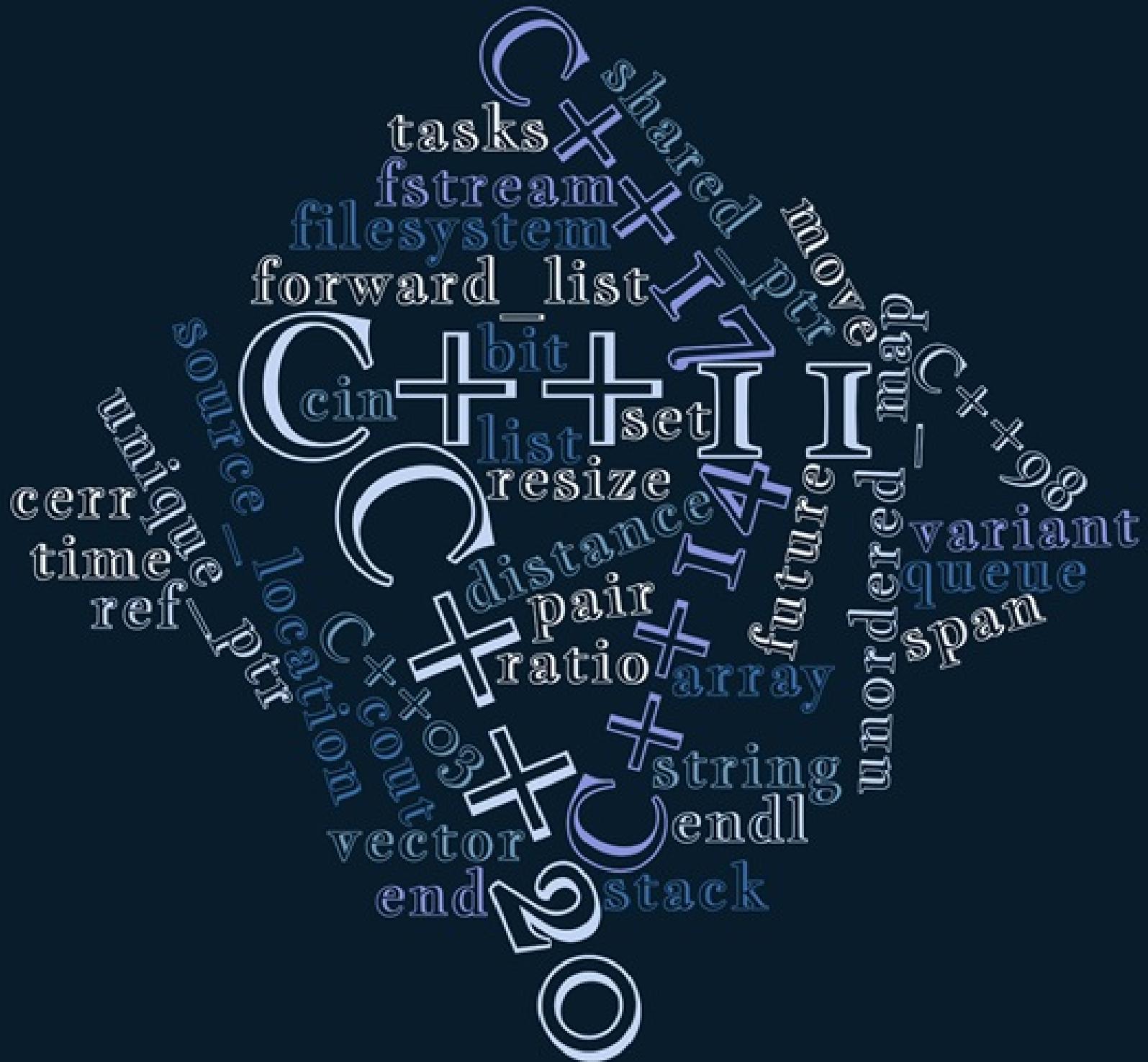


The C++ Standard Library

Third Edition includes C++20



Rainer
Grimm

The C++ Standard Library

What every professional C++ programmer should know about the C++ standard library.

Rainer Grimm

This book is for sale at <http://leanpub.com/cpplibrary>

This version was published on 2020-12-26



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.

* * * * *

© 2016 - 2020 Rainer Grimm

Table of Contents

[Reader Testimonials](#)

[English Edition](#)

[German Edition](#)

[Introduction](#)

[About Me](#)

[Purpose of this Book](#)

[Index](#)

[Conventions](#)

[Source Examples](#)

[Source Code](#)

[Value versus Object](#)

[Acknowledgements](#)

[Further Information](#)

[C++ versus C++11](#)

[1. The Standard Library](#)

[The History](#)

[Overview](#)

[Application of Libraries](#)

[2. Utilities](#)

[Useful Functions](#)

[Save Comparison of Integers](#)

[Adaptors for Functions](#)

[Pairs](#)

[Tuples](#)

[Reference Wrappers](#)

[Smart Pointers](#)

[Type Traits](#)

[Time Library](#)

[std::any, std::optional, and std::variant](#)

[3. Interface of All Containers](#)

[Create and delete](#)

[Size](#)
[Access](#)
[Assign and Swap](#)
[Compare](#)
[Erasure](#)

[4. Sequence Containers](#)

[Arrays](#)
[Vectors](#)
[Deques](#)
[Lists](#)
[Forward Lists](#)

[5. Associative Containers](#)

[Overview](#)
[Ordered Associative Containers](#)
[Unordered Associative Containers](#)

[6. Adaptors for Containers](#)

[Stack](#)
[Queue](#)
[Priority Queue](#)

[7. Views on Contiguous Sequences](#)

[8. Iterators](#)

[Categories](#)
[Iterator Creation](#)
[Useful Functions](#)
[Adaptors](#)

[9. Callable Units](#)

[Functions](#)
[Function Objects](#)
[Lambda Functions](#)

[10. Algorithms](#)

[Conventions](#)
[Iterators are the Glue](#)
[Sequential, Parallel, or Parallel Execution with Vectorisation](#)

[for_each](#)
[Non-Modifying Algorithms](#)
[Modifying Algorithms](#)
[Partition](#)
[Sort](#)
[Binary Search](#)
[Merge Operations](#)
[Heaps](#)
[Min and Max](#)
[Permutations](#)
[Numeric](#)

[11. Ranges](#)

[Range](#)
[View](#)
[Direct on the Containers](#)
[Function Composition](#)
[Lazy Evaluation](#)

[12. Numeric](#)

[Random Numbers](#)
[Numeric Functions Inherited from C](#)
[Mathematical Constants](#)

[13. Strings](#)

[Create and Delete](#)
[Conversion Between C++ and C Strings](#)
[Size versus Capacity](#)
[Comparison](#)
[String Concatenation](#)
[Element Access](#)
[Input and Output](#)
[Search](#)
[Check for a Prefix or a Suffix](#)
[Modifying Operations](#)
[Numeric Conversions](#)

[14. String Views](#)

[Create and Initialise](#)

[Non-modifying operations](#)

[Modifying operations](#)

[15. Regular Expressions](#)

[Character Types](#)

[Regular Expression Objects](#)

[The Search Result match_results](#)

[Match](#)

[Search](#)

[Replace](#)

[Format](#)

[Repeated Search](#)

[16. Input and Output Streams](#)

[Hierarchy](#)

[Input and Output Functions](#)

[Streams](#)

[User-defined Data Types](#)

[17. Formatting Library](#)

[Syntax](#)

[Format specification](#)

[User-defined formatter](#)

[18. Filesystem](#)

[Classes](#)

[Non-member functions](#)

[File types](#)

[19. Multithreading](#)

[Memory Model](#)

[Atomic Data Types](#)

[Threads](#)

[Stop Token](#)

[Shared Variables](#)

[Thread Local Data](#)

[Condition Variables](#)

[Semaphores](#)

[Coordination Types](#)

[Tasks](#)

[**20. Coroutines**](#)

[Awaitables](#)

[An Infinite Data Stream with `co_yield`](#)

[Index](#)

Reader Testimonials

English Edition

Rick Audet

Senior Engineer, Dolby Laboratories

"Arguably the quickest and easiest way to get up to speed on the most important parts of the C++ standard library. Recommended for any modern C++ programmer."

German Edition

Odin Holmes

CEO/CTO at Auto-Intern GmbH

"Das Buch beinhaltet, wie der Name schon sagt, eine recht ausführliche Beschreibung der STL. Dabei merkt Mann deutlich dass der Autor auch selber auf hohem Niveau programmiert. Es gibt oft den ein oder andere Tipp oder Wink in die richtige Richtung die bei Büchern von Berufsauteuren oft fehlen. Z.B. die Aussage dass std::vector für 95% aller Fälle die beste Wahl ist oder dass std::async meistens die erste Wahl sein sollte lenkt der Leser geschickt in die richtige Richtung.

Auch die Auswahl an Komponente aus der STL ist sehr gut getroffen (keiner kann in ein kürzen Buch die ganze STL beschreiben). Oft sehe ich, vor allem in Deutschsprachige Literatur, dass die Auswahl eher auf Komponente trifft die leicht zu beschreiben sind und nicht auf die Nützlichen. Eine gute und dennoch kürze Beschreibung vom std::regex z.B. ist weiß Gott nicht einfach aber in diesem Fall ist es der Autor sehr gelungen."

Ramon Wartala

Director Technology at Performance Media GmbH

"Die 215 Seiten plus Index des 'C++ kurz & gut' vom Autor Rainer Grimm stellen ein gelungene Destillat viel umfangreicherer Texte zum Thema da. So nimmt das Kapitel über die klassischen Algorithmen der Standardbibliothek ganze 131 Seiten ein. Selbst kurze Beispiele für die Anwendung der wichtigsten Bestandteile der Standardbibliothek passen eng gedruckt in das schmale Büchlein. Auch wenn heute Tools wie Dash oder entsprechend ausgestattete IDEs, mehr und mehr den Platz derartiger Desktop-Referenzen einnehmen, ist das 'kurz & gut' zu C++ Standardbibliothek ein leichter und mobiler Begleiter für jeden C++ Entwickler. Und als Kindle Version um so bequemer mitzunehmen."

Introduction

About Me



Rainer Grimm

I work as a software architect, team lead, and instructor since 1999. In 2002, I created a further education round at my company. I give seminars since 2002. My first seminars were about proprietary management software, but seminars for Python and C++ followed immediately. 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 Modernes Cpp and the German blog, hosted by Heise Developer.

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

Purpose of this Book

The C++ Standard Library is a quick reference to the standard library of the current C++ standard C++20 [ISO/IEC 14882:2020](#). C++20 has more than 1800 pages and is the next big C++ standard after C++17. C++17 is neither a big nor a

small C++ standard. C++14 is a small addition to C++11. C++11 had more than 1,300 pages and was published 2011. That was 13 years after the first and only C++ standard C++98. Of course, there is also C++03, published in 2003. But C++03 is considered a bug fix release.

The goal of this quick reference is to provide a concise reference to the C++ standard library. This book assumes that you are familiar with C++. If so you will get the most benefit out of this book. If C++ is new to you, you should start with a textbook about core C++. Once you have mastered a textbook about the core language, you can make your next big step by reading this book. To make your job easier, I have provided a lot of short code snippets to connect theory and practice.

