withoud 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"));
    });
```

[25] http://wg21.link/n4162

```
24     std::thread t3([&sharString]{
25         sharString.store(std::make_shared<std::string>(*sharString.load\
26 () +"Three"));
27     });
28     std::thread t4([&sharString]{
29         sharString.store(std::make_shared<std::string>(*sharString.load\
30 () +"Four"));
31     });
32     std::thread t5([&sharString]{
33         sharString.store(std::make_shared<std::string>(*sharString.load\
34 () +"Five"));
35     });
36 
37     t1.join();
38     t2.join();
39     t3.join();
40     t4.join();
41     t5.join();
42 
43     std::cout << *sharString.load() << '\n';
44 
45 }
```

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 modifying 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`[26] 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. `ATOMIC_FLAG_INIT` enables initializing the `std::atomic_flag` to `false`.

`std::atomic_flag` has two exciting properties, it is

- the only lock-free atomic.

[26] https://www.modernescpp.com/index.php/the-atomic-flag

- 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 changing it. This changes with C++20.

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 then if 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.

As compared to C++11, 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 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.

6.2.3.3 One Time Synchronization of Threads

Sender-receiver workflows are quite 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 provide a short refresher.

6.2.3.3.1 Condition Variables

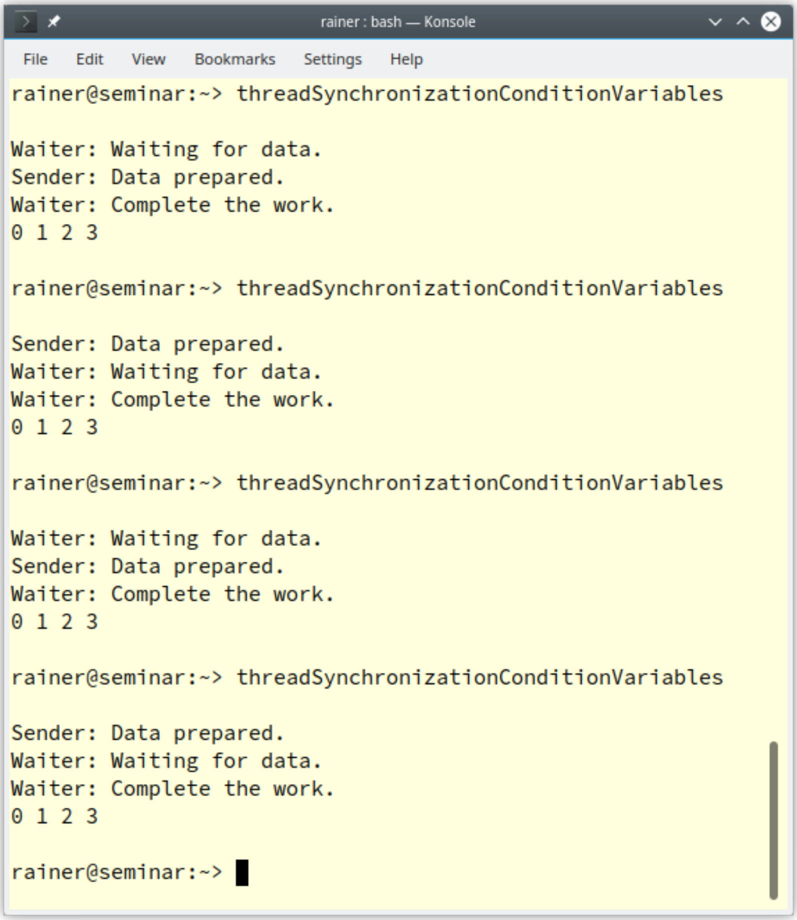
A condition variable can fulfill the role of a sender or a receiver. As a sender, it can notify one or more receivers.

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';

    std::thread t1(prepareWork);
    std::thread t2(completeWork);

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

    std::cout << '\n';

}
```

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 spurious wakeup and the second as 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 output of the program, 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"[27].

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';
```

[27] https://www.modernescpp.com/index.php/c-core-guidelines-be-aware-of-the-traps-of-condition-variables

```
24     for (auto i: myVec) std::cout << i << " ";
25     std::cout << '\n';
26 
27 }
28 
29 int main() {
30 
31     std::cout << '\n';
32 
33     std::promise<void> sendNotification;
34     auto waitForNotification = sendNotification.get_future();
35 
36     std::thread t1(prepareWork, std::move(sendNotification));
37     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[28].

6.2.3.3.3 `std::atomic_flag`

Now, I jump directly from C++11 to C++20.

[28] https://www.modernescpp.com/index.php/tag/tasks

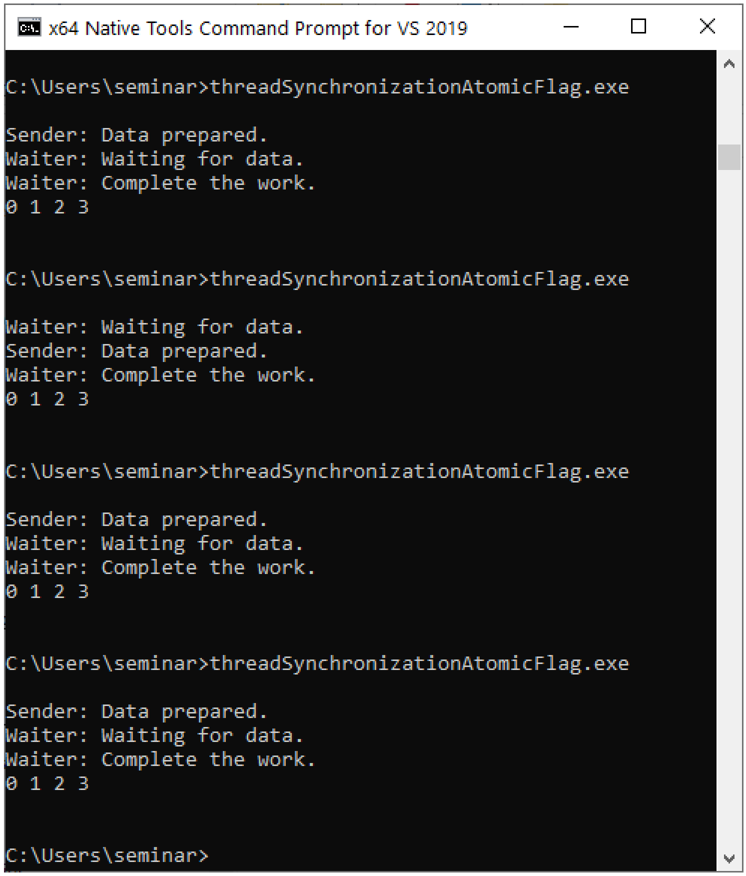
Thread synchronization with a `std::atomic_flag`

```cpp
// threadSynchronizationAtomicFlag.cpp

#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> myVec{};

std::atomic_flag atomicFlag{};

void prepareWork() {

    myVec.insert(myVec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared."  << '\n';
    atomicFlag.test_and_set();
    atomicFlag.notify_one();

}

void completeWork() {

    std::cout << "Waiter: Waiting for data." << '\n';
    atomicFlag.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';

```

```
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 completing the work waits 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`[29] 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 is available on all partial and full specializations of `std::atomic` (bools, integrals, floats and pointers) and `std::atomic_ref`.

[29] https://en.cppreference.com/w/cpp/atomic/atomic

Thanks to `atomic<bool>`, the previous program `threadSynchronizationAtomicFlag.cpp` can directly be reimplemented.

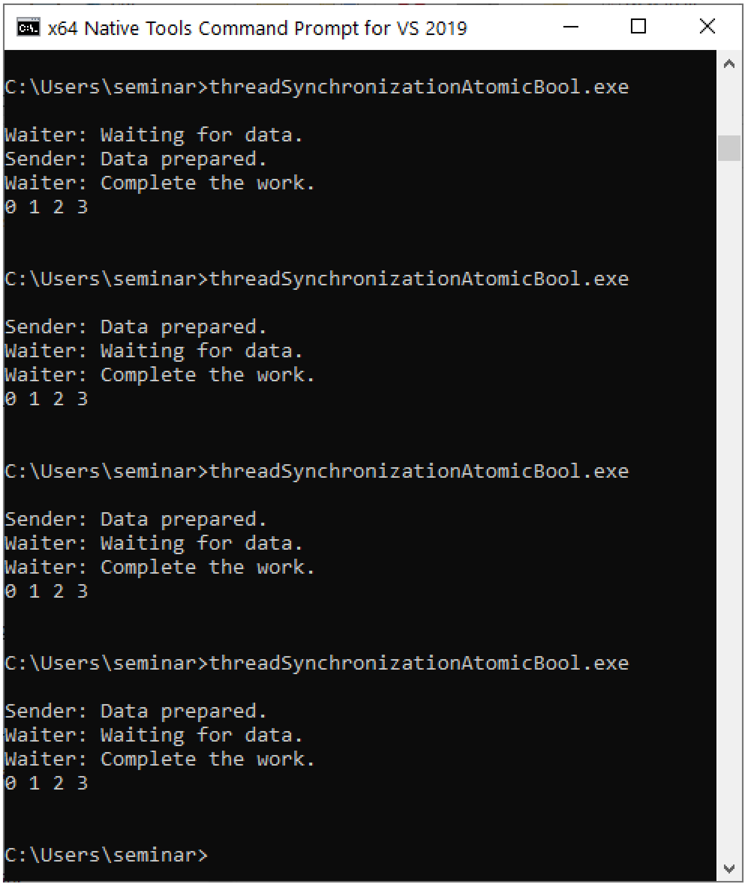
Thread synchronization with `std::atomic<bool>`

```cpp
// 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() {
```

```
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 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 is only one downside to using promises and futures: 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 of them 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'm only sure that I would not use a condition variable if possible.

Distilled Information

- `std::atomic_ref` applies atomic operations to the referenced object. Concurrent writing and reading is 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 you can use it for thread synchronization.

6.3 Semaphores

Cippi directs the train

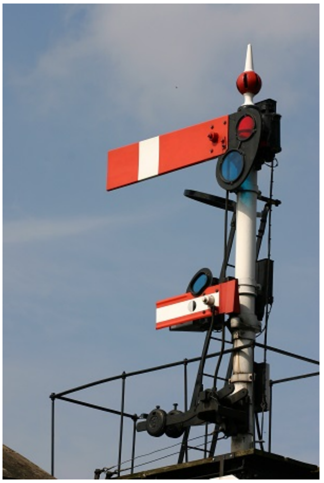
Semaphores are a synchronization mechanism used to control concurrent access to a shared resource. A counting semaphore is a special semaphore that has a counter that is bigger than zero. The counter is initialized in the constructor. Acquiring the semaphore decreases the counter and releasing the semaphore increases the counter. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

Edsger W. Dijkstra invented Semaphores

The Dutch computer scientist Edsger W. Dijkstra[30] 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[31].

Originally, a semaphore was a railway signal.

[30] https://en.wikipedia.org/wiki/Edsger_W._Dijkstra
[31] 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,[32]

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[33].

```
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 acquire and release of a semaphore call can happen on different threads. The following table presents the interface of a `std::counting_semaphore`.

[32] https://commons.wikimedia.org/w/index.php?curid=1972304
[33] https://en.cppreference.com/w/cpp/named_req/BasicLockable

Member functions of a `std::counting_semaphore sem`

Member function	Description
`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.
`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 an at least maximal value of 10 and a counter of 5. The call `sem.max()` returns the maximum possible value of the internal counter. The following realations 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](#chapterXXXTimeSSSDuration);` `the member function` sem.try_acquire_until(absTime) `needs a [time point](#chapterXXXTimeS` `The three calls` sem.try_acquire`,` sem.try_acquire_for`, and` sem.try_acquire_until‘ return a boolean indicating the success of the calls.

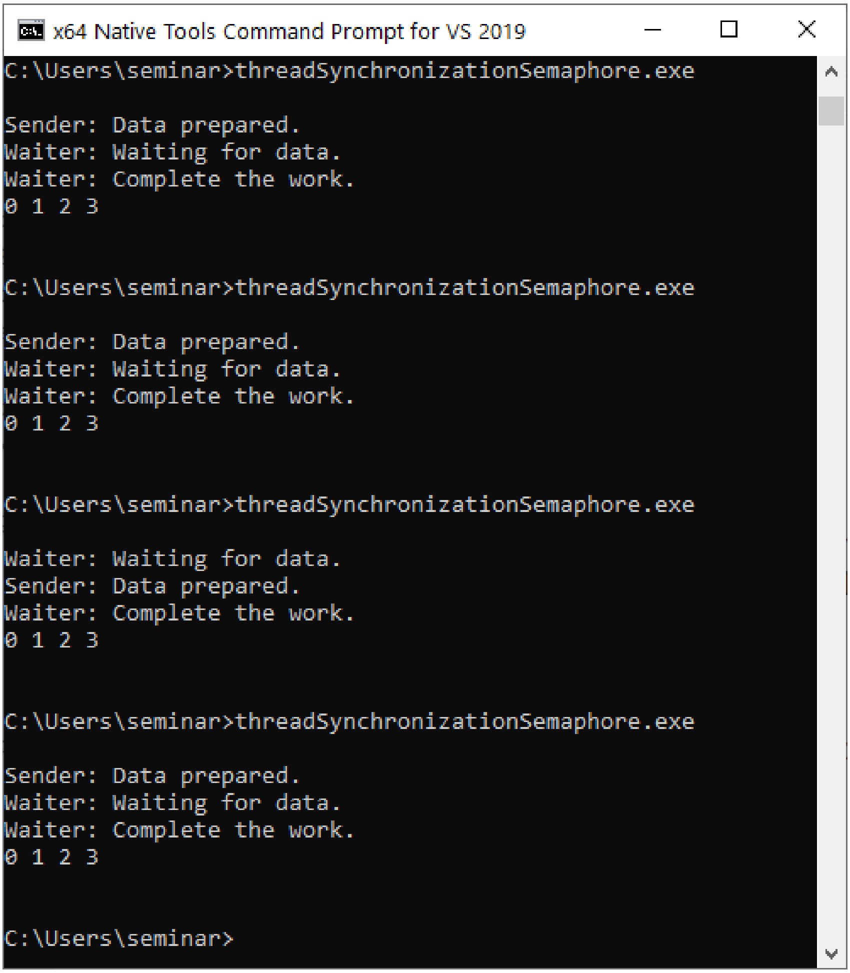
Semaphores are typically used in sender-receiver workflows. For example, initializing the semaphore `sem` with 0 will block the receiver's `sem.acquire()` call until the sender calls `sem.release()`. Consequently, the receiver waits for the notification of the sender. One-time synchronization of threads can easily be implemented using semaphores.

Thread synchronization with a `std::counting_semaphore`