Index

The book should be a reference for C++, and should, therefore, have an index. Leanpub does not support the creation of an index. So I've made it based on regular expressions, naming conventions, a lot of python magic - don't blame me - and a long, long table which I had to split for each page. Here is the problem. The index is only fully available in the pdf format of the book.

Conventions

I promise only a few conventions.

Special Fonts

Italic

I use *Italic* if something is essential.

Monospace

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

Special Boxes

I use boxes for unique information, tips, and warning.



Information headline

Information text.



Tip headline

Tip description.



Warning headline

Warning description.

Source Examples

I dislike using directives and declarations because they hide the namespace of the library functions, but because of the limited length of a page, I have to use them from time to time. I use them in such a way that the origin can always be deduced from the using *directive* (using namespace std;) or the *using declaration* (using std::cout;).

Only header files of the featured functionality are shown in the code snippets. True or false is displayed in the output code snippets for boolean values; and [std::boolalpha](#) is not used.

Source Code

To be concise, I only present short code snippets in this book. The name of the entire program is in the first line of the code snippet.

Value versus Object

I call instances of fundamental data types *values*, which C++ inherited from C. Instances of more advanced types which often consist of fundamental types, are called *objects*. Objects are typically instances of *user-defined* types or containers.

Acknowledgements

At first Alexandra Follenius, who was the lector at O'Reilly for the German book [C++ Standardbibliothek](#). You may have guessed that the German book is the ancestor of this book. For my book *C++ Standardbibliothek* Karsten Ahnert, Guntram Berti, Dmitry Ganyushin, Sven Johannsen, Torsten Robitzki, Bart Vandewoestyne, and Felix Winter were very valuable proofreaders. A lot of thanks to all of them.

For translating this book to English, I started a request in my English blog: [www.ModernesCpp.com](#). I received a much higher response than I expected. Special thanks to all of you, including my son Marius, who was the first proofreader.

Here are the names in alphabetic order: Mahesh Attarde, Rick Audet, Pete Barrow, Michael Ben-David, Dave Burns, Alvaro Fernandez, Juliette Grimm, George Haake, Clare Macrae, Arne Mertz, Ian Reeve, Jason Turner, Bart Vandewoestyne, Ivan Vergiliev, and Andrzej Warzynski.

Further Information

The idea of the book is quite easy to paraphrase: "What every professional C++ programmer should know about the C++ standard library." Because of this intention, I left a lot of answers unanswered; therefore, I provide you with the links to the details at the very beginning of each new topic. The link will be referring to the excellent online resource [www.cppreference.com](#).

C++ versus C++11

Who can characterise C++11 better than Bjarne Stroustrup, creator of C++:

Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever. (Bjarne Stroustrup, <http://www.stroustrup.com/C++11FAQ.html>)

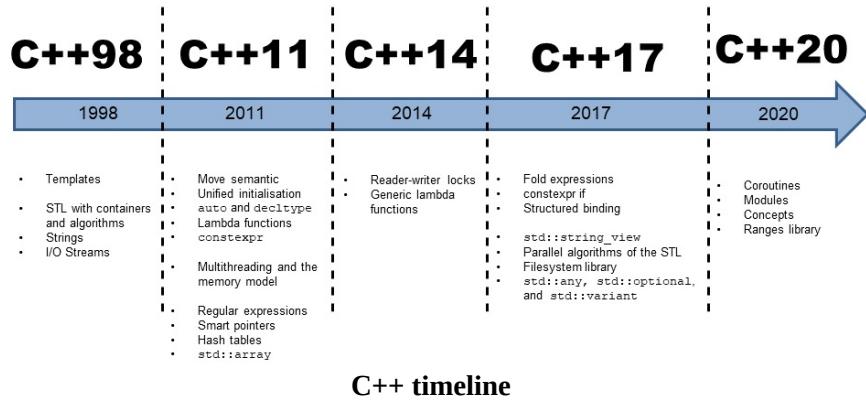
Bjarne Stroustrup is right. C++11 feels like a new language because it has a lot to offer in addition to classic C++. This is true for the core language and is even more true for the improved and extended standard library. The regular expression library for the manipulation of text, the type-trait library to get, compare or manipulate types, the new random numbers library or the chrono library are all new with C++11. But that's not all. There are the smart pointers for automatic memory management and the new containers `std::array` and `std::tuple`, which are further improved in C++14. C++11 is for the first time aware of multiple threads and offers a multithreading library.

1. The Standard Library

The C++ standard library consists of many components. This chapter serves two purposes. It should give you a quick overview of the components and a first idea how to use them.

The History

C++ and therefore the standard library have a long history. C++ started in the 1980s of the last millennium and ended now in 2020. Anyone who knows about software development knows how fast our domain evolves. So 30 years is a very long period. You may not be so astonished that the first components of C++, like I/O streams, were designed with a different mindset than the modern Standard Template Library (STL). This evolution in the area of software development in the last 30 years, which you can observe in the C++ standard library, is also an evolution in the way software problems are solved. C++ started as an object-oriented language, then incorporated generic programming with the STL and now has adopted a lot of functional programming ideas.



The first C++ standard library from 1998 had three components. Those were the previously mentioned I/O streams, mainly for file handling, the string library and the Standard Template Library. The Standard Template Library facilitates the transparent application of algorithms on containers.

The history continues in the year 2005 with Technical Report 1 (TR1). The extension to the C++ library ISO/IEC TR 19768 was not an official standard, but almost all of the components became part of C++11. These were, for example, the libraries for regular expressions, smart pointers, hash tables, random numbers and time, based on the boost libraries (<http://www.boost.org/>).

In addition to the standardisation of TR1, C++11 got one new component: the multithreading library.

C++14 was only a small update to the C++11 standard. Therefore only a few improvements to the already existing libraries for smart pointers, tuples, type traits and multithreading were added.

C++17 includes libraries for the file system and the two new data types `std::any` and `std::optional`.

C++20 has four outstanding features: concepts, ranges, coroutines, modules

Overview

As C++11 has a lot of libraries, it is often not so easy to find the convenient one for each use case.

Utilities

Utilities are libraries which have a general focus and therefore can be applied in many contexts.

Examples of utilities are functions to calculate the minimum or maximum of values or functions, the midpoint of two values, or to swap or move values.

Thanks to save comparison of integers, integral promotion does not kick in.

Other utilities are std::function, std::bind, or std::bind_front. With `std::bind` or `std::bind_front` you can easily create new functions from existing ones. To bind them to a variable and invoke them later, you have `std::function`.

With std::pair and its generalisation std::tuple you can create heterogeneous pairs and tuples of arbitrary length.

The [reference wrappers](#) `std::ref` and `std:: cref` are pretty handy. One can use them to create a reference wrapper for a variable, which for `std:: cref` is `const`.

Of course, the highlights of the utilities are the [smart pointers](#). They allow explicit automatic memory management in C++. You can model the concept of explicit ownership with `std::unique_ptr` and model shared ownership with `std::shared_ptr`. `std::shared_ptr` uses reference counting for taking care of its resource. The third one, `std::weak_ptr`, helps to break the cyclic dependencies among `std::shared_ptr`s, the classic problem of reference counting.

The [type traits](#) library is used to check, compare and manipulate type information at compile time.

The [time library](#) is an import addition of the new multithreading capabilities of C++. But it is also quite handy to make performance measurements and includes support for [calender](#) and [time zone](#).

With [`std::any`](#), [`std::optional`](#), and [`std::variant`](#), we get with C++17 three special datatypes that can have any, an optional value, or a variant of values.

The Standard Template Library



The three components of the STL

The Standard Template Library (STL) consists of three components from a bird's-eye view. Those are containers, algorithms that run on the containers, and iterators that connect both of them. This abstraction of generic programming enables you to combine algorithms and containers in a unique way. The containers have only minimal requirements for their elements.

The C++ Standard Library has a rich collection of containers. From a bird's eye we have sequence and associative containers. Associative containers can be classified as ordered or unordered associative containers.

Each of the [sequence containers](#) has a unique domain, but in 95 % of the use cases `std::vector` is the right choice. `std::vector` can dynamically adjust its size, automatically manages its memory and provides you with outstanding performance. In contrast, `std::array` is the only sequence container that cannot adjust its size at runtime. It is optimised for minimal memory and performance overhead. While `std::vector` is good at putting new elements at its end, you should use `std::deque` to put an element also at the beginning. With `std::list` being a doubly-linked list and `std::forward_list` as a singly linked list, we have two additional containers that are optimised for operations at arbitrary positions in the container, with high performance.

[Associative containers](#) are containers of key-value pairs. They provide their values by their respective key. A typical use case for an associative container is a phone book, where you use the key *family name* to retrieve the value *phone number*. C++ has eight different associative containers. On one side there are the associative containers with ordered keys: `std::set`, `std::map`, `std::multiset` and `std::multimap`. On the other side there are the unordered associative containers: `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset` and `std::unordered_multimap`.

Let's look first at the [ordered associative containers](#). The difference between `std::set` and `std::map` is that the former has no associated value. The difference between `std::map` and `std::multimap` is, that the latter can have more than one identical key. This naming conventions also holds for the [unordered associative containers](#), which have a lot in common with the ordered ones. The key difference is the performance. While the ordered associative containers have an access time depending logarithmically on the number of elements, the unordered associative containers allow constant access time. Therefore the access time of the unordered associative containers is independent of their size. The same rule holds true for `std::map` as it does for `std::vector`. In 95 % of all use cases `std::map` should be your first choice for an associative container because the keys are sorted.

[Container adapters](#) provide a simplified interface to the sequence containers. C++ has `std::stack`, `std::queue` and `std::priority_queue`.

`std::span` is a view on a [contiguous sequence of elements](#). C-array, `std::array`, `std::vector`, or `std::string` support views. A view is never an owner.

[Iterators](#) are the glue between the containers and the algorithms. The container creates them. As generalised pointers, you can use them to iterate forward and backwards or to an arbitrary position in the container. The type of iterator you get depends on the container. If you use an iterator adapter, you can directly access a stream.

The STL gives you more than 100 [algorithms](#). By specifying the [execution policy](#), you can run most of the algorithms sequential, parallel, or parallel and vectorised. Algorithms operate on elements or a range of elements. Two iterators define a range. The first one defines the beginning, the second one, called end iterator, defines the end of the range. It's important to know that the end iterator points to *one element past the end of the range*.

The algorithms can be used in a wide range of applications. You can find elements or count them, find ranges, compare or transform them. There are algorithms to generate, replace or remove elements from a container. Of course, you can sort, permute or partition a container or determine the minimum or maximum element of it. A lot of algorithms can be further customised by *callables* like functions, function objects or lambda-functions. The *callables* provide special criteria for search or elements transformation. They highly increase the power of the algorithm.

The algorithms of the [ranges library](#) are lazy, can work directly on the container and can easily be composed. They extend C++ with functional ideas.

Numeric

There are two libraries for numerics in C++: the random numbers library and the mathematical functions, which C++ inherited from C.

The [random numbers library](#) consists of two parts. On one side there is the random number generator, on the other side, the distribution of the generated random numbers. The random number generator generates a stream of numbers between a minimum and a maximum value, which the random number distribution maps onto the concrete distribution.