```
// threadSynchronizationSemaphore.cpp

#include <iostream>
#include <semaphore>
#include <thread>
#include <vector>

std::vector<int> myVec{};

std::counting_semaphore<1> prepareSignal(0);

void prepareWork() {

    myVec.insert(myVec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared."  << '\n';
    prepareSignal.release();
}

void completeWork() {

    std::cout << "Waiter: Waiting for data." << '\n';
    prepareSignal.acquire();
    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);
```

```
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 will block 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 a lot 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
`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`. When `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.

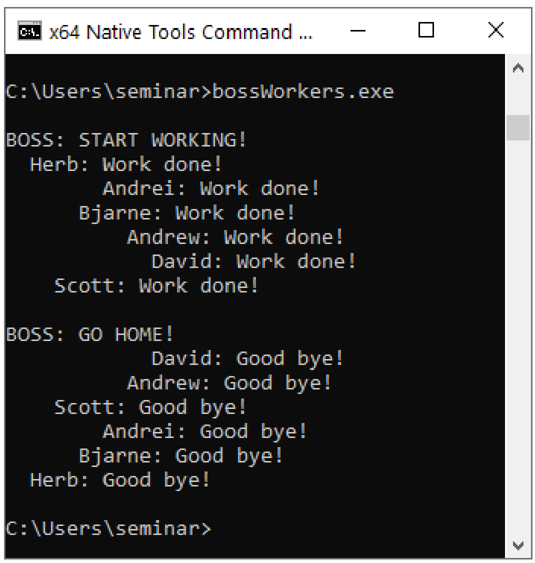
A boss-worker workflow using two `std::latch`

```
// bossWorkers.cpp

#include <iostream>
#include <mutex>
#include <latch>
#include <thread>

std::latch workDone(6);
std::latch goHome(1);

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() (){
          // notify the boss when work is done
          synchronizedOut(name + ": " + "Work done!\n");
          workDone.count_down();

          // waiting before going home
          goHome.wait();
          synchronizedOut(name + ": " + "Good bye!\n");
      }
 private:
    std::string name;
};

int main() {
```

```

    std::cout << '\n';

    std::cout << "BOSS: START WORKING! " << '\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);

    Worker andrew("          Andrew");
    std::thread andrewWork(andrew);

    Worker david("            David");
    std::thread davidWork(david);

    workDone.wait();

    std::cout << '\n';

    goHome.count_down();

    std::cout << "BOSS: GO HOME!" << '\n';

    herbWork.join();
    scottWork.join();
    bjarneWork.join();
    andreiWork.join();
    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 fulfill 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`

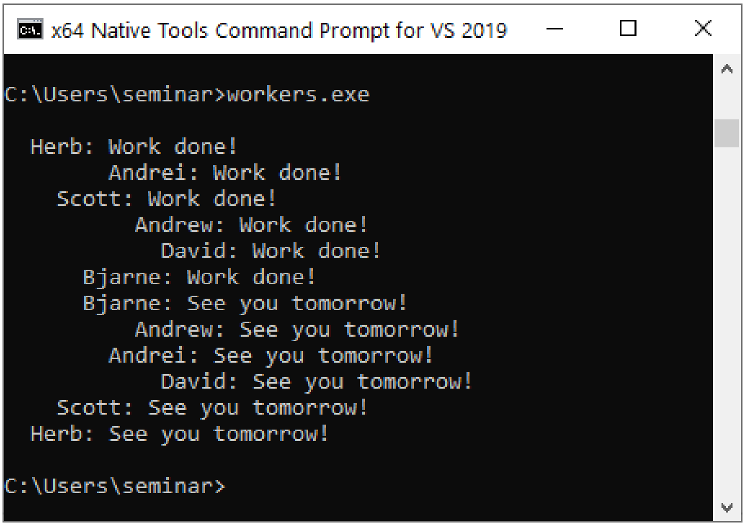
When you think about this workflow, you may notice that it can be done without a boss. Here it is.

A worker's workflow using a `std::latch`

```cpp
// 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);

```

```
36      Worker scott("    Scott");
37      std::thread scottWork(scott);
38
39      Worker bjarne("      Bjarne");
40      std::thread bjarneWork(bjarne);
41
42      Worker andrei("        Andrei");
43      std::thread andreiWork(andrei);
44
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();`. This means 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](#glossaryXXXCallableSSSUnitZZZcall is invoked. The` std::barrier‘ gets its callable in its constructor.

The completion step performs the following steps:

1. All threads are blocked.
2. An arbitrary thread is unblocked and executes the callable.
3. If the completion step is done, all threads are unblocked.

Member functions of a `std::barrier bar`

Member function	Description
`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 `wait(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.

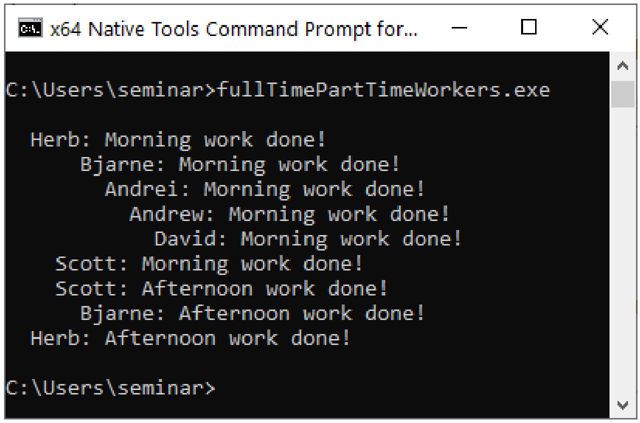
Full-time and part-time workers

```
// 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 do\
ne

    }
 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() {

    std::cout << '\n';

    FullTimeWorker herb("  Herb");
    std::thread herbWork(herb);

    FullTimeWorker scott("    Scott");
    std::thread scottWork(scott);
```

```
54
55     FullTimeWorker bjarne("      Bjarne");
56     std::thread bjarneWork(bjarne);
57
58     PartTimeWorker andrei("        Andrei");
59     std::thread andreiWork(andrei);
60
61     PartTimeWorker andrew("          Andrew");
62     std::thread andrewWork(andrew);
63
64     PartTimeWorker david("            David");
65     std::thread davidWork(david);
66
67     herbWork.join();
68     scottWork.join();
69     bjarneWork.join();
70     andreiWork.join();
71     andrewWork.join();
72     davidWork.join();
73
74 }
```

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, 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 manage repeated tasks by multiple threads.

6.5 Cooperative Interruption

Cippi stops in front of the stop sign

The additional functionality of the cooperative interruption thread is based on the `std::stop_source`, `std::stop_token`, and the `std::stop_callback` classes. `std::jthread` and `std::condition_variable_any` support cooperative interruption.

First, why it is 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 having 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 afterward be used to actively poll the token for a stop request or to register a callback via `std::stop_callback`. The stop request is sent by a `std::stop_source`. 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.

In the next subsecions, I provide more details about 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) constructs a `std::stop_source` with a new stop state. The constructor taking `std::nostopstate_t` (line 2) constructs an empty `std::stop_source` without 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
`src.get_token()`	If `stop_possible()`, returns a `stop_token` for the associated stop state. Otherwise, returns a default-constructed (empty) `stop_token`.
`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.

Member functions of `std::stop_source src`

Member function	Description
`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`.

The call `src.request_stop()` is visible to all `std::stop_source` and `std::stop_token` of the same associated stop state. Also, any registered callbacks for the associated `std::stop_token` and any `std::condiction_variable_any` waiting on the associated `std::stop_token() will be awoken. When a stop is requested, it cannot be withdrawn.` src.request_stop() `such as` src.stop_requested(), `and` src.stop_-possible()‘ is atomic.

`src.stop_requested()` returns `true` when `src` has an associated stop state and was not asked to stop earlier. `src.request_stop()` is successful and returns `true` if `src` has an associated stop state and it 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 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 to the interruptible waiting functions of `std::condition_variable_any`.

Member functions of `std::stop_token stoken`

Member function	Description
`stoken.stop_possible()`	Returns `true` if `stoken` has an associated stop state.
`stoken.stop_requested()`	`true` if `request_stop()` was called on the associated `std::stop_source src`, otherwise `false`.

`stoken.stop_possible()` also returns `true` if the stop request has already been made. A default-constructed `std::stop_token` that has no associated stop state.

`stoken.stop_requested()` returns `true` when the stop token has an associated stop state and has already received a stop request.

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

```
1 std::jthread jthr([](std::stop_token stoken) {
2     ...
3     std::stop_token interruptDisabled;
4     std::swap(stoken, interruptDisabled);
5     ...
6     std::swap(stoken, interruptDisabled);
7     ...
8 }
```

`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`

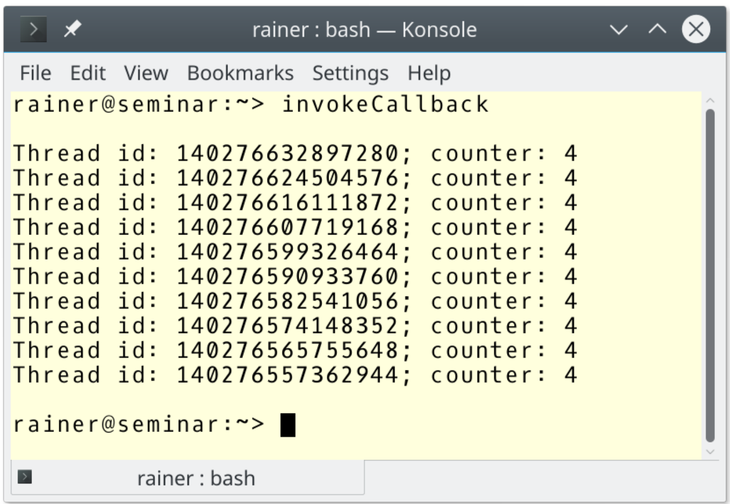
The following example shows the use of `std::stop_callback`.

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) {
        std::atomic<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() {

    std::cout << '\n';

    std::vector<std::jthread> vecThreads(10);
    for(auto& thr: vecThreads) thr = std::jthread(func);

    std::this_thread::sleep_for(1s);

    for(auto& thr: vecThreads) thr.request_stop();

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

```
36
37 }
```

Each of the ten threads invokes the lambda function `func` (lines 11 - 22). The callback in lines 14 - 17 displays the thread id and the counter. Due to the 1-second sleeping of the main thread and the sleeping of the child threads, the counter is four when the callbacks are invoked. The call `thr.request_stop()` triggers the callback on each thread.



Use of callbacks

6.5.3.1 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.

6.5.3.2 New `wait` Overloads for the `condition_variable_any`

`std::condition_variable_any` is a generalization of `std::condition_variable`[34]. `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`.