Because of C, C++ has a lot of [mathematical standard functions](#). For example there are logarithmic, exponential and trigonometric functions.

C++ supports basic and advanced [mathematical constants](#) such as e , π , or ϕ .

Text Processing

With strings and regular expressions, C++ has two powerful libraries to process text.

[`std::string`](#) possesses a rich collection of methods to analyze and modify its text. Because it has a lot in common with a `std::vector` of characters, the algorithms of the STL can be used for `std::string`. `std::string` is the successor of the C string but a lot easier and safer to use. C++ strings manage their own memory.

In contrast to a `std::string` a [`std::string_view`](#) is quite cheap to copy. A `std::string_view` is a non-owning reference to a `std::string`.

[Regular expression](#) is a language for describing text patterns. You can use regular expressions to determine whether a text pattern is present once or more times in a text. But that's not all. Regular expressions can be used to replace the content of the matched patterns with a text.

Input and Output

[I/O streams library](#) is a library, present from the start of C++, that allows communication with the outside world.

Communication means in this concrete case that the extraction operator (`>>`) enables it to read formatted or unformatted data from the input stream, and the insertion operator (`<<`) enables it to write the data on the output stream. Data can be formatted using *manipulators*.

The stream classes have an elaborate class hierarchy. Two stream classes are significant: First, string streams allow you to interact with strings and streams. Second, file streams allow you to read and write files easily. The state of streams is kept in flags, which you can read and manipulate.

By overloading the input operator and output operator, your class can interact with the outside world like a fundamental data type.

The [formatting library](#) provides a safe and extensible alternative to the `printf` family and extends the I/O streams library.

In contrast to the I/O streams library, [filesystem library](#) was added to the C++-Standard with C++17. The library is based on the three concepts file, file name and path. Files can be directories, hard links, symbolic links or regular files. Paths can be absolute or relative.

The filesystem library supports a powerful interface for reading and manipulating the filesystem.

Multithreading

C++ gets with the 2011 published C++ standard a multithreading library. This library has basic building blocks like atomic variables, threads, locks and condition variables. That's the base on which future C++ standards can build higher abstractions. But C++11 already knows tasks, which provide a higher abstraction than the cited basic building blocks.

At a low level, C++11 provides for the first time a [memory model](#) and atomic variables. Both components are the foundation for *well-defined* behaviour in multithreading programming.

A new [thread](#) in C++ will immediately start its work. It can run in the foreground or background and gets its data by copy or reference. Thanks to the [stop token](#), you can interrupt the improved thread [std::jthread](#).

The access of shared variables between threads has to be coordinated. This coordination can be done in different ways with mutexes or locks. But often it's sufficient to protect the initialisation of the data as it will be immutable during its lifetime.

Declaring a variable as [thread-local](#) ensures that a thread get its own copy, so there is no conflict.

[Condition variables](#) are a classic solution to implement sender-receiver workflows. The key idea is that the sender notifies the receiver when it's done with its work, so the receiver can start.

[Semaphores](#) are a synchronisation mechanism used to control concurrent access to a shared resource. A semaphore has a counter that is bigger than zero. Acquiring the semaphore decreases the counter and releasing the semaphore

increases the counter. A thread can only acquire the ressource when the counter is greater then zero.

Similar to semaphores, `std::latch` and `std::barrier` are [coordination types](#) which enable some threads to block until a counter becomes zero. In contrast to a `std::barrier`, you can reuse a `std::latch` for a new iteration and adjust its counter for this new iteration.

[Tasks](#) have a lot in common with threads. But while a programmer explicitly creates a thread, a task will be implicitly created by the C++ runtime. Tasks are like data channels. The promise puts data into the data channel, the future picks the value up. The data can be a value, an exception or simply a notification.

[Coroutines](#) are functions that can suspend and resume their execution while keeping their state. Coroutines are the usual way to write [event-driven applications](#). The event-driven application can be simulations, games, servers, user interfaces, or even algorithms. Coroutines are typically used for [cooperative multitasking](#). The key to cooperative multitasking is that each task takes as much time as it needs.

Application of Libraries

To use a library in a file you have to perform three steps. At first, you have to include the header files with the `#include` statement, so the compiler knows the names of the library. Because the names of the C++ standard library are in the namespace `std`, you can use them in the second step fully qualified or you have to import them in the global namespace. The third and final step is to specify the libraries for the linker to get an executable. This third step is often not necessary. The three steps are explained below.

Include Header Files

The preprocessor includes the file, following the `#include` statement. That is most of the time a header file. The header files will be enclosed in angle brackets:

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



Specify all necessary header files

The compiler is free to add additional headers to the header files. So it may happen that your program has all necessary headers although you didn't specify all of them. It's not recommended to rely on this feature. All needed headers should always be explicitly specified, otherwise, a compiler upgrade or code porting may provoke a compilation error.

Usage of Namespaces

If you use qualified names, you have to use them exactly as defined. For each namespace you must put the scope resolution operator `:::`. More libraries of the C++ standard library use nested namespaces.

```
#include <iostream>
#include <chrono>
...
std::cout << "Hello world:" << std::endl;
auto timeNow= std::chrono::system_clock::now();
```

Unqualified Use of Names

You can use names in C++ with the `using` declaration and the `using` directive.

Using Declaration

A `using` declaration adds a name to the visibility scope, in which you applied the `using` declaration:

```
#include <iostream>
#include <chrono>
...
using std::cout;
using std::endl;
using std::chrono::system_clock;
...
cout << "Hello world:" << endl; // unqualified name
auto timeNow= now();           // unqualified name
```

The application of a `using` declaration has the following consequences:

- An ambiguous lookup and therefore a compiler error occur, if the same name was declared in the same visibility scope.
- If the same name was declared in a surrounding visibility scope, it will be hidden by the `using` declaration.

Using Directive

The `using` directive permits it to use all names of a namespace without qualification.

```
#include <iostream>
#include <chrono>
...
using namespace std;
...
cout << "Hello world:" << endl;           // unqualified name
auto timeNow= chrono::system_clock::now(); // partially qualified name
```

A `using` directive adds no name to the current visibility scope, it only makes the name accessible. That implies:

- An ambiguous lookup and therefore a compiler error occur, if the same name was declared in the same visibility scope.
- A name in the local namespace hides a name declared in a surrounding namespace.
- An ambiguous lookup and therefore a compiler error occurs if the same name get visible from different namespaces or if a name in the namespace hides a name in the global scope.



Use `using` directives with great care in source files

`using` directives should be used with great care in source files, because by the directive `using namespace std` all names from `std` becomes visible. That includes names, which accidentally hide names in the local or surrounding namespace.

Don't use `using` directives in header files. If you include a header with `using namespace std` directive, all names from `std` become visible.

Namespace Alias

A namespace alias defines a synonym for a namespace. It's often convenient to use an alias for a long namespace or nested namespaces:

```
#include <chrono>
...
namespace sysClock= std::chrono::system_clock;
auto nowFirst= sysClock::now();
auto nowSecond= std::chrono::system_clock::now();
```

Because of the namespace alias, you can address the `now` function qualified and with the alias. A namespace alias must not hide a name.

Build an Executable

It is only seldom necessary to link explicitly against a library. That sentence is platform dependent. For example, with the current `g++` or `clang++` compiler, you have to link against the `pthread` library to get the multithreading functionality.

```
g++ -std=c++14 thread.cpp -o thread -pthread
```

2. Utilities

Utilities are useful tools which can be used in many different contexts. They are not bound to a typical domain. That sentence holds for the functions and libraries of this chapter. I explain to you functions which you can apply on arbitrary values or functions which you can use to create new functions and binds them to variables. You can store arbitrary values of arbitrary types in pairs and tuples, or you can build references to arbitrary values. Smart pointers are your tool to implement automatic memory management in C++. To get information about your types, use the type-trait library.

Useful Functions

The many variations of the `min`, `max` and `minmax` functions are applicable on values and initialiser lists. These functions need the header `<algorithm>`. Nearly the same holds for the functions `std::move`, `std::forward` and `std::swap`. You can apply them to arbitrary values. These three functions are defined in the header `<utility>`.

`std::min`, `std::max` and `std::minmax`

The functions `std::min`, `std::max` and `std::minmax`, defined in the header `<algorithm>`, act on values and initialiser lists and give you the requested value back as result. In the case of `std::minmax`, you get a `std::pair`. The first element of the pair is the minimum, the second the maximum of the values. By default, the less operator (`<`) is used, but you can specify your comparison operator. This function needs two arguments and returns a boolean. Functions either return true or false are called predicates.

The functions `std::min`, `std::max`, and `std::minmax`

```
// minMax.cpp
...
#include <algorithm>
...
using std::cout;
...
cout << std::min(2011, 2014);                                // 2011
cout << std::min({3, 1, 2011, 2014, -5});                  // -5
cout << std::min(-10, -5, [](int a, int b)
{ return std::abs(a) < std::abs(b); }); // -5
```

```

auto pairInt= std::minmax(2011, 2014);
auto pairSeq= std::minmax({3, 1, 2011, 2014, -5});
auto pairAbs= std::minmax({3, 1, 2011, 2014, -5}, [](int a, int b)
    { return std::abs(a) < std::abs(b); });

cout << pairInt.first << "," << pairInt.second; // 2011,2014
cout << pairSeq.first << "," << pairSeq.second; // -5,2014
cout << pairAbs.first << "," << pairAbs.second; // 1,2014

```

The table provides an overview of the functions `std::min`, `std::max` and `std::minmax`

Function	Description
<code>min(a, b)</code>	Returns the minimal value of a and b.
<code>min(a, b, comp)</code>	Returns the minimal value of a and b according to the predicate <code>comp</code> .
<code>min(initialiser list)</code>	Returns the minimal value of the initialiser list.
<code>min(initialiser list, comp)</code>	Returns the minimal value of the initialiser list according to the predicate <code>comp</code> .
<code>max(a, b)</code>	Returns the maximal value of a and b.
<code>max(a, b, comp)</code>	Returns the maximal value of a and b according to the predicate <code>comp</code> .
<code>max(initialiser list)</code>	Returns the maximal value of the initialiser list.
<code>max(initialiser list, comp)</code>	Returns the maximal value of the initialiser list according to the predicate <code>comp</code> .
<code>minmax(a, b)</code>	Returns the minimal and maximal value of a and b.
<code>minmax(a, b, comp)</code>	Returns the minimal and maximal value of a and b according to the predicate <code>comp</code> according to the

predicate `comp`.

`minmax(initialiser list)` Returns the minimal and maximal value of the initialiser list.

`minmax(initialiser list, comp)` Returns the minimal and maximal value of the initialiser list according to the predicate `comp`.

std::midpoint and std::lerp

The function `std::midpoint(a, b)` calculates the midpoint between `a` and `b`. `a` and `b` can be integers, floating-point numbers, or pointers. If `a` and `b` are pointers, they have to point to the same array object. The function `std::midpoint` requires the header `<numeric>`.

The function `std::lerp(a, b, t)` calculates the linear arithmetic of two numbers. It requires the header `<cmath>`. The return value is $a + t(b - a)$.

std::midpoint and std::lerp

```
// midpointLerp.cpp

#include <cmath>
#include <numeric>

...
std::cout << "std::midpoint(10, 20): " << std::midpoint(10, 20) << std::endl;

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";
}
```



```
x64 Native Tools Command Prompt for VS 2019
C:\Users\rainer>midpointLerp.exe
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

C:\Users\rainer>
```

std::move

The function `std::move`, defined in the header `<utility>`, empowers the compiler to move it's resource. In the so-called *move semantic*, the values from the source object are moved to the new object. Afterwards the source is in a *well-defined* but not specified state. Most of the times that is the default state of the source. By using `std::move`, the compiler converts the source `arg` to a rvalue reference: `static_cast<std::remove_reference<decltype(arg)>::type&&>(arg)`. If the compiler can not apply the *move semantic*, it falls back to the *copy semantic*:

```
#include <utility>
...
std::vector<int> myBigVec(10000000, 2011);
std::vector<int> myVec;

myVec = myBigVec;           // copy semantic
myVec = std::move(myBigVec); // move semantic
```



To move is cheaper than to copy

The move semantic has two advantages. Firstly, it is often a good idea to use cheap moving instead of expensive copying. So there is no superfluous allocation and deallocation of memory necessary. Secondly, there are objects, which can not be copied. E.g. a thread or a lock.

`std::forward`

The function `std::forward`, defined in the header `<utility>`, empower you to write function templates, which can identically forward their arguments. Typical use cases for `std::forward` are factory functions or constructors. Factory functions are functions which create an object and must therefore identically pass the arguments. Constructors often use their arguments to initialise their base class with identical arguments. So `std::forward` is the perfect tool for authors of generic libraries:

Perfect forwarding

```
// forward.cpp
...
#include <utility>
...
using std::initializer_list;

struct MyData{
    MyData(int, double, char){};
};
```

```

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

...

int a= createT<int>();
int b= createT<int>(1);

std::string s= createT<std::string>("Only for testing.");
MyData myData2= createT<MyData>(1, 3.19, 'a');

typedef std::vector<int> IntVec;
IntVec intVec= createT<IntVec>(initialiser_list<int>({1, 2, 3}));

```

The function template `createT` has to take their arguments as a [universal reference](#): `Args&&... args``. A universal reference or also called forwarding reference is a rvalue reference in a type deduction context.



std::forward in combination with variadic templates allows completely generic functions

If you use `std::forward` together with variadic templates, you can define completely generic function templates. Your function template can accept an arbitrary number of arguments and forward them unchanged.

std::swap

With the function [`std::swap`](#) defined in the header `<utility>`, you can easily swap two objects. The generic implementation in the C++ standard library internally uses the function `std::move`.

Move-semantic with `std::swap`

```

// swap.cpp
...
#include <utility>
...
template <typename T>
inline void swap(T& a, T& b) noexcept {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}

```

Save Comparison of Integers

The functions of the `<utility>` header `std::cmp_equal`, `std::cmp_not_equal`, `std::cmp_less`, `std::cmp_greater`, `std::cmp_less_equal`, and `std::cmp_greater_equal` provide safe comparison of integers. Safe comparison means, that the comparison of a negative signed integer always compares less than an unsigned integer, and that the comparison of values other than a signed or unsigned integer gives a compile time error.



Integer conversion with builtin integers

The following code snippet exemplifies the issue of signed/unsigned comparison.

```
-1 < 0u; // true
std::cmp_greater( -1, 0u); // false
```

`-1` as a signed integer is promoted to a very large unsigned type which causes the surprising result.

Adaptors for Functions

The two functions `std::bind`, `std::bind_front` and `std::function` fit very well together. While `std::bind` or `std::bind_front` enables you to create new function objects on the fly, `std::function` takes these temporary function objects and binds them to a variable. The function `std::bind` is more powerful than `std::bind_front` because `std::bind` allows it to bind the arguments to an arbitrary position. These functions are powerful tools from functional programming and require the header `<functional>`.

Creating and binding function objects

```
// bindAndFunction.cpp
...
#include <functional>
...
// for placeholder _1 and _2
using namespace std::placeholders;

using std::bind;
using std::bind_front;
using std::function
...
double divMe(double a, double b){ return a/b; }
function<double(double, double)> myDiv1 = bind(divMe, _1, _2);
function<double(double)> myDiv2 = bind(divMe, 2000, _1);
function<double(double)> myDiv3 = bind_front(divMe, 2000);

divMe(2000, 10) == myDiv1(2000, 10) == myDiv2(10) == myDiv3(10);
```



std::bind, std::bind_front, and std::function are mostly superfluous

std::bind and std::function, which where part of [TR1](#) and std::bind_front, which is part of C++20, are mostly not necessary any more in C++. You can use lambdas instead of std::bind or std::bind_front and most often you can use the automatic type deduction with auto instead of std::function.

std::bind

Because of [std::bind](#), you can create function objects in a variety of ways:

- bind the arguments to an arbitrary position,
- change the order of the arguments,
- introduce placeholders for arguments,
- partially evaluate functions,
- invoke the newly created function objects, use them in the algorithm of the STL or store them in std::function.

std::bind_front

[std::bind_front](#) creates an callable wrapper from a callable. A call std::bind_front(func, arg ...) binds all arguments arg to the front of func and returns a callable wrapper.

std::function

[std::function](#) can store arbitrary callables in variables. It's a kind of polymorphic function wrapper. A callable may be a lambda function, a function object or a function. std::function is always necessary and can't be replaced by auto, if you have to specify the type of the callable explicitly.

A dispatch table with std::function

```
// dispatchTable.cpp
...
#include <functional>
...
using std::make_pair;
using std::map;

map<const char, std::function<double(double, double)>> tab;
```

```

tab.insert(make_pair('+', [](double a, double b){ return a + b; }));
tab.insert(make_pair('-', [](double a, double b){ return a - b; }));
tab.insert(make_pair('*', [](double a, double b){ return a * b; }));
tab.insert(make_pair('/', [](double a, double b){ return a / b; }));

std::cout << tab['+'](3.5, 4.5); // 8
std::cout << tab['-'](3.5, 4.5); // -1
std::cout << tab['*'](3.5, 4.5); // 15.75
std::cout << tab['/'](3.5, 4.5); // 0.777778

```

The type parameter of `std::function` defines the type of callables `std::function` will accept.

Function type	Return type	Type of the arguments
double(double, double)	double	double
int()	int	
double(int, double)	double	int, double
void()		

Pairs

With [`std::pair`](#), you can build pairs of arbitrary types. The class template `std::pair` needs the header `<utility>`. `std::pair` has a default, copy and move constructor. Pair objects can be swapped: `std::swap(pair1, pair2)`.

Pairs will often be used in the C++ library. For example, the function [`std::minmax`](#) returns its result as a pair, the [`associative container`](#) `std::map`, `std::unordered_map`, `std::multimap` and `std::unordered_multimap` manage their key/value association in pairs.

To get the elements of a pair `p`, you can either access it directly or via an index. So, with `p.first` or `std::get<0>(p)` you get the first, with `p.second` or `std::get<1>(p)` you get the second element of the pair.

Pairs support the comparison operators `==`, `!=`, `<`, `>`, `<=` and `>=`. If you compare two pairs for identity, at first the members `pair1.first` and `pair2.first` will be compared and then `pair1.second` and `pair2.second`. The same strategy holds for the other comparison operators.

std::make_pair

C++ has the practical help function [std::make_pair](#) to generate pairs, without specifying their types. std::make_pair automatically deduces their types.

The helper function std::make_pair

```
// pair.cpp
...
#include <utility>
...
using namespace std;
...
pair<const char*, double> charDoub("str", 3.14);
pair<const char*, double> charDoub2= make_pair("str", 3.14);
auto charDoub3= make_pair("str", 3.14);

cout << charDoub.first << ", " << charDoub.second;      // str, 3.14
charDoub.first="Str";
get<1>(charDoub)= 4.14;
cout << charDoub.first << ", " << charDoub.second;      // str, 4.14
```

Tuples

You can create tuples of arbitrary length and types with [std::tuple](#). The class template needs the header <tuple>. std::tuple is a generalization of std::pair. You can convert between tuples with two elements and pairs. The tuple has, like his small brother std::pair, a default, a copy and a move constructor. You can swap tuples with the function std::swap.

The i-th element of a tuple t can be referenced by the function template std::get: std::get<i-1>(t). By std::get<type>(t) you can directly refer to the element of the type type.

Tuples support the comparison operators ==, !=, <, >, <= and >=. If you compare two tuples, the elements of the tuples will be compared lexicographically. The comparison starts at the index 0.

std::make_tuple

The helper function [std::make_tuple](#) is quite convenient for the creation of tuples. You don't have to provide the types. The compiler automatically deduces them.

The helper function std::make_tuple

```
// tuple.cpp
...
#include <tuple>
...
```

```

using std::get;

std::tuple<std::string, int, float> tup1("first", 3, 4.17f);
auto tup2= std::make_tuple("second", 4, 1.1);

std::cout << get<0>(tup1) << ", " << get<1>(tup1) << ", "
    << get<2>(tup1) << std::endl; // first, 3, 4.17
std::cout << get<0>(tup2) << ", " << get<1>(tup2) << ", "
    << get<2>(tup2) << std::endl; // second, 4, 1.1
std::cout << (tup1 < tup2) << std::endl; // true

get<0>(tup2)= "Second";

std::cout << get<0>(tup2) << ", " << get<1>(tup2) << ", "
    << get<2>(tup2) << std::endl; // Second, 4, 1.1
std::cout << (tup1 < tup2) << std::endl; // false

auto pair= std::make_pair(1, true);
std::tuple<int, bool> tup= pair;

```

std::tie and std::ignore

[std::tie](#) enables you to create tuples, which elements reference variables. With [std::ignore](#) you can explicitly ignore elements of the tuple.

The helper functions std::tie and std::ignore

```

// tupleTie.cpp
...
#include <tuple>
...
using namespace std;

int first= 1;
int second= 2;
int third= 3;
int fourth= 4;
cout << first << " " << second << " "
    << third << " " << fourth << endl;           // 1 2 3 4

auto tup= tie(first, second, third, fourth);      // bind the tuple
    = std::make_tuple(101, 102, 103, 104);          // create the tuple
                                                // and assign it
cout << get<0>(tup) << " " << get<1>(tup) << " " << get<2>(tup)
    << " " << get<3>(tup) << endl;              // 101 102 103 104
cout << first << " " << second << " " << third << " "
    << fourth << endl;                            // 101 102 103 104

first= 201;
get<1>(tup)= 202;
cout << get<0>(tup) << " " << get<1>(tup) << " " << get<2>(tup)
    << " " << get<3>(tup) << endl;              // 201 202 103 104
cout << first << " " << second << " " << third << " "
    << fourth << endl;                            // 201 202 103 104

int a, b;
tie(std::ignore, a, std::ignore, b)= tup;
cout << a << " " << b << endl;                  // 202 104

```

Reference Wrappers

A reference wrapper is a [copy-constructible](#) and [copy-assignable](#) wrapper for a object of type`&`, which is defined in the header `<functional>`. So you have an object, that behaves like a reference, but can be copied. In opposite to classic references, [`std::reference_wrapper`](#) objects support two additional use cases:

- You can use them in containers of the Standard Template Library.
`std::vector<std::reference_wrapper<int>> myIntRefVector`
- You can copy instances of classes, which have `std::reference_wrapper` objects. That is in general not possible with references.