Three new wait overloads

```cpp
template <class Predicate>
bool wait(Lock& lock,
          stop_token stoken,
          Predicate pred);

template <class Rep, class Period, class Predicate>
bool wait_for(Lock& lock,
              stop_token stoken,
              const chrono::duration<Rep, Period>& rel_time,
              Predicate pred);

template <class Clock, class Duration, class Predicate>
bool wait_until(Lock& lock,
```

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

```
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`. This returned boolean is independent of whether a stop was requested or whether 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) retu\
rn 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{
```

```
27          std::cout << "Stop request received" << '\n';
28      }
29  }
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 triggered a stop request in line 50. The output of the program 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 afterward be used to poll the token for a stop request actively or register a callback via `std::stop_callback`.
- 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`[35] from C++11, `std::jthread` automatically joins in its destructor and can cooperatively be interrupted.

The following table gives you a concise overview of the `std::jthread t` functionality. For additional details, please refer to cppreference.com[36].

[35] https://en.cppreference.com/w/cpp/thread/thread
[36] https://en.cppreference.com/w/cpp/thread/jthread

Functions of a `std::jthread` `t`

Method	Description
`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)` and `std::swap(t1, t2)`	Swaps the threads.
`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.

6.6.1 Automatically Joining

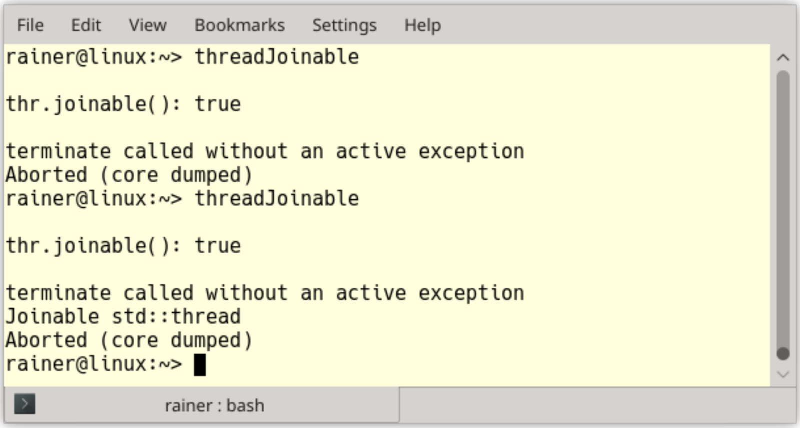
This is the *non-intuitive* behavior of `std::thread`. If a `std::thread` is still joinable, `std::terminate`[37] is called in its destructor. A thread `thr` is joinable if neither `thr.join()` nor `thr.detach()` was called.

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.

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



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".

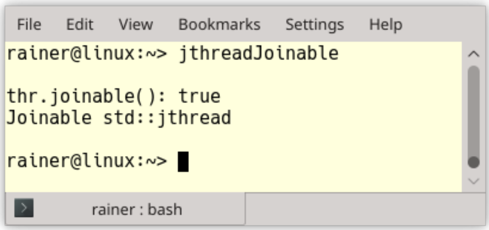
In the next example, I use `std::jthread` from the current C++20 standard.

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`

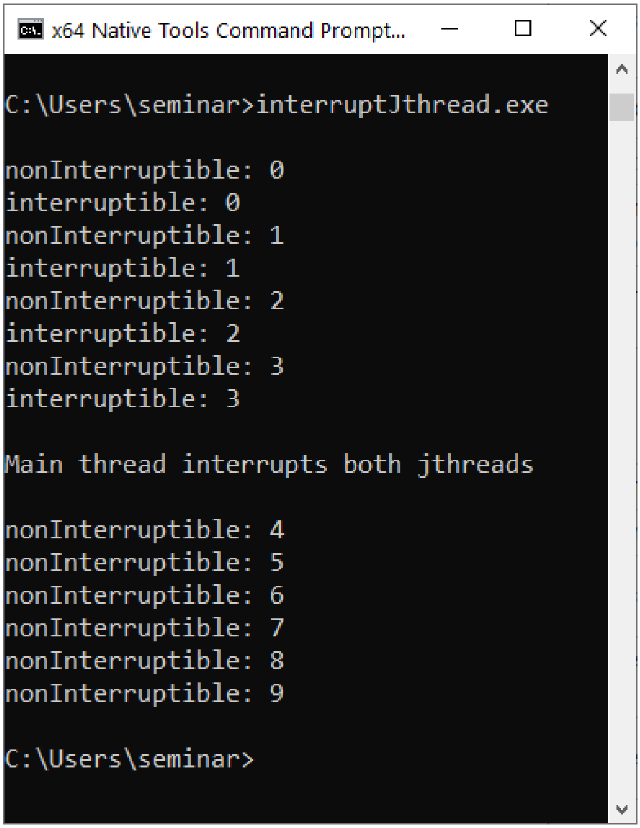
To get the general idea, let me present a simple example.

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';
```

```
36     nonInterruptible.request_stop();
37     interruptible.request_stop();
38 
39     std::cout << '\n';
40 
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.

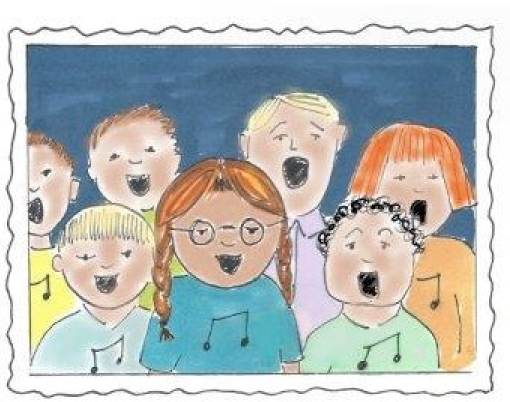


Interrupt a non-interruptible and interruptible `std::jthread`

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. 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`

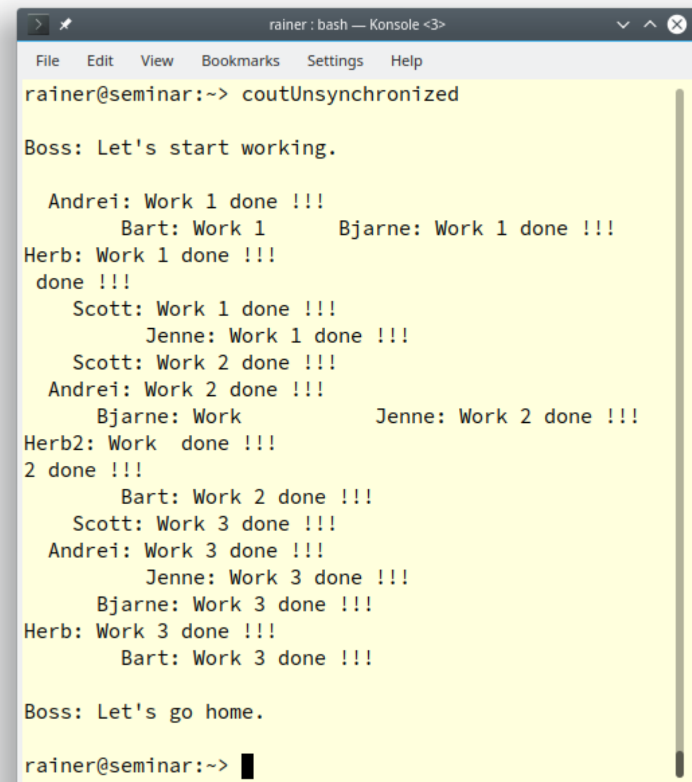
```
// coutUnsynchronized.cpp

#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:
  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"));


  herb.join();
  andrei.join();
  scott.join();
  bjarne.join();
  bart.join();
  jenne.join();

  std::cout << "\n" << "Boss: Let's go home." << '\n';

  std::cout << '\n';

```

```
49 }
```

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 his 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`[38] to synchronize the access to `std::cout`.

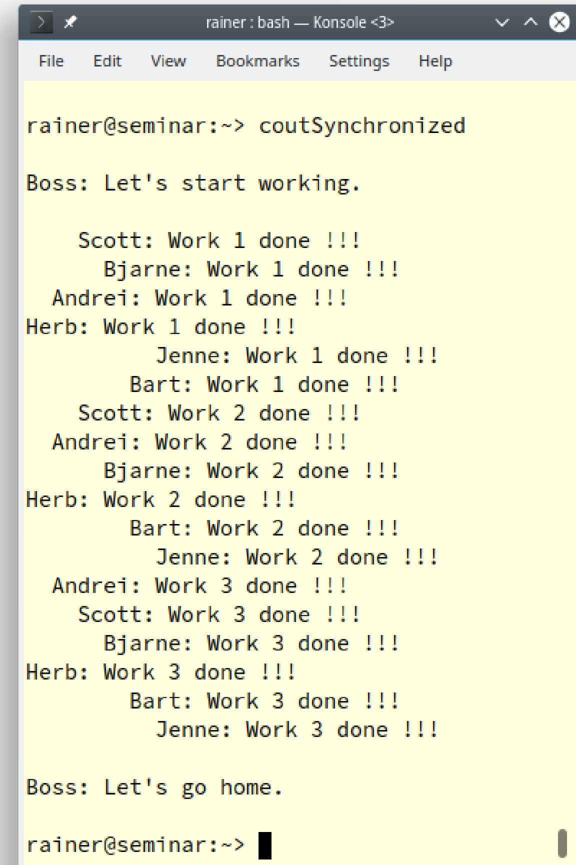
Synchronized access to `std::cout`

```cpp
// 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";
```

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

```
21        }
22      }
23  private:
24    std::string name;
25  };
26
27
28  int main() {
29
30    std::cout << '\n';
31
32    std::cout << "Boss: Let's start working." << "\n\n";
33
34    std::thread herb= std::thread(Worker("Herb"));
35    std::thread andrei= std::thread(Worker("  Andrei"));
36    std::thread scott= std::thread(Worker("    Scott"));
37    std::thread bjarne= std::thread(Worker("      Bjarne"));
38    std::thread bart= std::thread(Worker("        Bart"));
39    std::thread jenne= std::thread(Worker("          Jenne"));
40
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 mess 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`[39]. 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

[39]https://en.cppreference.com/w/cpp/io/basic_streambuf

interleaving of characters can happen.

Thanks to `std::basic_osyncstream`, you can directly write synchronously to `std::cout`.

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';

```

```
30    std::cout << "Boss: Let's start working.\n\n";
31
32    std::thread herb= std::thread(Worker("Herb"));
33    std::thread andrei= std::thread(Worker("  Andrei"));
34    std::thread scott= std::thread(Worker("    Scott"));
35    std::thread bjarne= std::thread(Worker("      Bjarne"));
36    std::thread bart= std::thread(Worker("        Bart"));
37    std::thread jenne= std::thread(Worker("          Jenne"));
38
39
40    herb.join();
41    andrei.join();
42    scott.join();
43    bjarne.join();
44    bart.join();
45    jenne.join();
46
47    std::cout << "\n" << "Boss: Let's go home." << '\n';
48
49    std::cout << '\n';
50
51  }
```

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 << " do\
ne !!!"
                                << '\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.

cppreference.com[40] 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';
```

[40]https://en.cppreference.com/w/cpp/io/basic_osyncstream/get_wrapped

```
} // emits the contents of bout1
```

```
Goodbye, Planet!
Hello, World!
```

Synchronized access of `std::cout`

Distilled Information

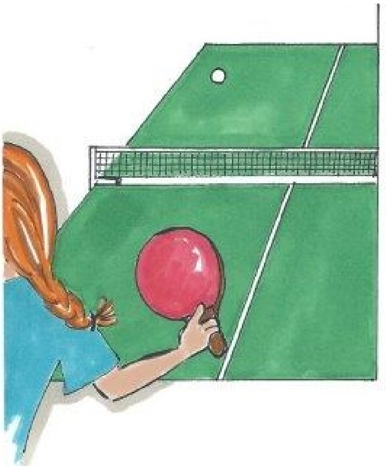
- Although `std::cout` is thread-safe, you may get an interleaving of output operations when threads concurrently write to `std::cout`. This is only a visual issue but not a data race.
- C++20 supports synchronized output streams. They accumulate output in an internal buffer and write their content in 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.

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? 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`.

7.1 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 of which algorithms may work and which algorithms may not work. 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 want 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. When you want the exact number on your platform, you have to repeat the tests.

Let me start the comparison with C++11.

7.1.1 Condition Variables

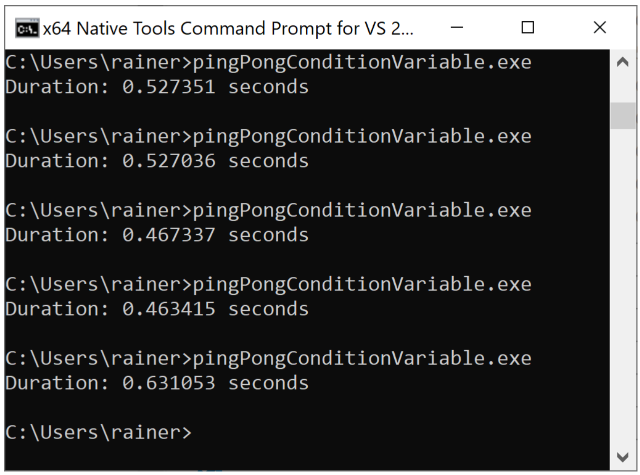
Multiple time synchronization with a condition variable

```
// pingPongConditionVariable.cpp

#include <condition_variable>
#include <iostream>
#include <atomic>
#include <thread>

bool dataReady{false};

std::mutex mutex_;
std::condition_variable condVar1;
std::condition_variable condVar2;

std::atomic<int> counter{};
constexpr int countlimit = 1'000'000;

void ping() {

    while(counter <= countlimit) {
        {
            std::unique_lock<std::mutex> lck(mutex_);
```

```
            condVar1.wait(lck, []{return dataReady == false;});
            dataReady = true;
        }
        ++counter;
        condVar2.notify_one();
  }
}

void pong() {

    while(counter < countlimit) {
        {
            std::unique_lock<std::mutex> lck(mutex_);
            condVar2.wait(lck, []{return dataReady == true;});
            dataReady = false;
        }
        condVar1.notify_one();
  }

}

int main(){

    auto start = std::chrono::system_clock::now();

    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';
}
```

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.