To access the reference of a `std::reference_wrapper<int> myInt(1)`, the `get` method can be used: `myInt.get()`. You can use a reference wrapper to encapsulate and invoke a callable.

Reference wrappers

```
// referenceWrapperCallable.cpp
...
#include <functional>
...
void foo(){
    std::cout << "Invoked" << std::endl;
}

typedef void callableUnit();
std::reference_wrapper<callableUnit> refWrap(foo);

refWrap();                                // Invoked
```

std::ref and std:: cref

With the helper functions [`std::ref`](#) and [`std:: cref`](#) you can easily create reference wrappers to variables. `std::ref` will create a non-constant reference wrapper, `std:: cref` a constant one:

The helper functions `std::ref` and `std:: cref`

```
// referenceWrapperRefCref.cpp
...
#include <functional>
...
void invokeMe(const std::string& s){
    std::cout << s << ": const " << std::endl;
}

template <typename T>
void doubleMe(T t){
    t *= 2;
}

std::string s{"string"};
```

```
invokeMe(std::cref(s));      // string  
  
int i= 1;  
std::cout << i << std::endl; // 1  
  
doubleMe(i);  
std::cout << i << std::endl; // 1s  
  
doubleMe(std::ref(i));  
std::cout << i << std::endl; // 2
```

So it's possible to invoke the function `invokeMe`, which gets a constant reference to a `std::string`, with a non-constant `std::string s`, which is wrapped in a `std::cref(s)`. If I wrap the variable `i` in the helper function `std::ref`, the function template `doubleMe` will be invoked with a reference. So the variable `i` will be doubled.

Smart Pointers

Smart pointers are one of the most important additions to C++ because they empower you to implement explicit memory management in C++. Beside the *deprecated* `std::auto_ptr`, C++ offers three different smart pointers. They are defined in the header `<memory>`.

Firstly there is the `std::unique_ptr`, which models the concept of exclusive ownership. Secondly, there is the `std::shared_ptr`, who models the concept of shared ownership. Lastly, there is the `std::weak_ptr`. `std::weak_ptr` is not so smart, because it has only a thin interface. Its job is it to break cycles of `std::shared_ptr`. It models the concept of temporary ownership.

The smart pointers manage their resource according to the RAII idiom. So the resource is automatically released if the smart pointer goes out of scope.



Resource Acquisition Is Initialization

Resource Acquisition Is Initialization, short RAII, stands for a popular technique in C++, in which the resource acquisition and release is bound to the lifetime of an object. This means for the smart pointer that the memory is allocated in the constructor and deallocated in the destructor. You can use this technique in C++ because the destructor is called when the object goes out of scope.

Name	Standard	Description
std::auto_ptr <i>(deprecated)</i>	C++98	Owns exclusively the resource. Moves the resource while copying.
std::unique_ptr	C++11	Owns exclusively the resource. Can't be copied.
std::shared_ptr	C++11	Has a reference counter for the shared variable. Manages the reference counter automatically. Deletes the resource, if the reference counter is 0.
std::weak_ptr	C++11	Helps to break cycles of std::shared_ptr. Doesn't modify the reference counter.

std::unique_ptr

[std::unique_ptr](#) exclusively takes care of its resource. It automatically releases the resource if it goes out of scope. If there is no copy semantic required, it can be used in containers and algorithms of the Standard Template Library. std::unique_ptr is as cheap and fast as a raw pointer if you use no special deleter.



Don't use std::auto_ptr

Classical C++03 has a smart pointer `std::auto_ptr`, which exclusively takes care of the lifetime of a resource. But `std::auto_ptr` has a conceptional issue. If you implicitly or explicitly copy a `std::auto_ptr`, the resource is moved. So instead of copy semantic, you have hidden move semantic and therefore you often have undefined behaviour. So `std::auto_ptr` is *deprecated* in C++11 and you should use instead `std::unique_ptr`. You can neither implicitly or explicitly copy a `std::unique_ptr`. You can only move it:

```
#include <memory>
...
std::auto_ptr<int> ap1(new int(2011));
std::auto_ptr<int> ap2 = ap1; // OK

std::unique_ptr<int> up1(new int(2011));
std::unique_ptr<int> up2 = up1; // ERROR
std::unique_ptr<int> up3 = std::move(up1); // OK
```

These are the methods of `std::unique_ptr`.

Methods of <code>std::unique_ptr</code>	
Name	Description
get	Returns a pointer to the resource.
get_deleter	Returns the delete function.
release	Returns a pointer to the resource and releases it.
reset	Resets the resource.
swap	Swaps the resources.

In the following code sample you can see the application of these methods:

The `std::unique_ptr`

```
// uniquePtr.cpp
...
#include <utility>
...
using namespace std;

struct MyInt{
```

```

MyInt(int i):i_(i){}
~MyInt(){
    cout << "Good bye from " << i_ << endl;
}
int i_;
};

unique_ptr<MyInt> uniquePtr1{new MyInt(1998)};
cout << uniquePtr1.get() << endl; // 0x15b5010

unique_ptr<MyInt> uniquePtr2{move(uniquePtr1)};
cout << uniquePtr1.get() << endl; // 0
cout << uniquePtr2.get() << endl; // 0x15b5010
{
    unique_ptr<MyInt> localPtr{new MyInt(2003)}; // Good bye from 2003
}
uniquePtr2.reset(new MyInt(2011)); // Good bye from 1998
MyInt* myInt= uniquePtr2.release(); // Good by from 2011
delete myInt;

unique_ptr<MyInt> uniquePtr3{new MyInt(2017)};
unique_ptr<MyInt> uniquePtr4{new MyInt(2022)};
cout << uniquePtr3.get() << endl; // 0x15b5030
cout << uniquePtr4.get() << endl; // 0x15b5010

swap(uniquePtr3, uniquePtr4);
cout << uniquePtr3.get() << endl; // 0x15b5010
cout << uniquePtr4.get() << endl; // 0x15b5030

```

`std::unique_ptr` has a specialisation for arrays:

`std::unique_ptr` array

```

// uniquePtrArray.cpp
...
#include <memory>
...
using namespace std;

class MyStruct{
public:
    MyStruct():val(count){
        cout << (void*)this << " Hello: " << val << endl;
        MyStruct::count++;
    }
    ~MyStruct(){
        cout << (void*)this << " Good Bye: " << val << endl;
        MyStruct::count--;
    }
private:
    int val;
    static int count;
};

int MyStruct::count= 0;
...
{
    // generates a myUniqueArray with three `MyStructs`
    unique_ptr<MyStruct[]> myUniqueArray{new MyStruct[3]};
}
// 0x1200018 Hello: 0
// 0x120001c Hello: 1
// 0x1200020 Hello: 2

```

```
// 0x1200020 GoodBye: 2
// 0x120001c GoodBye: 1
// 0x1200018 GoodBye: 0
```

Special Deleters

`std::unique_ptr` can be parametrized with special deleters:
`std::unique_ptr<int, MyIntDeleter> up(new int(2011),
myIntDeleter());` `std::unique_ptr` uses by default the deleter of the resource.

`std::make_unique`

The helper function `std::make_unique` was unlike its sibling `std::make_shared` forgotten in the C++11 standard. So `std::make_unique` was added with the C++14 standard. `std::make_unique` enables it to create a `std::unique_ptr` in a single step: `std::unique_ptr<int> up= std::make_unique<int>(2014)`.

`std::shared_ptr`

`std::shared_ptr` shares the ownership of the resource. They have two handles. One for the resource and one for the reference counter. By copying a `std::shared_ptr`, the reference count is increased by one. It is decreased by one if the `std::shared_ptr` goes out of scope. If the reference counter becomes the value 0 and therefore there is no `std::shared_ptr` referencing the resource, the C++ runtime automatically releases the resource. The release of the resource takes place at exactly the time at which the last `std::shared_ptr` goes out of scope. The C++ runtime guarantees that the call of the reference counter is an atomic operation. Because of this management, `std::shared_ptr` uses more time and memory than a raw pointer or `std::unique_ptr`.

In the following table are the methods of `std::shared_ptr`.

Methods of <code>std::shared_ptr</code>	
Name	Description
<code>get</code>	Returns a pointer to the resource.
<code>get_deleter</code>	Returns the delete function
<code>reset</code>	Resets the resource
<code>swap</code>	Swaps the resources.

`unique` Checks if the `std::shared_ptr` is the exclusive owner of the resource.

`use_count` Returns the value of the reference counter.

`std::make_shared`

The helper function `std::make_shared` creates the resource and returns it in a `std::shared_ptr`. You should use `std::make_shared` instead of the direct creation of a `std::shared_ptr`, because `std::make_shared` is a lot faster.

The following code sample shows a typical use case of a `std::shared_ptr`.

```
std::shared_ptr
```

```
// sharedPtr.cpp
...
#include <memory>
...
class MyInt{
public:
    MyInt(int v):val(v){
        std::cout << "Hello: " << val << std::endl;
    }
    ~MyInt(){
        std::cout << "Good Bye: " << val << std::endl;
    }
private:
    int val;
};

auto sharPtr= std::make_shared<MyInt>(1998);           // Hello: 1998
std::cout << sharPtr.use_count() << std::endl;      // 1

{
    std::shared_ptr<MyInt> locSharPtr(sharPtr);
    std::cout << locSharPtr.use_count() << std::endl; // 2
}
std::cout << sharPtr.use_count() << std::endl;        // 1

std::shared_ptr<MyInt> globSharPtr= sharPtr;
std::cout << sharPtr.use_count() << std::endl;        // 2

globSharPtr.reset();
std::cout << sharPtr.use_count() << std::endl;        // 1
sharPtr= std::shared_ptr<MyInt>(new MyInt(2011));   // Hello:2011
                                                        // Good Bye: 1998
...
// Good Bye: 2011
```

The callable is in this example a function object. Therefore you can easily count how many instances of a class were created. The result is in the static variable `count`.

`std::shared_ptr` from this

You can create with the class [`std::enable_shared_from_this`](#) objects which return a `std::shared_ptr` on itself. For that you have to derive the class public from `std::enable_shared_from_this`. So the class support the method `shared_from_this` to return `std::shared_ptr` to this:

`std::shared_ptr` from this

```
// enableShared.cpp
...
#include <memory>
...
class ShareMe: public std::enable_shared_from_this<ShareMe>{
    std::shared_ptr<ShareMe> getShared(){
        return shared_from_this();
    }
};

std::shared_ptr<ShareMe> shareMe(new ShareMe);
std::shared_ptr<ShareMe> shareMe1= shareMe->getShared();

std::cout << (void*)shareMe.get() << std::endl;    // 0x152d010
std::cout << (void*)shareMe1.get() << std::endl;   // 0x152d010
std::cout << shareMe.use_count() << std::endl;    // 2
```

You can see in the code sample that the `get` methods reference the same object.