7.1.2 `std::atomic_flag`

Here is the same workflow using two atomic flags and then one.

7.1.2.1 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.

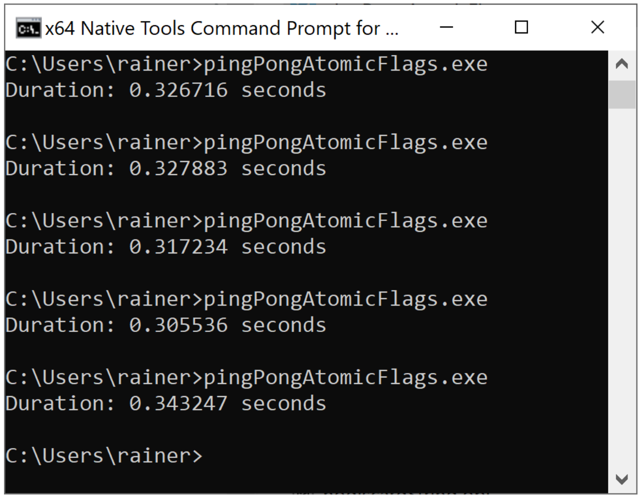
Multiple time synchronization with two atomic flags

```
// pingPongAtomicFlags.cpp

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

std::atomic_flag condAtomicFlag1{};
std::atomic_flag condAtomicFlag2{};

std::atomic<int> counter{};
constexpr int countlimit = 1'000'000;

void ping() {
    while(counter <= countlimit) {
        condAtomicFlag1.wait(false);
        condAtomicFlag1.clear();

        ++counter;

        condAtomicFlag2.test_and_set();
        condAtomicFlag2.notify_one();
    }
}

void pong() {
    while(counter < countlimit) {
        condAtomicFlag2.wait(false);
        condAtomicFlag2.clear();

        condAtomicFlag1.test_and_set();
        condAtomicFlag1.notify_one();
    }
}

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(\
47  ) - start;
48      std::cout << "Duration: " << dur.count() << " seconds" << '\n';
49
50  }
```

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.

7.1.2.2 One Atomic Flag

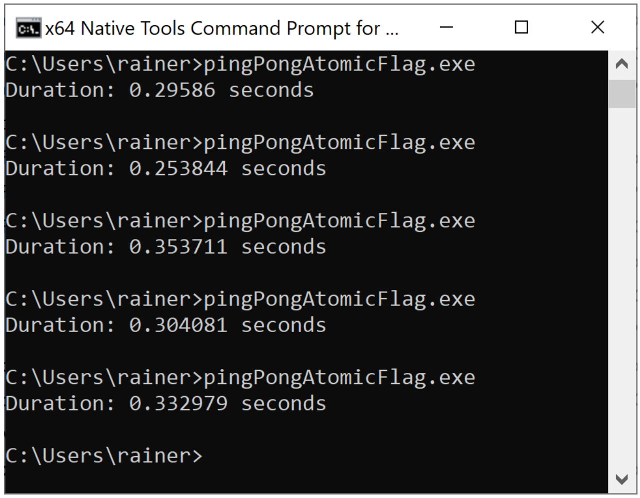
Using one atomic flag makes the workflow easier to understand.

Multiple time synchronization with one atomic flag

```cpp
// 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 in this example `std::atomic_flag` such as an atomic boolean. Let's give it another try with `std::atomic<bool>`.

7.1.3 `std::atomic<bool>`

The following C++20 implementation is based on `std::atomic`.

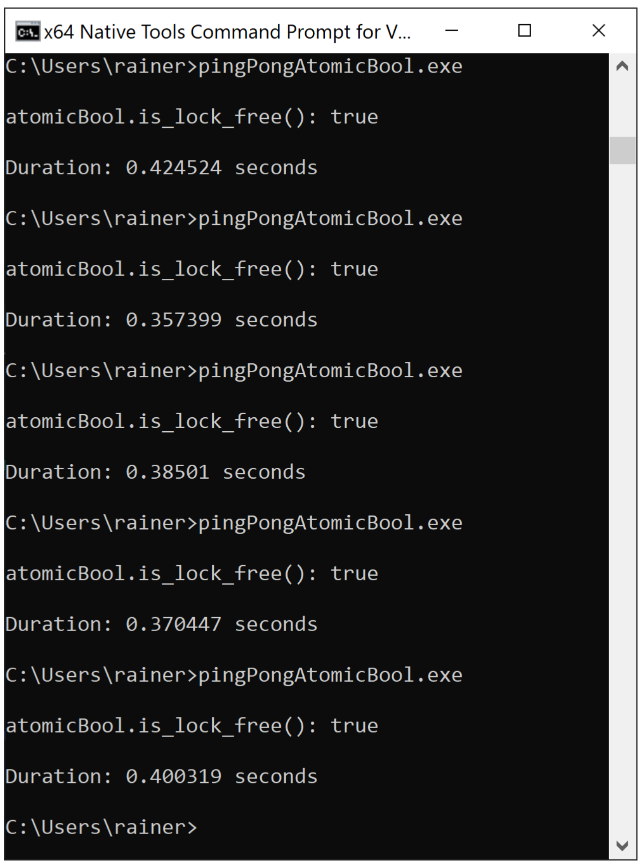
Multiple time synchronization with an atomic bool

```
// pingPongAtomicBool.cpp

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

std::atomic<bool> atomicBool{};

std::atomic<int> counter{};
```

```
constexpr int countlimit = 1'000'000;

void ping() {
    while(counter <= countlimit) {
        atomicBool.wait(true);
        atomicBool.store(true);

        ++counter;

        atomicBool.notify_one();
    }
}

void pong() {
    while(counter < countlimit) {
        atomicBool.wait(false);
        atomicBool.store(false);
        atomicBool.notify_one();
    }
}

int main() {

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

    std::cout << "atomicBool.is_lock_free(): "
              << atomicBool.is_lock_free()  << '\n';

    std::cout << '\n';

    auto start = std::chrono::system_clock::now();

    atomicBool.store(true);
    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';

}
```

`std::atomic<bool>` can internally use a locking mechanism such as 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.

7.1.4 Semaphores

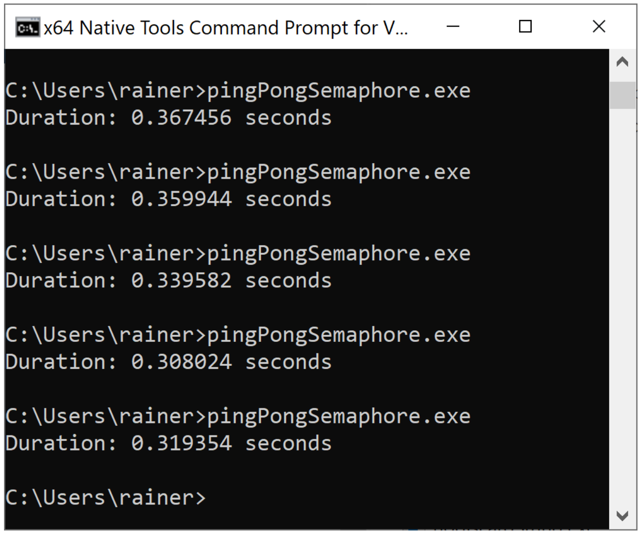
Semaphores promise to be faster than condition variables. Let's see if this is true.

Multiple time synchronization with semaphores

```
// 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();
    }
```

```
26 }
27
28 int main() {
29
30     auto start = std::chrono::system_clock::now();
31
32     signal2Ping.release();
33     std::thread t1(ping);
34     std::thread t2(pong);
35
36     t1.join();
37     t2.join();
38
39     std::chrono::duration<double> dur = std::chrono::system_clock::now(\
40 ) - start;
41     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
42
43 }
```

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.

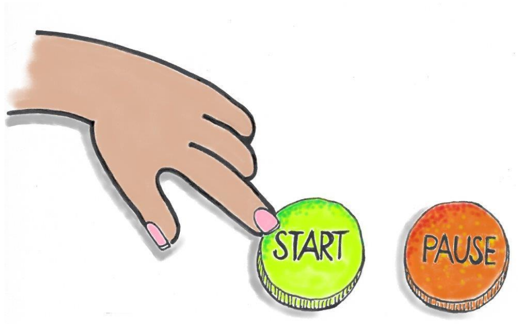
7.1.5 All Numbers

As expected, condition variables are the slowest way, and atomic flag the fastest way to synchronize threads. The performance of a `std::atomic<bool>` is in between. 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

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

```cpp
// 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';
        }
        ~promise_type() {
            std::cout << "        promise_type::~promise_type" << '\n';
        }
        MyFuture<T> get_return_object() {
            std::cout << "        promise_type::get_return_object" << '\
\n';
            return ptr;
        }
        void return_value(T v) {
            std::cout << "        promise_type::return_value" << '\n';
            *ptr = v;
        }
        std::suspend_never initial_suspend() {
            std::cout << "        promise_type::initial_suspend" << '\n\
';
            return {};
        }
        std::suspend_never final_suspend() noexcept {
            std::cout << "        promise_type::final_suspend" << '\n';
            return {};
        }
        void unhandled_exception() {
            std::exit(1);
        }
    };
};

MyFuture<int> createFuture() {
```

```
54     std::cout << "createFuture" << '\n';
55     co_return 2021;
56 }
57
58 int main() {
59
60     std::cout << '\n';
61
62     auto fut = createFuture();
63     auto res = fut.get();
64     std::cout << "res: " << res << '\n';
65
66     std::cout << '\n';
67
68 }
```

The call `createFuture` (line 60) causes the creating 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[1].

[1] https://godbolt.org/z/Y9naEx

```
        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 runs only if 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

```
// lazyFuture.cpp

#include <coroutine>
#include <iostream>
#include <memory>

template<typename T>
struct MyFuture {
    struct promise_type;
    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 {};
```

```
49          }
50          std::suspend_always final_suspend() noexcept {
51              std::cout << "        promise_type::final_suspend" << '\n';
52              return {};
53          }
54          void unhandled_exception() {
55              std::exit(1);
56          }
57      };
58  };
59  
60  MyFuture<int> createFuture() {
61      std::cout << "createFuture" << '\n';
62      co_return 2021;
63  }
64  
65  int main() {
66  
67      std::cout << '\n';
68  
69      auto fut = createFuture();
70      auto res = fut.get();
71      std::cout << "res: " << res << '\n';
72  
73      std::cout << '\n';
74  
75  }
```

Let's first study the promise. The promise always suspends at the beginning (line 44) and 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[2].

```
        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.

[2] https://godbolt.org/z/EejWcj

```
    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 to handle 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, bevor 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

```
// lazyFutureOnOtherThread.cpp

#include <coroutine>
#include <iostream>
#include <memory>
#include <thread>

template<typename T>
struct MyFuture {
    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;
    handle_type coro;

    MyFuture(handle_type h): coro(h) {}
    ~MyFuture() {
        if ( coro ) coro.destroy();
    }

    T get() {
        std::cout << "    MyFuture::get:  "
                  << "std::this_thread::get_id(): "
                  << std::this_thread::get_id() << '\n';

        std::thread t([this] { coro.resume(); });
        t.join();
        return coro.promise().result;
    }

    struct promise_type {
        promise_type(){
            std::cout << "        promise_type::promise_type:  "
                      << "std::this_thread::get_id(): "
                      << std::this_thread::get_id() << '\n';
        }
        ~promise_type(){
```

```
            std::cout << "        promise_type::~promise_type:  "
                      << "std::this_thread::get_id(): "
                      << std::this_thread::get_id() << '\n';
        }

        T result;
        auto get_return_object() {
            return MyFuture{handle_type::from_promise(*this)};
        }
        void return_value(T v) {
            std::cout << "        promise_type::return_value:  "
                      << "std::this_thread::get_id(): "
                      << std::this_thread::get_id() << '\n';
            std::cout << v << std::endl;
            result = v;
        }
        std::suspend_always initial_suspend() {
            return {};
        }
        std::suspend_always final_suspend() noexcept {
            std::cout << "        promise_type::final_suspend:  "
                      << "std::this_thread::get_id(): "
                      << std::this_thread::get_id() << '\n';
            return {};
        }
        void unhandled_exception() {
            std::exit(1);
        }
    };
};

MyFuture<int> createFuture() {
    co_return 2021;
}

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[3] 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, finishes before it returns `coro.promise().result`.

[3]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 to 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

```
// infiniteDataStreamComments.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) {
        std::cout << "        Generator::Generator" << '\n';
    }
    handle_type coro;

    ~Generator() {
        std::cout << "        Generator::~Generator" << '\n';
        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;
    }
    int getNextValue() {
        std::cout << "        Generator::getNextValue" << '\n';
        coro.resume();
        return coro.promise().current_value;
    }
    struct promise_type {
        promise_type() {
            std::cout << "            promise_type::promise_type" << '\n';
        }

        ~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 {};
        }
        auto get_return_object() {
            std::cout << "            promise_type::get_return_object" \
<< '\n';
            return Generator{handle_type::from_promise(*this)};
        }

        std::suspend_always yield_value(int value) {
            std::cout << "            promise_type::yield_value" << '\n\
';
            current_value = value;
            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[4] makes the control flow transparent.

[4] 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 want to make a first modification of the control flow.

7.3.1 Modifications

The snippets and line numbers are all based on the previous program `infiniteDataStreamComment` 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 immediately pauses.