`std::weak_ptr`

To be honest, [`std::weak_ptr`](#) is not a smart pointer. `std::weak_ptr` supports no transparent access to the resource because it only borrows the resource from a `std::shared_ptr`. `std::weak_ptr` does not change the reference counter:

`std::weak_ptr`

```
// weakPtr.cpp
...
#include <memory>
...
auto sharedPtr= std::make_shared<int>(2011);
std::weak_ptr<int> weakPtr(sharedPtr);

std::cout << weakPtr.use_count() << std::endl;      // 1
std::cout << sharedPtr.use_count() << std::endl;     // 1

std::cout << weakPtr.expired() << std::endl;         // false
if( std::shared_ptr<int> sharedPtr1= weakPtr.lock() ) {
    std::cout << *sharedPtr << std::endl; // 2011
}
else{
    std::cout << "Don't get it!" << std::endl;
}

weakPtr.reset();

if( std::shared_ptr<int> sharedPtr1= weakPtr.lock() ) {
    std::cout << *sharedPtr << std::endl;
```

```

    }
else{
    std::cout << "Don't get it!" << std::endl;           // Don't get it!
}

```

The table provides an overview of the methods of `std::weak_ptr`.

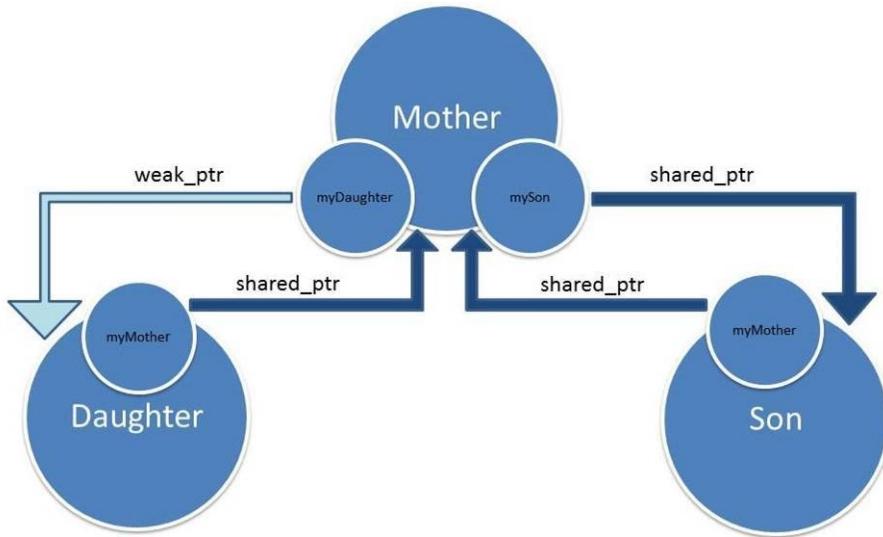
Methods of <code>std::weak_ptr</code>	
Name	Description
<code>expired</code>	Checks if the resource was deleted.
<code>lock</code>	Creates a <code>std::shared_ptr</code> on the resource.
<code>reset</code>	Resets the resource
<code>swap</code>	Swaps the resources.
<code>use_count</code>	Returns the value of the reference counter.

There is one reason for the existence of `std::weak_ptr`. It breaks the cycle of `std::shared_ptr`.

Cyclic References

You get cyclic references of `std::shared_ptr` if they refer to each other. So, the resource counter never becomes 0 and the resource is not automatically released. You can break this cycle if you embed a `std::weak_ptr` in the cycle. `std::weak_ptr` does not modify the reference counter.

The result of the code sample is that the daughter automatically released, but not the son nor the mother. The mother refers to her son via a `std::shared_ptr`, her daughter via a `std::weak_ptr`. Maybe it helps to see the structure of the code in an image.



Cyclic references

Finally the source code.

Cyclic references

```
// cyclicReference.cpp
...
#include <memory>
...
using namespace std;

struct Son, Daughter;

struct Mother{
    ~Mother(){cout << "Mother gone" << endl;}
    void setSon(const shared_ptr<Son> s){mySon= s;}
    void setDaughter(const shared_ptr<Daughter> d){myDaughter= d;}
    shared_ptr<const Son> mySon;
    weak_ptr<const Daughter> myDaughter;
};

struct Son{
Son(shared_ptr<Mother> m):myMother(m){}
    ~Son(){cout << "Son gone" << endl;}
    shared_ptr<const Mother> myMother;
};

struct Daughter{
Daughter(shared_ptr<Mother> m):myMother(m){}
    ~Daughter(){cout << "Daughter gone" << endl;}
    shared_ptr<const Mother> myMother;
};

{
    shared_ptr<Mother> mother= shared_ptr<Mother>(new Mother);
    shared_ptr<Son> son= shared_ptr<Son>(new Son(mother));
    shared_ptr<Daughter> daugh= shared_ptr<Daughter>(new Daughter(mother));
    mother->setSon儿子;
}
```

```
    mother->setDaughter(daugh);
}
// Daughter gone
```

Type Traits

The [type traits library](#) enables you, to check, to compare and to modify types at compile time. So, there is no overhead on the runtime of your program. There are two reasons for using the type traits library: Optimization and Correctness. Optimization, because the introspection capabilities of the type traits library make it possible to choose the faster code automatically. Correctness, because you can specify requirements for your code, which is checked at compile time.



The type traits library and static_assert are a powerful pair

The type traits library and the function `static_assert` are a powerful pair. On one side, the functions of the type traits library provide the type information at compile time, on the other side, the `static_assert` function checks the given information at compile time. This all happens transparently to the runtime of the program:

```
#include <type_traits>
...
template <typename T> T fac(T a){
    static_assert(std::is_integral<T>::value, "T not integral");
    ...
}
fac(10);
fac(10.1); // with T= double; T not integral
```

The GCC compiler quits the function invocation `fac(10.1)`. The message at compile is that `T` is of type `double` and therefore no integral type.

Check Type Information

With the type traits library, you can check primary and composite type categories. The attribute `value` gives you the result.

Primary Type Categories

There are 14 different type categories. They are complete and don't overlap. So each type is only a member of one type category. If you check a type category for your type, the request is independent of the `const` or `volatile` qualifiers.

```

template <class T> struct is_void;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_null_pointer;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;
template <class T> struct is_lvalue_reference;
template <class T> struct is_rvalue_reference;

```

The following code samples show all primary type categories.

All primary type categories

```

// typeCategories.cpp
...
#include <type_traits>
using std::cout;

cout << std::is_void<void>::value;           // true
cout << std::is_integral<short>::value;        // true
cout << std::is_floating_point<double>::value; // true
cout << std::is_array<int [] >::value;         // true
cout << std::is_pointer<int*>::value;          // true
cout << std::is_reference<int&>::value;        // true

struct A{
    int a;
    int f(int){ return 2011; }
};

cout << std::is_member_object_pointer<int A::*>::value; // true
cout << std::is_member_function_pointer<int (A::*)(int)>::value; // true

enum E{
    e= 1,
};
cout << std::is_enum<E>::value;                // true

union U{
    int u;
};
cout << std::is_union<U>::value;                // true

cout << std::is_class<std::string>::value;       // true
cout << std::is_function<int * (double)>::value; // true
cout << std::is_lvalue_reference<int&>::value;   // true
cout << std::is_rvalue_reference<int&&>::value; // true

```

Composite Type Categories

Based on the 14 primary type categories, there are seven composite type categories.

Composite type categories

Composite type categories

Primary type category

is_arithmetic	is_floating_point or is_integral
is_fundamental	is_arithmetic or is_void
is_object	is_arithmetic or is_enum or is_pointer or is_member_pointer
is_reference	is_lvalue_reference or is_rvalue_reference
is_compound	complement of is_fundamental
is_member_pointer	is_member_object_pointer or is_member_function_pointer
is_scalar	is_arithmetic or is_enum or is_pointer or is_is_member_pointer or is_null_pointer

Type Properties

In addition to the primary and composite type categories, there are many type properties.

```
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_trivial;
template <class T> struct is_trivially_copyable;
template <class T> struct is_standard_layout;
template <class T> struct is_pod;
template <class T> struct has_unique_object_represenation;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct is_final;
template <class T> struct is_aggregate;

template <class T> struct is_signed;
template <class T> struct is_unsigned;

template <class T, class... Args> struct is_constructible;
template <class T> struct is_default_constructible;
template <class T> struct is_copy_constructible;
template <class T> struct is_move_constructible;
```

```

template <class T, class U> struct is_assignable;
template <class T> struct is_copy_assignable;
template <class T> struct is_move_assignable;
template <class T> struct is_destructible;
template <class T, class... Args> struct is_trivially_constructible;
template <class T> struct is_trivially_default_constructible;
template <class T> struct is_trivially_copy_constructible;
template <class T> struct is_trivially_move_constructible;
template <class T, class U> struct is_triviallyAssignable;
template <class T> struct is_trivially_copyAssignable;
template <class T> struct is_triviallyMoveAssignable;

template <class T> struct is_trivially_destructible;

template <class T, class... Args> struct is_nothrow_constructible;
template <class T> struct is_nothrow_default_constructible;
template <class T> struct is_nothrow_copy_constructible;
template <class T> struct is_nothrow_move_constructible;

template <class T, class U> struct is_nothrow_assignable;
template <class T> struct is_nothrow_copy_assignable;
template <class T> struct is_nothrow_move_assignable;

template <class T> struct is_nothrow_destructible;
template <class T> struct has_virtual_destructor;

template <class T> struct is_swappable_with;
template <class T> struct is_swappable;
template <class T> struct is_nothrow_swappable_with;
template <class T> struct is_nothrow_swappable;

```

Type Comparisons

The library supports three kinds of type comparisons.

Function	Type comparison	Description
<pre>template <class Base, class Derived> struct is_base_of</pre>	is_base_of	Checks if <code>Derived</code> is derived from <code>Base</code> .
<pre>template <class From, class To> struct is_convertible</pre>	is_convertible	Checks if <code>From</code> can be converted to <code>To</code> .
<pre>template <class T, class U> struct is_same</pre>	is_same	Checks if the types <code>T</code> and <code>U</code> are the same.

Type modifications

The type traits library enables you to modify types during compile time. You can modify the constness of a type:

Type modifications

```
// typeTraitsModifications.cpp
...
#include <type_traits>
...
using namespace std;

cout << is_const<int>::value;           // false
cout << is_const<const int>::value;      // true
cout << is_const<add_const<int>::type>::value; // true

typedef add_const<int>::type myConstInt;
cout << is_const<myConstInt>::value;     // true

typedef const int myConstInt2;
cout << is_same<myConstInt, myConstInt2>::value; // true

cout << is_same<int, remove_const<add_const<int>::type>::type>::value; // true
cout << is_same<const int, add_const<add_const<int>::type>::type>::value; // true
```

The function `std::add_const` adds the constness to a type, while `std::remove_const` removes it.

There are a lot more functions available in the type traits library. You can modify the const-volatile properties of a type.

```
template <class T> struct remove_const;
template <class T> struct remove_VOLATILE;
template <class T> struct remove_cv;

template <class T> struct add_const;
template <class T> struct add_VOLATILE;
template <class T> struct add_cv;
```

You can change at compile time the sign,

```
template <class T> struct make_signed;
template <class T> struct make_unsigned;
```

or the reference or pointer properties of a type.

```
template <class T> struct remove_reference;
template <class T> struct add_lvalue_reference;
template <class T> struct add_rvalue_reference;

template <class T> struct remove_pointer;
template <class T> struct add_pointer;
```

The three following functions are especially valuable for the writing of generic libraries.

```
template <class B> struct enable_if;
template <class B, class T, class F> struct conditional;
template <class... T> common_type;
```

You can conditionally hide with `std::enable_if` a function overload or template specialization from overload resolution. `std::conditional` provides you with the ternary operator at compile time and `std::common_type` gives you the type, to which all type parameter can be implicitly converted to. `std::common_type` is a [variadic template](#), therefore the number of type parameters can be arbitrary.



C++ has a shorthand for `::type` and has a shorthand for `::value`

If you want to get a `const int` from an `int` you have to ask for the type:

`std::add_const<int>::type`. With the C++14 standard use simply `std::add_const_t<int>` instead of the verbose form: `std::add_const<int>::type`. This rule works for all type traits functions.

Accordingly, with C++17 you can use the shorthand `std::is_integral_v<T>` for the predicate `std::is_integral<T>::value`.

Const evaluated Context

The call `std::is_constant_evaluated` allows you to detect if a function call occurs at compile time.

Detect if a function call occurs at compile time

```
// constantEvaluated.cpp
#include <type_traits>
...

constexpr double power(double b, int x) {
    if (std::is_constant_evaluated() && !(b == 0.0 && x < 0)) {

        if (x == 0)
            return 1.0;
        double r = 1.0, p = x > 0 ? b : 1.0 / b;
        auto u = unsigned(x > 0 ? x : -x);
        while (u != 0) {
            if (u & 1) r *= p;
            u /= 2;
    }
}
```

```

        p *= p;
    }
    return r;
}
else {
    return std::pow(b, double(x));
}
}

constexpr double kilo1 = power(10.0, 3);      // execution at compile time

int n = 3;
double kilo2 = power(10.0, n);              // execution at runtime
std::cout << "kilo2: " << kilo2 << std::endl;

```

Time Library

The [time library](#) consists of the three main components, time point, time duration, and clock. Additionally, the library provides the time of day functionality, calendar support, time zone support, and support for in- and output.

Time point

Time point is defined by a starting point, the so-called epoch, and an additional time duration.

Time duration

Time duration is the difference between two time-points. It is given by the number of ticks.

Clock

A clock consists of a starting point (epoch) and a tick so that the current time point can be calculated.

Time of day

The duration since midnight split into hours:minutes:seconds.

Calendar

Calendar stands for various calendar days such as year, a month, a weekday, or the n-th day of a week.

Time zone

Represents time specific to a geographic area.



The time library as key component for multithreading

The time library is a key component of the new multithreading capabilities of C++. You can put the current thread by `std::this_thread::sleep_for(std::chrono::milliseconds(15))` for 15 milliseconds to sleep, or you try to acquire a lock for 2 minutes: `lock.try_lock_until(now + std::chrono::minutes(2))`.

Time Point

A duration consists of a span of time, defined as some number of ticks of some time unit. A time point consists of a clock and a time duration. This time duration can be positive or negative.

```
template <class Clock, class Duration= typename Clock::duration>
class time_point;
```

The epoch is not defined for the clocks `std::chrono::steady_clock`, `std::chrono::high_resolution_clock` and `std::chrono::system`. But on the popular platform the epoch of `std::chrono::system` is usually defined as 1.1.1970. You can calculate the time since 1.1.1970 in the resolutions nanoseconds, seconds and minutes.

Time since epoch

```
// epoch.cpp
...
#include <chrono>
...
auto timeNow= std::chrono::system_clock::now();
auto duration= timeNow.time_since_epoch();
std::cout << duration.count() << "ns"           // 1413019260846652ns

typedef std::chrono::duration<double> MySecondTick;
MySecondTick mySecond(duration);
std::cout << mySecond.count() << "s";        // 1413019260.846652s

const int minute= 60;
typedef std::chrono::duration<double, <minute>> MyMinuteTick;
MyMinuteTick myMinute(duration);
std::cout << myMinute.count() << "m";        // 23550324.920572m
```

Thanks to the function `std::chrono::clock_cast` you can cast time points between various clocks.



Easy performance tests with the time library

Performance measurement

```
// performanceMeasurement.cpp
...
#include <chrono>
...
std::vector<int> myBigVec(10000000, 2011);
std::vector<int> myEmptyVec1;

auto begin= std::chrono::high_resolution_clock::now();
myEmptyVec1 = myBigVec;
auto end= std::chrono::high_resolution_clock::now() - begin;

auto timeInSeconds = std::chrono::duration<double>(end).count();
std::cout << timeInSeconds << std::endl; // 0.0150688800
```

Time Duration

Time duration is the difference between the two time-points. Time duration is measured in the number of ticks.

```
template <class Rep, class Period = ratio<1>> class duration;
```

If Rep is a floating point number, the time duration supports fractions of ticks. The most important time durations are predefined in the chrono library:

```
typedef duration<signed int, nano> nanoseconds;
typedef duration<signed int, micro> microseconds;
typedef duration<signed int, milli> milliseconds;
typedef duration<signed int> seconds;
typedef duration<signed int, ratio< 60>> minutes;
typedef duration<signed int, ratio<3600>> hours;
```

How long can a time duration be? The C++ standard guarantees that the predefined time durations can store +/- 292 years. You can easily define your time duration like a German school hour: `typedef std::chrono::duration<double, std::ratio<2700>> MyLessonTick`. Time durations in natural numbers have to be explicitly converted to time durations in floating pointer numbers. The value will be truncated:

Durations

```
// duration.cpp
...
#include <chrono>
#include <ratio>
```

```

using std::chrono;

typedef duration<long long, std::ratio<1>> MySecondTick;
MySecondTick aSecond(1);

milliseconds milli(aSecond);
std::cout << milli.count() << " milli";           // 1000 milli

seconds seconds(aSecond);
std::cout << seconds.count() << " sec";          // 1 sec

minutes minutes(duration_cast<minutes>(aSecond));
std::cout << minutes.count() << " min";           // 0 min

typedef duration<double, std::ratio<2700>> MyLessonTick;
MyLessonTick myLesson(aSecond);
std::cout << myLesson.count() << " less";        // 0.00037037 less

```



std::ratio

std::ratio supports arithmetic at compile time with rational numbers. A rational number has two template arguments. One is the nominator, the other the denominator. C++11 predefines lots of rational numbers.

```

typedef ratio<1, 1000000000000000000> atto;
typedef ratio<1, 1000000000000000> femto;
typedef ratio<1, 1000000000000> pico;
typedef ratio<1, 1000000000> nano;
typedef ratio<1, 1000000> micro;
typedef ratio<1, 1000> milli;
typedef ratio<1, 100> centi;
typedef ratio<1, 10> deci;
typedef ratio< 10, 1> deca;
typedef ratio< 100, 1> hecto;
typedef ratio< 1000, 1> kilo;
typedef ratio< 1000000, 1> mega;
typedef ratio< 1000000000, 1> giga;
typedef ratio< 1000000000000, 1> tera;
typedef ratio< 1000000000000000, 1> peta;
typedef ratio< 1000000000000000000, 1> exa;

```

C++14 has built-in literals for the most used time durations.

Built-in literals for time durations

Type	Suffix	Example
------	--------	---------

std::chrono::hours	h	5h
--------------------	---	----

<code>std::chrono::minutes</code>	<code>min</code>	<code>5min</code>
<code>std::chrono::seconds</code>	<code>s</code>	<code>5s</code>
<code>std::chrono::milliseconds</code>	<code>ms</code>	<code>5ms</code>
<code>std::chrono::microseconds</code>	<code>us</code>	<code>5us</code>
<code>std::chrono::nanoseconds</code>	<code>ns</code>	<code>5ns</code>

Clock

The clock consists of a starting point and a tick. You can get the current time with the method now.

`std::chrono::system_clock`

System time, which you can synchronise with the external clock.

`std::chrono::steady_clock`

Clock, which can not be adjusted.

`std::chrono::high_resolution_clock`

System time with the greatest accuracy.

`std::chrono::system_clock` will refer typically to the 1.1.1970. You can not adjust `std::steady_clock` forward or backward in opposite to two other clocks. The methods `to_time_t` and `from_time_t` can be used to convert between `std::chrono::system_clock` and `std::time_t` objects.

Time of Day

`std::chrono::time_of_day` splits the duration since midnight into hours:minutes:seconds. The functions `std::chrono::is_am` and `std::chrono::is_pm` checks if the time is before midday (ante meridiem) or after midday (post meridiem).

A `std::chrono::time_of_day` object `t0fDay` supports various member functions.

Member functions of `std::chrono::time_of_day`

Member function	Description
------------------------	--------------------

<code>tOfDay.hours()</code>	Returns the hour component since midnight.
<code>tOfDay.minutes()</code>	Returns the minute component since midnight.
<code>tOfDay.seconds()</code>	Returns the second component since midnight.
<code>tOfDay.subseconds()</code>	Returns the fractiona second component since midnight.
<code>tOfDay.to_duration()</code>	Returns the time duration since midnight.
<code>std::chrono::make12(hr)</code>	Returns the 12-hour (24-hour) equivalent of a 24-hour (12-hour) format time.
<code>std::chrono::make24(hr)</code>	
<code>std::chrono::is_am(hr)</code>	Detects wheter the 24-hour format time is a.m. or p.m..
<code>std::chrono::is_pm(hr)</code>	

Calendar

Calendar stands for various calender dates such as year, a month, a weekday, or the n-th day of a week.

The current time

```
// currentTime.cpp
...
#include <chrono>
using std::chrono;
...
auto now = system_clock::now();
std::cout << "The current time is " << now << " UTC\n";
auto currentYear = year_month_day(floor<days>(now).year());
std::cout << "The current year is " << currentYear << '\n';
auto h = floor<hours>(now) - sys_days(January/1/currentYear);
std::cout << "It has been " << h << " since New Years!\n";
std::cout << std::endl;
auto birthOfChrist = year_month_weekday(sys_days(January/01/0000));
std::cout << "Weekday: " << birthOfChrist.weekday() << '\n';
```

The output of the program shows information referring the current time.

```
The current time is 2020-07-18 20:39:12.356023527 UTC
The current year is 2020
It has been 4796h since New Years!

Weekday: Sat
```

The following table gives an overview on the calender types.

Class	Various calendar types
Description	
last_spec	Indicates the last day or weekday in a month
day	Represents a day of a month.
month	Represents a month of a year.
year	Represents a year in the Gregorian calendar.
weekday	Represents a day of the week in the Gregorian calendar.
weekday_indexed	Represents the n-th weekday of a month.
weekday_last	Represents the last weekday of a month.
month_day	Represents a specific day of a specific month.
month_day_last	Represents the last day of a specific month.
month_weekday	Represents the n-th weekday of a specific month.
month_weekday_last	Represents the last weekday of a specific month.
year_month	Represents a specific month of a specific year.

<code>year_month_day</code>	Represents a specific year, month, and day.
<code>year_month_day_last</code>	Represents the last day of a specific year and month.
<code>year_month_weekday</code>	Represents the last day of a specific year and month.
<code>year_month_weekday_last</code>	Represents the last weekday of a specific year and month.

Time Zone

Time zones represent time specific to a geographic area. The following program snippe displays the local time in various time zones.