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 start 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() {}

        ~promise_type() {}

        std::suspend_always initial_suspend() {
            return {};
        }
        std::suspend_always final_suspend() noexcept {
            return {};
        }
        auto get_return_object() {
            return Generator{handle_type::from_promise(*this)};
        }

        std::suspend_always yield_value(const T value) {
            current_value = value;
            return {};
        }
         void return_void() {}
        void unhandled_exception() {
            std::exit(1);
        }

        T current_value;
    };

};

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[5].

```
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 elements of the container. After each iteration, the function template pauses. The return type `Generator<typename Cont::value_type>` may look surprising to you. `Cont::value_type` is 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`.

[5]https://godbolt.org/z/j9znva

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)

```
// startJobWithComments.cpp

#include <coroutine>
#include <iostream>

struct MySuspendAlways {
    bool await_ready() const noexcept {
        std::cout << "        MySuspendAlways::await_ready"  << '\n';
```

```
        return false;
    }
    void await_suspend(std::coroutine_handle<>) const noexcept {
        std::cout << "        MySuspendAlways::await_suspend"  << '\n';

    }
    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 {};
        }
        void return_void() {}
        void unhandled_exception() {}

    };
};

Job prepareJob() {
    co_await MySuspendNever();
}

int main() {

    std::cout <<  "Before job" << '\n';

    auto job = prepareJob();
    job.start();

    std::cout <<  "After job" <<  '\n';

}
```

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[6].

```
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.

[6] https://godbolt.org/z/T5rcE4

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() {
            std::cout << "    Job prepared" << '\n';
            return {};
        }
        std::suspend_always final_suspend() noexcept {
            std::cout << "    Job finished" << '\n';
            return {};
        }
```

```
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. When the boolean is `true`, the control flow jumps directly to the member function `await_resume` (line 23), when `false`, the coroutine is immediately suspended and, therefore, the function `await_suspend` runs (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[7].

```
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.

[7] 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 {
```

```
36          int JobNumber{JobCounter};
37          Job get_return_object() { return {}; }
38          std::suspend_never initial_suspend() {
39              std::cout << "    Job " << JobNumber << " prepared on threa\
40  d "
41                        << std::this_thread::get_id();
42              return {};
43          }
44          std::suspend_never final_suspend() noexcept {
45              std::cout << "    Job " << JobNumber << " finished on threa\
46  d "
47                        << std::this_thread::get_id() << '\n';
48              return {};
49          }
50          void return_void() {}
51          void unhandled_exception() { }
52      };
53  };
54  
55  Job performJob(std::jthread& out) {
56      co_await MyAwaitable{out};
57  }
58  
59  int main() {
60  
61      std::vector<std::jthread> threads(8);
62      for (auto& thr: threads) performJob(thr);
63  
64  }
```

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) return 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[8].

```
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.

[8] https://wandbox.org/permlink/skHgWKF0SYAwp8Dm

Distilled Information

- When you want to synchronize 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 the fastest way to synchronize threads. The performance of `std::atomic<bool>` is in between. Semaphores are nearly as fast as atomic flags.
- 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 own thread.
- Modifications of the generator for an infinite data stream reveals 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.
- Modification of the Awaitable empowers you to automatically resume the coroutine on a separate thread.

Epilogue

Congratulations! When you read these lines, you have mastered the challenging and thrilling C++20 standard. C++20 is a C++ standard that likely has the same influence for 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 revolutionize 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, can be evaluated lazily, and can 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. There is work to be done in C++23 to fully integrate and use the potential of the *Big Four* in C++. Let me give you a few ideas about the near C++ future.

- The Standard Template Library was designed by Alexander Stephanov[9] with concepts in mind. Still, the integration of concepts is missing in C++20.
- We can expect a modularized Standard Template Library and hope for a packaging system in C++.
- Many algorithms known from functional programming are still missing in the ranges library. A future C++ standard should improve the interplay of the range algorithms and the standard containers.
- We don't have coroutines. We only have a framework for building powerful coroutines. A coroutines library will be, with high probability, in C++23.

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, shiny future.

[9] https://en.wikipedia.org/wiki/Alexander_Stepanov

Further Information

8. C++23 and Beyond

Whoever might think that a significant C++ standard is followed by a small C++ standard is wrong. C++23 will provide just as powerful extensions as C++20 does. Ville Voutilainen's proposal P0592R4[1] "To boldly suggest an overall plan for C++23" gives a first idea of the upcoming C++23 standard. Ville names seven features.

- C++23
  - Library support for coroutines
  - A modular standard library
  - Executors
  - Networking
- C++23 or later
  - Reflection
  - Pattern Matching
  - Contracts

The first four features are aimed for C++23, and the remaining three have no specific schedule. It is likely that reflection, pattern matching, and contracts are successively added to the C++ standards.

"Prediction is very difficult, especially if it's about the future." (Niels Bohr[2]). Consequently, you should read this chapter as my best attempt to predict the C++ future.

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html
[2] https://www.goodreads.com/quotes/23796-prediction-is-very-difficult-especially-about-the-future

8.1 C++23

The coroutines library, the modularized standard library, and the executors have something in common: they are supposed to be part of C++23.

8.1.1 The Coroutines Library

Coroutines in C++20 are no more than a framework for the implementation of concrete coroutines. This means that it is up to the software developer to implement coroutines. The cppcoro[3] library from Lewis Baker gives the first idea how a library of coroutines could look like. His library provides what C++20 could not offer: high-level coroutines.

[3] https://github.com/lewissbaker/cppcoro

Using cppcoro

The cppcoro library is based on the coroutines TS. The TS stands for technical specification and is the preliminary version of the coroutines functionality we get with C++20. Lewis will presumably port the cppcoro library from the coroutines TS to the coroutines defined in C++20. The library can be used on Windows (Visual Studio 2017) or Linux (Clang 5.0/6.0 and libc++). For my experiments, I used the following command line for all examples:

cppcoro command line

```
clang++ -std=c++17 -fcoroutines-ts -Iinclude -stdlib=libc++ libcppcoro.\
a
        cppcoroTask.cpp -pthread
```

- `-std=c++17`: support for C++17
- `-fcoroutines-ts` : support for the C++ coroutines TS
- `-Iinclude` : cppcoro headers
- `-stdlib=libc++`: LLVM[4] implementation of the standard library
- `libcppcoro.a`: cppcoro library

As I already mentioned, when cppcoro is based on C++20 coroutines, you can use them with each compiler that supports C++20. Additionally, they give you a flavor for the concrete coroutines we may get with C++23.

In the rest of this section to the coroutines library, I want to demonstrate a few examples that show the power of coroutines. My demonstration starts with the coroutine types.

8.1.1.1 Coroutine Types

cppcoro has various kinds of tasks and generators.

8.1.1.1.1 `task<T>`

What is a task? This is the definition used in cppcoro:

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

- A `task` represents an asynchronous computation that is executed lazily in that the execution of the coroutine does not start until the task is awaited.

A task is a coroutine. In the following program, the function `main` waits for the function `first`, `first` waits for `second`, and second waits for `third`.

Coroutines first sleeping

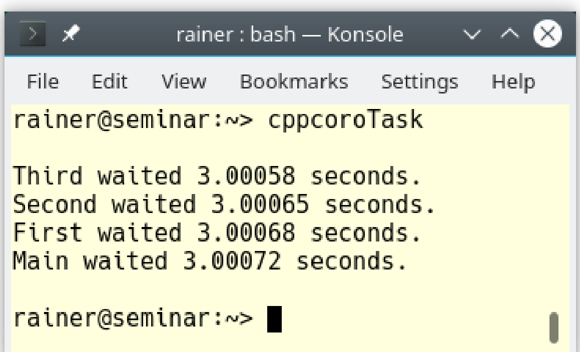
```
// cppcoroTask.cpp

#include <chrono>
#include <iostream>
#include <string>
#include <thread>

#include <cppcoro/sync_wait.hpp>
#include <cppcoro/task.hpp>

using std::chrono::high_resolution_clock;
using std::chrono::time_point;
using std::chrono::duration;

using namespace std::chrono_literals;

auto getTimeSince(const time_point<high_resolution_clock>& start) {

    auto end = high_resolution_clock::now();
    duration<double> elapsed = end - start;
    return elapsed.count();

}

cppcoro::task<> third(const time_point<high_resolution_clock>& start) {

    std::this_thread::sleep_for(1s);
    std::cout << "Third waited " << getTimeSince(start) << " seconds." \
<< '\n';
```

```
    co_return;
}
cppcoro::task<> second(const time_point<high_resolution_clock>& start) {
    auto thi = third(start);
    std::this_thread::sleep_for(1s);
    co_await thi;
    std::cout << "Second waited " <<  getTimeSince(start) << " seconds.\
" << '\n';
}
cppcoro::task<> first(const time_point<high_resolution_clock>& start) {
    auto sec = second(start);
    std::this_thread::sleep_for(1s);
    co_await sec;
    std::cout << "First waited " <<  getTimeSince(start)  << " seconds.\
" << '\n';
}
int main() {
    std::cout << '\n';
    auto start = high_resolution_clock::now();
    cppcoro::sync_wait(first(start));
    std::cout << "Main waited " <<  getTimeSince(start) << " seconds." \
<< '\n';
```

```
66
67     std::cout << '\n';
68
69 }
```

Admittedly, the program doesn't do anything meaningful, but it helps to understand the workflow of coroutines.

First of all, the `main` function can't be a coroutine. `cppcoro::sync_wait` (line 59) often serves, such as in this case, as a starting top-level task and waits until the task is finished. The coroutine `first`, similar to the other coroutines, gets as an argument the start time and displays its execution time. What does the coroutine `first` do? It starts the coroutine `second` (line 36 and 46), which is immediately paused, sleeps for a second, and resumes the coroutine via its handle `sec` (line 38 and 48). The coroutine `second` performs the same workflow, but not the coroutine `third`. As for `third` it is a coroutine that returns nothing and does not wait on another coroutine. When `third` is done, all other coroutines are executed. Consequently, each coroutine takes 3 seconds.



Coroutines first sleeping

Let's vary the program a little. What happens if the coroutines sleep after the `co_-await` call?

Coroutines first waiting

```
// cppcoroTask2.cpp

#include <chrono>
#include <iostream>
#include <string>
#include <thread>

#include <cppcoro/sync_wait.hpp>
#include <cppcoro/task.hpp>

using std::chrono::high_resolution_clock;
using std::chrono::time_point;
using std::chrono::duration;

using namespace std::chrono_literals;

auto getTimeSince(const time_point<::high_resolution_clock>& start) {

    auto end = high_resolution_clock::now();
    duration<double> elapsed = end - start;
    return elapsed.count();

}

cppcoro::task<> third(const time_point<high_resolution_clock>& start) {

    std::cout << "Third waited " << getTimeSince(start) << " seconds." \
<< '\n';
    std::this_thread::sleep_for(1s);
    co_return;

}

cppcoro::task<> second(const time_point<high_resolution_clock>& start) {

```

```
    auto thi = third(start);
    co_await thi;

    std::cout << "Second waited " <<  getTimeSince(start) << " seconds.\
" << '\n';
    std::this_thread::sleep_for(1s);

}

cppcoro::task<> first(const time_point<high_resolution_clock>& start) {

    auto sec = second(start);
    co_await sec;

    std::cout << "First waited " <<  getTimeSince(start)  << " seconds.\
" << '\n';
    std::this_thread::sleep_for(1s);

}

int main() {

    std::cout << '\n';

    auto start = ::high_resolution_clock::now();

    cppcoro::sync_wait(first(start));

    std::cout << "Main waited " <<  getTimeSince(start) << " seconds." \
<< '\n';

    std::cout << '\n';

}
```

You may have guessed it. The main function waits 3 seconds, but each iteratively-invoked coroutine one second less.

!Coroutines first waiting](images/Cpp23/cppcoroTask2.png)

The next coroutine that cppcoro provides is a `generator<T>`.

8.1.1.1.2 `generator<T>`

Here is cppcoro's definition of a `generator`:

- A `generator` represents a coroutine type that produces a sequence of values of type T, where values are produced lazily and synchronously.

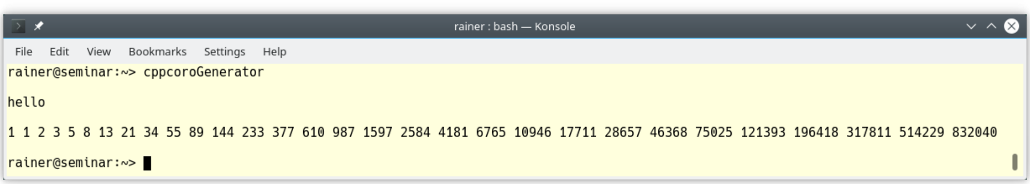
Without further ado, the program `cppcoroGenerator.cpp` demonstrates two generators in action.

Use of two generators

```cpp
// cppcoroGenerator.cpp

#include <iostream>
#include <cppcoro/generator.hpp>

cppcoro::generator<char> hello() {
    co_yield 'h';
    co_yield 'e';
    co_yield 'l';
    co_yield 'l';
    co_yield 'o';
}

cppcoro::generator<const long long> fibonacci() {
    long long a = 0;
    long long b = 1;
    while (true) {
        co_yield b;
        auto tmp = a;
```

```
20          a = b;
21          b += tmp;
22      }
23  }
24
25  int main() {
26
27      std::cout << '\n';
28
29      for (auto c: hello()) std::cout << c;
30
31      std::cout << "\n\n";
32
33      for (auto i: fibonacci()) {
34          if (i > 1'000'000) break;
35          std::cout << i << " ";
36      }
37
38      std::cout << "\n\n";
39
40  }
```

The first coroutine `hello` returns on request the next character and the coroutine `fibonacci` the next fibonacci number. `fibonacci` creates an infinite data stream. What happens in line 33? The range-based for loop triggers the execution of the coroutine. The first iteration starts the coroutines, returns the value at `co_yield b` (line 18), and pauses. Subsequent calls of the range-based for loop resume the coroutine `fibonacci` and return the next fibonacci number.



Executing two generators

cppcoro provides more awaitable types.

8.1.1.2 Awaitable Types

cppcoro supports various awaitable types:

- `single_consumer_event`
- `single_consumer_async_auto_reset_event`
- `async_mutex`
- `async_manual_reset_event`
- `async_auto_reset_event`
- `async_latch`
- `sequence_barrier`
- `multi_producer_sequencer`
- `single_producer_sequencer`

I want to have a closer look at the awaitables `single_consumer_event` and `async_mutex`.

8.1.1.2.1 `single_consumer_event`

The `single_consumer_event` is, according to the documentation, a simple manual-reset event type that supports only a single coroutine awaiting it at a time. `single_consumer_event` provides a new way for the one-time synchronization of threads.

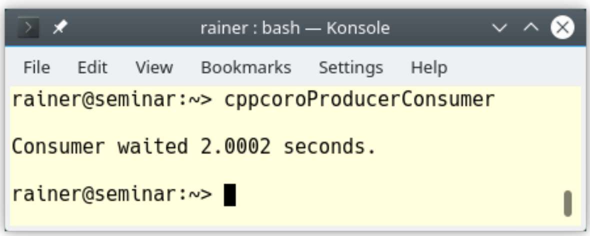
One-time thread synchronization with cppcoro

```
// cppcoroProducerConsumer.cpp

#include <cppcoro/single_consumer_event.hpp>
#include <cppcoro/sync_wait.hpp>
#include <cppcoro/task.hpp>

#include <future>
#include <iostream>
#include <string>
```

```cpp
#include <thread>
#include <chrono>

cppcoro::single_consumer_event event;

cppcoro::task<> consumer() {

    auto start = std::chrono::high_resolution_clock::now();

    co_await event;  // suspended until some thread calls event.set()

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Consumer waited " << elapsed.count() << " seconds." <\
< '\n';

    co_return;
}

void producer() {

    using namespace std::chrono_literals;
    std::this_thread::sleep_for(2s);

    event.set();  // resumes the consumer

}

int main() {

    std::cout << '\n';

    auto con = std::async([]{ cppcoro::sync_wait(consumer()); });
    auto prod = std::async(producer);

    con.get(), prod.get();
```

```
46
47     std::cout << '\n';
48
49 }
```

The code should be self-explanatory. The `consumer` (line 41) and the `producer` (line 42) run in their thread. The call `cppcoro::sync_wait(consumer())` (line 41) serves as a top-level task because the main function cannot be a coroutine. The call waits until the coroutine consumer is done. The coroutine consumer waits in the call `co_await event` (line 19) until someone calls `event.set()` (line 33). The function `producer` sends its event after a sleep of two seconds.



One-time thread synchronization with cppcoro

cppcoro also supports a mutex[5].

8.1.1.2.2 `async_mutex`

A mutex such as `cppcoro::async_mutex` is a synchronization mechanism to protect shared data from being accessed by multiple threads simultaneously.

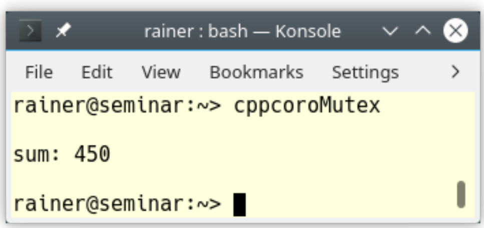
[5] https://en.cppreference.com/w/cpp/named_req/Mutex

Mutual exclusion with cppcoro

```
// cppcoroMutex.cpp

#include <cppcoro/async_mutex.hpp>
#include <cppcoro/sync_wait.hpp>
#include <cppcoro/task.hpp>

#include <iostream>
#include <thread>
#include <vector>


cppcoro::async_mutex mutex;

int sum{};                                                                  \


cppcoro::task<> addToSum(int num) {
    cppcoro::async_mutex_lock lockSum = co_await mutex.scoped_lock_asyn\
c();
    sum += num;

}                                                                           \


int main() {

    std::cout << '\n';

    std::vector<std::thread> vec(10);                                       \


    for(auto& thr: vec) {
        thr = std::thread([]{
            for(int n = 0; n < 10; ++n) cppcoro::sync_wait(addToSum(n))\
; } );
```

```
36      }
37
38      for(auto& thr: vec) thr.join();
39
40      std::cout << "sum: " << sum << '\n';
41
42      std::cout << '\n';
43
44  }
```

Line 26 creates ten threads. Each thread adds the numbers 0 to 9 to the shared `sum` variable (line 14). The function `addToSum` is the coroutine. The coroutine waits in the expression `co_await mutex.scoped_lock_async()` (line 17) until the mutex is acquired. The coroutine that waits for the mutex is not blocked but suspended. The previous lock holder resumes the waiting coroutine in its unlock call. As the name suggests, the mutex stays locked until the end of its scope (line 20).



Mutual exclusion with cppcoro

8.1.1.3 Functions

There are more interesting functions to handle awaitables.

- `sync_wait()`
- `when_all()`
- `when_all_ready()`
- `fmap()`
- `schedule_on()`

- `resume_on()`

The function `when_all` creates an awaitable that waits for all its input-awaitables, and returns an aggregate of their individual results.

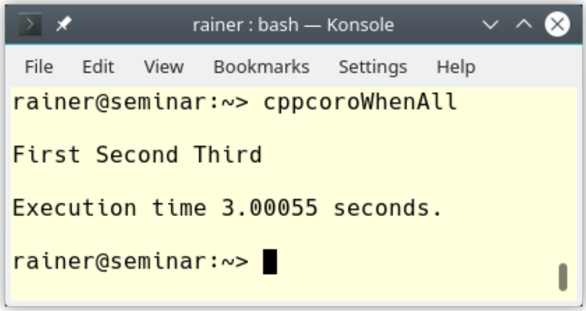
The following example should give you the first impression:

Waiting for all awaitables with `when_all`

```
// cppcoroWhenAll.cpp

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

#include <cppcoro/sync_wait.hpp>
#include <cppcoro/task.hpp>
#include <cppcoro/when_all.hpp>

using namespace std::chrono_literals;

cppcoro::task<std::string> getFirst() {
    std::this_thread::sleep_for(1s);
    co_return "First";
}

cppcoro::task<std::string> getSecond() {
     std::this_thread::sleep_for(1s);
    co_return "Second";
}

cppcoro::task<std::string> getThird() {
     std::this_thread::sleep_for(1s);
    co_return "Third";
}


```

```
29 cppcoro::task<> runAll() {
30 
31     auto[fir, sec, thi] = co_await cppcoro::when_all(getFirst(), getSec\
32 ond(),
33                                                      getThird());
34 
35     std::cout << fir << " " << sec << " " << thi << '\n';
36 
37 }
38 
39 int main() {
40 
41     std::cout << '\n';
42 
43     auto start = std::chrono::steady_clock::now();
44 
45     cppcoro::sync_wait(runAll());
46 
47     std::cout << '\n';
48 
49     auto end = std::chrono::high_resolution_clock::now();
50     std::chrono::duration<double> elapsed = end - start;
51     std::cout << "Execution time " << elapsed.count() << " seconds." <<\
52  '\n';
53 
54     std::cout << '\n';
55 
56 }
```

The top-level task `cppcoro::sync_wait(runAll())` (line 44) awaits the awaitable `runAll`, which awaits the awaitables `getFirst`, `getSecond`, and `getThird` (line 31). The awaitables `runAll`, `getFirst`, `getSecond`, and `getThird` are coroutines. Each of the get functions sleeps for one second (line 14, 19, and 24). Three times one second makes three seconds. This is the time the call `cppcoro::sync_wait(runAll())` waits for the coroutines. Line 49 displays the time duration.



Waiting for all awaitables with `when_all`

You can combine `when_all` with thread pools in cppcoro.

8.1.1.4 `static_thread_pool`

`static_thead_pool` schedules work on a fixed-size pool of threads.

`cppcoro::static_thread_pool` can be invoked with and without a number. The number stands for the number of threads that are created. If you don't specify a number, the C++11 function `std::thread::hardware_concurrency()` is used. std::thread::hardware_concurrency[6] gives you a hint for the number of hardware threads supported by your system. This may be the number of processors or cores you have.

Let me try it out. The following example is based on the previous one `cppcoroWhenAll.cpp` using the awaitable `when_any`. This time, the coroutines are executed concurrently.

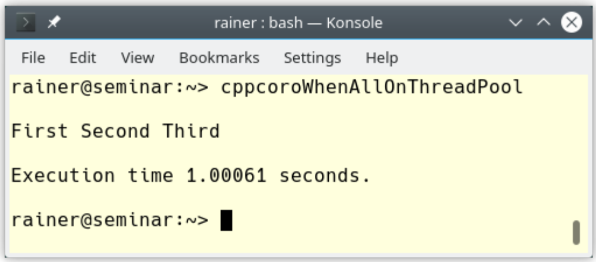
[6]https://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency

Waiting for concurrently running awaitables with `when_all`

```
// cppcoroWhenAllOnThreadPool.cpp

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

#include <cppcoro/sync_wait.hpp>
#include <cppcoro/task.hpp>
#include <cppcoro/static_thread_pool.hpp>
#include <cppcoro/when_all.hpp>


using namespace std::chrono_literals;

cppcoro::task<std::string> getFirst() {
    std::this_thread::sleep_for(1s);
    co_return "First";
}

cppcoro::task<std::string> getSecond() {
    std::this_thread::sleep_for(1s);
    co_return "Second";
}

cppcoro::task<std::string> getThird() {
    std::this_thread::sleep_for(1s);
    co_return "Third";
}

template <typename Func>
cppcoro::task<std::string> runOnThreadPool(cppcoro::static_thread_pool&\
 tp,
                                           Func func) {
    co_await tp.schedule();
    auto res = co_await func();
```

```cpp
    co_return res;
}

cppcoro::task<> runAll(cppcoro::static_thread_pool& tp) {

    auto[fir, sec, thi] = co_await cppcoro::when_all(
        runOnThreadPool(tp, getFirst),
        runOnThreadPool(tp, getSecond),
        runOnThreadPool(tp, getThird));

    std::cout << fir << " " << sec << " " << thi << '\n';

}

int main() {

    std::cout << '\n';

    auto start = std::chrono::steady_clock::now();

    cppcoro::static_thread_pool tp;
    cppcoro::sync_wait(runAll(tp));

    std::cout << '\n';

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Execution time " << elapsed.count() << " seconds." <<\
 '\n';

    std::cout << '\n';

}
```

This is the crucial difference with the previous program `cppcoroWhenAll.cpp`. At line 55, I create a thread pool `tp` and use it as an argument for the function `runAll(tp)` (line 56). The function `runAll` uses the thread pool to start the coroutines concurrently. Thanks to structured binding (line 40), the values of each coroutine can be easily aggregated and assigned to a variable. In the end, the main function takes one instead of three seconds.



Waiting for all awaitables with `when_all`

8.1.2 Modularized Standard Library for Modules

Maybe you'd like to stop using Standard Library headers? Microsoft supports modules for all STL headers according to the C++ proposal P0541[7]. Microsoft's implementation gives you the first idea of how a modularized standard library for modules could look like. Here is what I have found in the post Using C++ Modules in Visual Studio 2017[8] from the Microsoft C++ team blog.

8.1.2.1 C++ modules in Visual Studio 2017

- `std.regex` provides the content of the header `<regex>`
- `std.filesystem` provides the content of the header `<experimental/filesystem>`
- `std.memory` provides the content of the header `<memory>`
- `std.threading` provides the contents of headers `<atomic>`, `<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>`, and `<thread>`
- `std.core` provides everything else in the C++ Standard Library

To use the Microsoft Standard Library modules, you have to specify the exception handling model (`/EHsc`) and the multithreading library (`/MD`). Additionally, you have to use the flags `/std:c++latest` and `/experimental:module`.

In the section on modules, I used the following module definition.

A module definition with a global module fragment

```
// math1.ixx

module;

#include <numeric>
#include <vector>

export module math;

export int add(int fir, int sec){
```

[7] http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0581r0.pdf
[8] https://devblogs.microsoft.com/cppblog/cpp-modules-in-visual-studio-2017/

```
    return fir + sec;
}

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

This module definition can directly be refactored using the modularized standard library. You have to replace the headers `<numeric>` and `<vector>` with the module `std.core`.

Importing the module `std.core` into the interface file

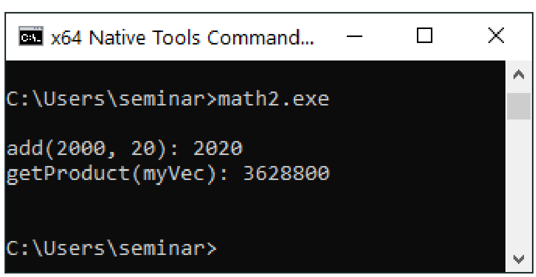
```
// math2.ixx

module;

export module math;

import std.core;

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

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

Furthermore, you must use the module `std.core` instead of the standard header files:

Importing the module std.core into the client program

```
// client2.cpp

import math;
import std.core;

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 program produces the expected output:



Using the module `std.core` on Windows

8.1.3 Executors

Executors have quite a history in C++. The discussion began at early as 2010. For the details, Detlef Vollmann gives in his presentation Finally Executors for C++[9] an excellent overview.

My introduction to executors is mainly based on the proposals for the design of executors P0761[10], and their formal description P0443[11]. I also refer to the relatively new Modest Executor Proposal P1055[12].

First of all. What are Executors?

Executors are the basic building blocks for execution in C++ and fulfill a similar role for execution, such as allocators for the containers in C++. Many proposals for executors are published, and many design decisions are still open. They should be part of C++23, but can probably be used much earlier to extend the C++ standard.

An executor consists of rules about where, when, and how to run a callable.

- **Where**: The callable may run on an internal or external processor, and that the result is read back from the internal or external processor.
- **When**: The callable may run immediately or just be scheduled.
- **How**: The callable may run on a CPU or GPU or even be executed in a vectorized way.

The concurrency and parallelism features of C++ heavily depend on executors as building blocks for execution. This dependency holds for existing concurrency features, such as the parallel algorithms of the Standard Template Library[13], but also for new concurrency features, such as latches and barriers, coroutines, the network library, extended futures[14], transactional memory[15], or task blocks[16].

[9] http://www.vollmann.ch/en/presentations/executors2018.pdf
[10] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0761r2.pdf
[11] http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0443r7.html
[12] http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1055r0.pdf
[13] https://www.modernescpp.com/index.php/parallel-algorithm-of-the-standard-template-library
[14] https://www.modernescpp.com/index.php/std-future-extensions
[15] https://www.modernescpp.com/index.php/transactional-memory
[16] https://www.modernescpp.com/index.php/task-blocks

8.1.3.1 First Examples

The following code snippets should give you a first impression of executors.

8.1.3.1.1 Using an Executor

- The promise `std::async`

`std::async` uses an executor

```
// get an executor through some means
my_executor_type my_executor = ...

// launch an async using my executor
auto future = std::async(my_executor, [] {
    std::cout << "Hello world, from a new execution agent!" < '\n';
});
```

- The STL algorithm `std::for_each`

`std::for_each` uses an executor

```
// get an executor through some means
my_executor_type my_executor = ...

// execute a parallel for_each "on" my executor
std::for_each(std::execution::par.on(my_executor),
              data.begin(), data.end(), func);
```

8.1.3.1.2 Obtaining an Executor

There are various ways to obtain an executor.

- From the execution context `static_thread_pool`

An exector from the `static_thread_pool`

```
// create a thread pool with 4 threads
static_thread_pool pool(4);

// get an executor from the thread pool
auto exec = pool.executor();

// use the executor on some long-running task
auto task1 = long_running_task(exec);
```

- From the system executor

The system executor is the default executor used if not specified otherwise.

- From an executor adapter

Adapting an executor

```
// get an executor from a thread pool
auto exec = pool.executor();

// wrap the thread pool's executor in a logging_executor
logging_executor<decltype(exec)> logging_exec(exec);

// use the logging executor in a parallel sort
std::sort(std::execution::par.on(logging_exec), my_data.begin(), my_dat\
a.end());
```

`logging_executor` is a wrapper for the pool executor.

8.1.3.2 Goals of an Executor Concept

What are the goals of an executor concept according to proposal P1055[17]?

- **Batchable**: control the trade-off between the cost of the transition of the callable and its size.
- **Heterogenous**: allow the callable to run on heterogeneous contexts and get the result back.
- **Orderable**: specify the order in which the callables are invoked. The goal includes ordering guarantees such as LIFO (Last In, First Out), FIFO (First In, First Out) execution, priority or time constraints, or even sequential execution.
- **Controllable**: the callable has to be targetable to a specific compute resource, deferred, or even canceled.
- **Continuable**: for non-blocking submission of work units, signals from the work units are needed. These signals have to indicate, whether the result is available, whether an error occurred, when the callable is done or if the callee wants to cancel the callable. The explicit starting of the callable or the stopping of the staring should also be possible.
- **Layerable**: hierarchies allow new capabilities to be added without increasing the complexity of the simpler use-cases.
- **Usable**: ease of use for the implementer and the user should be the main goal.
- **Composable**: allows a user to extend the executors for features that are not part of the standard.
- **Minimal**: nothing should exist on the executor concepts that could be added externally in a library on top of the concept.

8.1.3.3 Execution Function

An executor provides one or more execution functions for creating execution agents from a callable. An executor has to support at least one of the six following functions.

[17] http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1055r0.pdf

Exuction functions of a executor

Member function	Cardinality	Direction
`execute`	`single`	`oneway`
`twoway_execute`	`single`	`twoway`
`then_execute`	`single`	`then`
`bulk_execute`	`bulk`	`oneway`
`bulk_twoway_execute`	`bulk`	`twoway`
`bulk_then_execute`	`bulk`	`then`

Each execution function has two properties: cardinality and direction.

- Cardinality:
  - `single`: creates one execution agent
  - `bulk`: creates a group of execution agents
- Direction:
  - `oneway`: creates an execution agent and does not return a result
  - `twoway`: creates an execution agent and returns a future that can be used to wait for execution to complete
  - `then`: creates an execution agent and returns a future that can be used to wait for execution to complete. The execution agent begins execution after a given future becomes ready.

The next lines give a more formal explanation of the execution functions.

First, I refer to the single cardinality case:

- A `oneway` execution function is a fire-and-forget job. It's quite similar to a fire-and-forget future, but it does not automatically block in the destructor of the future[18].
- A `twoway` execution function returns you a future which you can use to pick up the result. This behaves similarly to a `std::promise`[19] that gives you back the handle to the associated `std::future`.

[18]https://www.modernescpp.com/index.php/the-special-futures
[19]https://www.modernescpp.com/index.php/promise-and-future

- A `then` execution function is a continuation. It gives you back a future, but the execution agent runs only if the provided future is ready.

Second, the bulk cardinality case is more complicated. These functions create a group of execution agents, and each of these execution agents calls the given callable. They return the result of a factory and not the result of a single callable `f` invoked by the execution agents. The user is responsible for disambiguating the right result via this factory.

8.1.3.3.1 `execution::require`

How can you be sure that your executor supports the specific execution function?

In the special case, you know it:

An executor using the execution function `execute`

```
void concrete_context(const my_oneway_single_executor& ex)
{
    auto task = ...;
    ex.execute(task);
}
```

In the general case, you can use the function `execution::require` to ask for it.

An executor requiring a `single` and `twoway` execution function

```
template <typename Executor>
void generic_context(const Executor& ex)
{
    auto task = ...;

    // ensure .twoway_execute() is available with execution::require()
    execution::require(ex, execution::single, execution::twoway).twoway\
_execute(task);
}
```

In this case, the executor `ex` has to support single cardinality and twoway direction execution.

8.1.4 The Network Library

The network library in C++23 is based on the boost::asio[20] library from Christopher M. Kohlhoff. The library targets the network and low-level I/O programming.

The following components are part of the network library:

- TCP, UDP, and multicast
- Client/Server applications
- Scalability for more concurrent connections
- IPv4 and IPv6
- Name resolution (DNS)
- Clocks

However, the following components are not part of the network library:

- Implementation of network protocols such as HTTP, SMTP, or FTP
- Encryption (SSL or TLS)
- Operating specific multiplexing interfaces, such as `select` or `poll`
- Support for realtime
- TCP/IP protocols like ICMP

Thanks to the network library, you can directly implement an echo server.

A simple echo server

```
template <typename Iterator>
void uppercase(Iterator begin, Iterator end) {
    std::locale loc("");
    for (Iterator iter = begin; iter != end; ++iter)
    *iter = std::toupper(*iter, loc);
}

void sync_connection(tcp::socket& socket) {
    try {
```

[20] https://www.boost.org/doc/libs/1_75_0/doc/html/boost_asio.html

```
10         std::vector<char> buffer_space(1024);
11         while (true) {
12             std::size_t length = socket.read_some(buffer(buffer_space))\
13 ;
14             uppercase(buffer_space.begin(), buffer_space.begin() + leng\
15 th);
16             write(socket, buffer(buffer_space, length));               \
17 
18         }
19     }
20     catch (std::system_error& e) {
21         // ...
22     }
23 }
```

The server gets the client socket `socket socket` (line 8), reads the text (line 12), transforms the text into capital letters (line 13), and sends the text back to the client (line 14).

The boost library has more examples of chat or HTTP servers. Additionally, the server can run synchronously - such as presented in the program - or asynchronously.

8.2 C++23 or Later

It is not sure that the following three features, contracts, reflection, and pattern matching, will be part of C++23. The general idea is, therefore, that they should be part of an upcoming C++ standard. This means that they are partially supported in C++23.

8.2.1 Contracts

Contracts were planned to be the fifth big feature of C++20. Because of design issues, they were removed in the standardization committee meeting in July 2019 in Cologna. At the same time, the study group 21 for contracts[21] 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 in 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:

[21] https://isocpp.org/std/the-committee

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. This 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:

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 can, therefore, not 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 at least to C++23. Herb Sutter's thoughts on Sutter's Mill[22] 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.*"

[22] 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[23] is a powerful tool for reflection, but the proposal P0385[24] for static reflection goes much further.

The following code snippet should give you a first impression on 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:

[23] https://en.cppreference.com/w/cpp/header/type_traits
[24] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0385r2.pdf

```
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.

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<M\
etaClass>
             << '\n';

        do_print_data_members<reflect::get_public_data_members_t<MetaCl\
```

```
ass>>();
    }

    template <typename MetaClass>
    void print_all_data_members(void) {
        using namespace std;

        cout << "All data members of " << reflect::get_base_name_v<Meta\
Class>
             << '\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`[25] or `std::variant`[26] need new ways to work with their elements. Simple `if` or `switch` conditions or functions like `std::apply`[27] or `std::visit`[28] can only provide basic functionality. Pattern matching, heavily used in functional programming, enables the more powerful handling of the new data types.

The following code snippets from the proposal P1371R2[29] on pattern matching compares 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

[25] https://en.cppreference.com/w/cpp/utility/tuple
[26] https://en.cppreference.com/w/cpp/utility/variant
[27] https://en.cppreference.com/w/cpp/utility/apply
[28] https://en.cppreference.com/w/cpp/utility/variant/visit
[29] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1371r2.pdf

`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";
}
```

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[30].

8.3 Further Information about C++23

The proposal P0592R4[31] gives only a rough idea of C++23 and concentrates on the main features. Features such as task blocks[32], unified futures[33], transactional memory[34], or the data-parallel vector library[35], which supports SIMD[36], are not even

[30]https://en.wikipedia.org/wiki/Binary_expression_tree
[31]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html
[32]https://www.modernescpp.com/index.php/task-blocks
[33]https://www.modernescpp.com/index.php/the-end-of-the-detour-unified-futures
[34]https://www.modernescpp.com/index.php/transactional-memory
[35]https://en.cppreference.com/w/cpp/experimental/simd
[36]https://en.wikipedia.org/wiki/SIMD

mentioned. When you want more insight into the future of C++20, you have to study cppreference.com/compiler_support[37] or read the standardization committee papers[38] related to C++23.

[37]https://en.cppreference.com/w/cpp/compiler_support
[38]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/

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 stands for the year and the 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. This is the time I visit cppreference.com/compiler_support[1], search 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 XLC++	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>
```

[2]https://en.cppreference.com/w/cpp/feature_test

```
# endif
#endif

#define COMPILER_FEATURE_VALUE(value) #value
#define COMPILER_FEATURE_ENTRY(name) { #name, COMPILER_FEATURE_VALUE(na\
me) },

#ifdef __has_cpp_attribute
# define COMPILER_ATTRIBUTE_VALUE_AS_STRING(s) #s
# define COMPILER_ATTRIBUTE_AS_NUMBER(x) COMPILER_ATTRIBUTE_VALUE_AS_ST\
RING(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 = nul\
```

```
lptr)
        : 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)
COMPILER_FEATURE_ENTRY(__cpp_lib_quoted_string_io)
COMPILER_FEATURE_ENTRY(__cpp_lib_result_of_sfinae)
COMPILER_FEATURE_ENTRY(__cpp_lib_robust_nonmodifying_seq_ops)
COMPILER_FEATURE_ENTRY(__cpp_lib_shared_timed_mutex)
COMPILER_FEATURE_ENTRY(__cpp_lib_string_udls)
COMPILER_FEATURE_ENTRY(__cpp_lib_transformation_trait_aliases)
COMPILER_FEATURE_ENTRY(__cpp_lib_transparent_operators)
COMPILER_FEATURE_ENTRY(__cpp_lib_tuple_element_t)
```

```
COMPILER_FEATURE_ENTRY(__cpp_lib_tuples_by_type)
};

static CompilerFeature cxx17[] = {
COMPILER_FEATURE_ENTRY(__cpp_aggregate_bases)
COMPILER_FEATURE_ENTRY(__cpp_aligned_new)
COMPILER_FEATURE_ENTRY(__cpp_capture_star_this)
COMPILER_FEATURE_ENTRY(__cpp_constexpr)
COMPILER_FEATURE_ENTRY(__cpp_deduction_guides)
COMPILER_FEATURE_ENTRY(__cpp_enumerator_attributes)
COMPILER_FEATURE_ENTRY(__cpp_fold_expressions)
COMPILER_FEATURE_ENTRY(__cpp_guaranteed_copy_elision)
COMPILER_FEATURE_ENTRY(__cpp_hex_float)
COMPILER_FEATURE_ENTRY(__cpp_if_constexpr)
COMPILER_FEATURE_ENTRY(__cpp_inheriting_constructors)
COMPILER_FEATURE_ENTRY(__cpp_inline_variables)
COMPILER_FEATURE_ENTRY(__cpp_namespace_attributes)
COMPILER_FEATURE_ENTRY(__cpp_noexcept_function_type)
COMPILER_FEATURE_ENTRY(__cpp_nontype_template_args)
COMPILER_FEATURE_ENTRY(__cpp_nontype_template_parameter_auto)
COMPILER_FEATURE_ENTRY(__cpp_range_based_for)
COMPILER_FEATURE_ENTRY(__cpp_static_assert)
COMPILER_FEATURE_ENTRY(__cpp_structured_bindings)
COMPILER_FEATURE_ENTRY(__cpp_template_template_args)
COMPILER_FEATURE_ENTRY(__cpp_variadic_using)
};
static CompilerFeature cxx17lib[] = {
COMPILER_FEATURE_ENTRY(__cpp_lib_addressof_constexpr)
COMPILER_FEATURE_ENTRY(__cpp_lib_allocator_traits_is_always_equal)
COMPILER_FEATURE_ENTRY(__cpp_lib_any)
COMPILER_FEATURE_ENTRY(__cpp_lib_apply)
COMPILER_FEATURE_ENTRY(__cpp_lib_array_constexpr)
COMPILER_FEATURE_ENTRY(__cpp_lib_as_const)
COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_is_always_lock_free)
COMPILER_FEATURE_ENTRY(__cpp_lib_bool_constant)
COMPILER_FEATURE_ENTRY(__cpp_lib_boyer_moore_searcher)
```

```
COMPILER_FEATURE_ENTRY(__cpp_lib_byte)
COMPILER_FEATURE_ENTRY(__cpp_lib_chrono)
COMPILER_FEATURE_ENTRY(__cpp_lib_clamp)
COMPILER_FEATURE_ENTRY(__cpp_lib_enable_shared_from_this)
COMPILER_FEATURE_ENTRY(__cpp_lib_execution)
COMPILER_FEATURE_ENTRY(__cpp_lib_filesystem)
COMPILER_FEATURE_ENTRY(__cpp_lib_gcd_lcm)
COMPILER_FEATURE_ENTRY(__cpp_lib_hardware_interference_size)
COMPILER_FEATURE_ENTRY(__cpp_lib_has_unique_object_representations)
COMPILER_FEATURE_ENTRY(__cpp_lib_hypot)
COMPILER_FEATURE_ENTRY(__cpp_lib_incomplete_container_elements)
COMPILER_FEATURE_ENTRY(__cpp_lib_invoke)
COMPILER_FEATURE_ENTRY(__cpp_lib_is_aggregate)
COMPILER_FEATURE_ENTRY(__cpp_lib_is_invocable)
COMPILER_FEATURE_ENTRY(__cpp_lib_is_swappable)
COMPILER_FEATURE_ENTRY(__cpp_lib_launder)
COMPILER_FEATURE_ENTRY(__cpp_lib_logical_traits)
COMPILER_FEATURE_ENTRY(__cpp_lib_make_from_tuple)
COMPILER_FEATURE_ENTRY(__cpp_lib_map_try_emplace)
COMPILER_FEATURE_ENTRY(__cpp_lib_math_special_functions)
COMPILER_FEATURE_ENTRY(__cpp_lib_memory_resource)
COMPILER_FEATURE_ENTRY(__cpp_lib_node_extract)
COMPILER_FEATURE_ENTRY(__cpp_lib_nonmember_container_access)
COMPILER_FEATURE_ENTRY(__cpp_lib_not_fn)
COMPILER_FEATURE_ENTRY(__cpp_lib_optional)
COMPILER_FEATURE_ENTRY(__cpp_lib_parallel_algorithm)
COMPILER_FEATURE_ENTRY(__cpp_lib_raw_memory_algorithms)
COMPILER_FEATURE_ENTRY(__cpp_lib_sample)
COMPILER_FEATURE_ENTRY(__cpp_lib_scoped_lock)
COMPILER_FEATURE_ENTRY(__cpp_lib_shared_mutex)
COMPILER_FEATURE_ENTRY(__cpp_lib_shared_ptr_arrays)
COMPILER_FEATURE_ENTRY(__cpp_lib_shared_ptr_weak_type)
COMPILER_FEATURE_ENTRY(__cpp_lib_string_view)
COMPILER_FEATURE_ENTRY(__cpp_lib_to_chars)
COMPILER_FEATURE_ENTRY(__cpp_lib_transparent_operators)
COMPILER_FEATURE_ENTRY(__cpp_lib_type_trait_variable_templates)
```

```
COMPILER_FEATURE_ENTRY(__cpp_lib_uncaught_exceptions)
COMPILER_FEATURE_ENTRY(__cpp_lib_unordered_map_try_emplace)
COMPILER_FEATURE_ENTRY(__cpp_lib_variant)
COMPILER_FEATURE_ENTRY(__cpp_lib_void_t)
};

static CompilerFeature cxx20[] = {
COMPILER_FEATURE_ENTRY(__cpp_aggregate_paren_init)
COMPILER_FEATURE_ENTRY(__cpp_char8_t)
COMPILER_FEATURE_ENTRY(__cpp_concepts)
COMPILER_FEATURE_ENTRY(__cpp_conditional_explicit)
COMPILER_FEATURE_ENTRY(__cpp_consteval)
COMPILER_FEATURE_ENTRY(__cpp_constexpr)
COMPILER_FEATURE_ENTRY(__cpp_constexpr_dynamic_alloc)
COMPILER_FEATURE_ENTRY(__cpp_constexpr_in_decltype)
COMPILER_FEATURE_ENTRY(__cpp_constinit)
COMPILER_FEATURE_ENTRY(__cpp_deduction_guides)
COMPILER_FEATURE_ENTRY(__cpp_designated_initializers)
COMPILER_FEATURE_ENTRY(__cpp_generic_lambdas)
COMPILER_FEATURE_ENTRY(__cpp_impl_coroutine)
COMPILER_FEATURE_ENTRY(__cpp_impl_destroying_delete)
COMPILER_FEATURE_ENTRY(__cpp_impl_three_way_comparison)
COMPILER_FEATURE_ENTRY(__cpp_init_captures)
COMPILER_FEATURE_ENTRY(__cpp_modules)
COMPILER_FEATURE_ENTRY(__cpp_nontype_template_args)
COMPILER_FEATURE_ENTRY(__cpp_using_enum)
};
static CompilerFeature cxx20lib[] = {
COMPILER_FEATURE_ENTRY(__cpp_lib_array_constexpr)
COMPILER_FEATURE_ENTRY(__cpp_lib_assume_aligned)
COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_flag_test)
COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_float)
COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_lock_free_type_aliases)
COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_ref)
COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_shared_ptr)
COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_value_initialization)
```

```
COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_wait)
COMPILER_FEATURE_ENTRY(__cpp_lib_barrier)
COMPILER_FEATURE_ENTRY(__cpp_lib_bind_front)
COMPILER_FEATURE_ENTRY(__cpp_lib_bit_cast)
COMPILER_FEATURE_ENTRY(__cpp_lib_bitops)
COMPILER_FEATURE_ENTRY(__cpp_lib_bounded_array_traits)
COMPILER_FEATURE_ENTRY(__cpp_lib_char8_t)
COMPILER_FEATURE_ENTRY(__cpp_lib_chrono)
COMPILER_FEATURE_ENTRY(__cpp_lib_concepts)
COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_algorithms)
COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_complex)
COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_dynamic_alloc)
COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_functional)
COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_iterator)
COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_memory)
COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_numeric)
COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_string)
COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_string_view)
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);
    }
```

```
}

int main() {
    if (print.general_features) show("C++ GENERAL", cxx);
    if (print.cxx11 && print.core_features) show("C++11 CORE", cxx11);
    if (print.cxx14 && print.core_features) show("C++14 CORE", cxx14);
    if (print.cxx14 && print.lib_features ) show("C++14 LIB" , cxx14lib\
);
    if (print.cxx17 && print.core_features) show("C++17 CORE", cxx17);
    if (print.cxx17 && print.lib_features ) show("C++17 LIB" , cxx17lib\
);
    if (print.cxx20 && print.core_features) show("C++20 CORE", cxx20);
    if (print.cxx20 && print.lib_features ) show("C++20 LIB" , cxx20lib\
);
    if (print.cxx23 && print.core_features) show("C++23 CORE", cxx23);
    if (print.cxx23 && print.lib_features ) show("C++23 LIB" , cxx23lib\
);
    if (print.attributes) show("ATTRIBUTES", attributes);
}
```

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:

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

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 that 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**

```
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**

x64 Native Tools Command Prompt for VS 2019

```
C++20 CORE
__cpp_aggregate_paren_init                          ------
__cpp_char8_t                                       201811
__cpp_concepts                                      201811
__cpp_conditional_explicit                          201806
__cpp_consteval                                     ------
__cpp_constexpr                                     201603
__cpp_constexpr_dynamic_alloc                       ------
__cpp_constexpr_in_decltype                         ------
__cpp_constinit                                     ------
__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                         201911
__cpp_using_enum                                    201907
```

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 Callable

see Callable Unit.

10.2 Callable Unit

A callable unit (short callable) 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.

10.3 Concurrency

Concurrency means that the execution of several tasks overlaps. Concurrency is a superset of parallelism.

10.4 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 one point in time should enter a critical section.

10.5 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.

10.6 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.

10.7 Eager Evaluation

In case of eager evaluation, the expression is evaluated immediately. This evaluation strategy is opposite to lazy evaluation. Eager evaluation is often called greedy evaluation.

10.8 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.

10.9 Function Objects

First of all, don't call them functors[1]. That's a *well-defined* term from a branch of mathematics called category theory[2].

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.

```
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 1\
00
```

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 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())`

10.10 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

[1]https://en.wikipedia.org/wiki/Functor
[2]https://en.wikipedia.org/wiki/Category_theory

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
```

10.11 Lazy Evaluation

In the case of lazy evaluation[3], the expression is only evaluated if needed. This evaluation strategy is opposite to eager evaluation. Lazy evaluation is often called call-by-need.

10.12 Lock-free

A non-blocking algorithm is lock-free if there is guaranteed system-wide progress.

10.13 Lost Wakeup

A lost wakeup is a situation in which a thread misses its wake-up notification due to a race condition.

10.14 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

[3] https://en.wikipedia.org/wiki/Lazy_evaluation

10.15 Memory Location

A memory location is according to cppreference.com[4]

- 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.

10.16 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?

10.17 Non-blocking

An algorithm is called non-blocking if 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[5].

10.18 Object

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

10.19 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.

[4] http://en.cppreference.com/w/cpp/language/memory_model
[5] http://jcip.net/

10.20 Predicate

Predicates are callable units that return 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.

10.21 RAII

Resource **A**cquisition **I**s **I**nitialization, in short RAII, 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[6] that handle the lifetime of their elements.

10.22 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 spot. Whether they occur depends on the interleaving of the threads. That means the number of cores, the utilization of your system, or the optimization level of your executable may all be reasons why a race condition appears or does not.

10.23 Regular

In addition to the requirements of the concept `SemiRegular`, the concept `Regular` requires that the type is equally comparable.

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

10.24 Scalar

A scalar type is either an arithmetic type (see `std::is_arithmetic`[7]), an `enum`, a pointer, a member pointer, or a `std::nullptr_t`.

10.25 SemiRegular

A `semiregular` type `X` has to support the Big Six and has to be swappable: `swap(X&, X&)`

10.26 Spurious Wakeup

A spurious wakeup is an erroneous notification. The waiting component of a condition variable or an atomic flag can get a notification, although the notification component didn't send the signal.

10.27 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.

[7] https://en.cppreference.com/w/cpp/types/is_arithmetic

- 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.

10.28 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()`

10.29 Thread

In computer science, a thread of execution is the smallest sequence of programmed instructions that a scheduler can manage independently that is typically a 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[8].

10.30 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.

[8]https://en.wikipedia.org/wiki/Thread_(computing)

10.31 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`, performs conditional inclusion with directives such as `#ifdef`, or `#ifndef`, and expands macros. The compiler uses the translation unit to create an object file.

10.32 Undefined Behavior

All bets are off. Your program can produce the correct result, 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.

Index

Entries in capital letters stand for sections and subsections.