Local time displayed in various time time zones

```
// timezone.cpp
...
#include <chrono>
using 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 = std::chrono::zoned_time<milliseconds>("America/New_York", time);
auto tokyoTime = zoned_time<milliseconds>("Asia/Tokyo", time);

std::cout << time << std::endl;
std::cout << localTime << std::endl;
std::cout << berlinTime << std::endl;
std::cout << newYorkTime << std::endl;
std::cout << tokyoTime << std::endl;
```

The time zone functionality supports the access of the [IANA time zone database](#), enables the opeartion with various time zones, and provides information about leap seconds.

The following table gives an overview of the time zone functionaliy. For more detailed information, refer to [cpprefere.com](#).

Type	Description	Time zone information
<code>tzdb</code>	Describes the IANA time zone database.	

<code>locate_zone</code>	Locates a <code>time_zone</code> bases on its name..
<code>current_zone</code>	Returns the current <code>time_zone</code> .
<code>time_zone</code>	Represents a time zone.
<code>sys_info</code>	Returns information about a time zone at a specific time point.
<code>local_info</code>	Represents infomation about the local time to UNIS time conventions.
<code>zoned_time</code>	Represents a time zone and a time point.
<code>leap_second</code>	Contains information about a leap second insertion.

Chrono I/O

The function `std::chrono::parse` parses a chrono object from a stream.

Parsing a time point and a time zone

```
std::istringstream inputStream{"1999-10-31 01:30:00 -08:00 US/Pacific"};
std::chrono::local_seconds timePoint;
std::string timeZone;
inputStream >> std::chrono::parse("%F %T %E %Z", timePoint, timeZone);
```

The parse functionality provides various format specifier to deal with time of day and calendar dates such as year, month, week, and day. cppreference.com provides detailed information to the format spezifieres.

`std::any, std::optional, and std::variant`

The new C++17 data types `std::any`, `std::optional`, and `std::variant` are all based on the [Boost libraries](#)

`std::any`

[`std::any`](#) is a type-safe container for single values of any type which is copy-constructible. There are a few ways to create a `std::any` container any. You can use the various constructors or the factory function `std::make_any`. By using

`any.emplace`, you directly construct one value into `any`. `any.reset` lets you destroy the contained object. If you want to know whether the container `any` has a value, use the method `any.has_value`. You can even get the typeid of the container object via `any.type`. Thanks to the generic function `std::any_cast` you have access to the contained object. If you specify the wrong type, you will get a `std::bad_any_cast` exception.

Here is a code snippet showing the basic usage of `std::any`.

```
std::any  
// any.cpp  
...  
#include <any>  
struct MyClass{};  
  
...  
  
std::vector<std::any> anyVec{true, 2017, std::string("test"), 3.14, MyClass()};  
std::cout << std::any_cast<bool>(anyVec[0]); // true  
int myInt= std::any_cast<int>(anyVec[1]);  
std::cout << myInt << std::endl; // 2017  
  
std::cout << anyVec[0].type().name(); // b  
std::cout << anyVec[1].type().name(); // i
```

The program snippet defines a `std::vector<std::any>`. To get one of its elements, you have to use `std::any_cast`. As mentioned, if you use the wrong type, you will get a `std::bad_any_cast` exception.



The string representation of the typeid

The string representation of the typeid is implementation defined. If `anyVec[1]` is of type `int` the expression `anyVec[1].type().name()` will return `i` with the [GCC C++ compiler](#) and `int` with the [Microsoft Visual C++ compiler](#).

`std::any` can have objects of arbitrary types; `std::optional` may or may not have a value.

`std::optional`

[`std::optional`](#) is quite comfortable for calculations such as database queries that may have a result.



Don't use no-results

Before C++17 it was common practice to use a special value such as a null pointer, an empty string, or a unique integer to denote the absence of a result. These special values or no-results are very error-prone because you have to misuse the type system to check the return value. This means that for the type system that you have to use a regular value such as an empty string to define an irregular value.

The various constructors and the convenience function `std::make_optional` let you define an optional object `opt` with or without a value. `opt.emplace` will construct the contained value in-place and `opt.reset` will destroy the container value. You can explicitly ask a `std::optional` container if it has a value or you can check it in a logical expression. `opt.value` returns the value and `opt.value_or` returns the value or a default value. If `opt` has no contained value, the call `opt.value` will throw a `std::bad_optional_access` exception.

Here is a short example of using `std::optional`.

```
std::optional  
-----  
// optional.cpp  
...  
#include <optional>  
  
std::optional<int> getFirst(const std::vector<int>& vec){  
    if (!vec.empty()) return std::optional<int>(vec[0]);  
    else return std::optional<int>();  
}  
  
...  
  
std::vector<int> myVec{1, 2, 3};  
std::vector<int> myEmptyVec;  
  
auto myInt= getFirst(myVec);  
  
if (myInt){  
    std::cout << *myInt << std::endl; // 1  
    std::cout << myInt.value() << std::endl; // 1  
    std::cout << myInt.value_or(2017) << std::endl; // 1  
}  
  
auto myEmptyInt= getFirst(myEmptyVec);  
  
if (!myEmptyInt){  
    std::cout << myEmptyInt.value_or(2017) << std::endl; // 2017  
}
```

I use `std::optional` in the function `getFirst`. `getFirst` returns the first element if it exists. If not, you will get a `std::optional<int>` object. The main function has two vectors. Both invoke `getFirst` and return a `std::optional` object. In the case of `myInt` the object has a value; in the case of `myEmptyInt`, the object has no value. The program displays the value of `myInt` and `myEmptyInt`. `myInt.value_or(2017)` returns the value, but `myEmptyInt.value_or(2017)` returns the default value.

`std::variant`, explained in the next section, can have more than one value.

`std::variant`

`std::variant` is a type-safe union. An instance of `std::variant` has a value from one of its types. The type must not be a reference, array or void. A `std::variant` can have a type more than once. A default-initialised `std::variant` is initialised with its first type; therefore, its first type must have a default constructor. By using `var.index` you get the zero-based index of the alternative held by the `std::variant` `var`. `var.valueless_by_exception` returns false if the variant holds a value. By using `var.emplace` you can create a new value in-place. There are a few global functions used to access a `std::variant`. The function template `var.holds_alternative` lets you check if the `std::variant` holds a specified alternative. You can use `std::get` with an index and with a type as argument. By using an index, you will get the value. If you invoke `std::get` with a type, you only will get the value if it is unique. If you use an invalid index or a non-unique type, you will get a `std::bad_variant_access` exception. In contrast to `std::get` which eventually returns an exception, `std::get_if` returns a null pointer in the case of an error.

The following code snippet shows you the usage of a `std::variant`.

```
std::variant  
-----  
// variant.cpp  
...  
#include <variant>  
  
...  
  
std::variant<int, float> v, w;           // v contains int  
v = 12;                                     // same effect as the previous line  
int i = std::get<int>(v);  
w = std::get<int>(v);  
w = std::get<0>(v);                         // same effect as the previous line  
w = v;  
  
// std::get<double>(v);                      // error: no double in [int, float]  
// std::get<3>(v);                           // error: valid index values are 0 and 1
```

```

try{
    std::get<float>(w);           // w contains int, not float: will throw
}
catch (std::bad_variant_access&) {}

std::variant<std::string> v("abc"); // converting constructor must be unambiguous
v = "def";                         // converting assignment must be unambiguous

```

v and w are two variants. Both can have an int and a float value. Their default value is 0. v becomes 12 and the following call std::get<int>(v) returns the value. The next three lines show three possibilities to assign the variant v to w, but you have to keep a few rules in mind. You can ask for the value of a variant by type std::get<double>(v) or by index: std::get<3>(v). The type must be unique and the index valid. The variant w holds an int value; therefore, I get a std::bad_variant_access exception if I ask for a float type. If the constructor call or assignment call is unambiguous, a conversion can take place. This is the reason that it's possible to construct a std::variant<std::string> from a C-string or assign a new C-string to the variant.

std::variant has an interesting non-member function std::visit that allows you to execute a [callable](#) on a list of variants. A callable is something which you can invoke. Typically this can be a function, a function object, or lambda expression. For simplicity reasons, I use a lambda function in this example.

```

std::visit


---


// visit.cpp
...
#include <variant>
...

std::vector<std::variant<char, long, float, int, double, long long>>
vecVariant = {5, '2', 5.4, 100ll, 2011l, 3.5f, 2017};

for (auto& v: vecVariant){
    std::visit([](auto&& arg){std::cout << arg << " ";}, v);
                           // 5 2 5.4 100 2011 3.5 2017
}

// display each type
for (auto& v: vecVariant){
    std::visit([](auto&& arg){std::cout << typeid(arg).name() << " ";}, v);
                           // int char double __int64 long float int
}

// get the sum
std::common_type<char, long, float, int, double, long long>::type res{};

std::cout << typeid(res).name() << std::endl;           // double

for (auto& v: vecVariant){
    std::visit([&res](auto&& arg){res+= arg;}, v);
}
```

```

}

std::cout << "res: " << res << std::endl; // 4191.9

// double each value
for (auto& v: vecVariant){
    std::visit([&res](auto&& arg){arg *= 2;}, v);
    std::visit([](auto&& arg){std::cout << arg << " ";}, v);
        // 10 d 10.8 200 4022 7 4034
}

```

Each variant in this example can hold a `char`, `long`, `float`, `int`, `double`, or `long long`. The first visitor `[](auto&& arg){std::cout << arg << " " ;}` will output the various variants. The second visitor `std::cout << typeid(arg).name() << " " ;` will display its types.

Now I want to sum up the elements of the variants. First I need the right result type at compile time. `std::common_type` from the [type traits library](#) will provide it. `std::common_type` gives the type to which all types `char`, `long`, `float`, `int`, `double`, and `long long` can implicitly be converted to. The final `{}` in `res{}` causes it to be initialised to `0.0`. `res` is of type `double`. The visitor `[&res](auto&& arg){arg *= 2;}` calculates the sum and the following line displays it.

3. Interface of All Containers

Although the sequence and associative containers of the Standard Template library are two quite different classes of containers, they have a lot in common. For example, the operations, to create or delete a container, to determine its size, to access its elements, to assign or swap are all independent of the type of elements of a container. It is common for the containers that you can define them with an arbitrary size, and each container has an allocator. That's the reason the size of a container can be adjusted at runtime. The allocator works most of the time in the background. This can be seen for a `std::vector`. The call `std::vector<int>` results in a call `std::vector<int, std::allocator<int>>`. Because of the `std::allocator`, you can adjust except for `std::array` the size of all containers dynamically. However, they have yet more in common. You can access the elements of a container quite easily with an iterator.

Having so much in common, the containers differ in the details. The chapters [Sequence Container](#) and [Associative Container](#) provide the details.

With the sequence containers [`std::array`](#), [`std::vector`](#), [`std::deque`](#), [`std::list`](#), and [`std::forward_list`](#) C++ has an expert on each domain.

The same holds true for the associative containers, which can be classified in the order and unordered ones.

Create and delete

You can construct each container by a multitude of constructors. To delete all elements of a container `cont`, you can use `cont.clear()`. It makes no difference if you create a container, if you delete them or if you add or remove elements. Each time the container takes care of the memory management.

The table shows you the constructors and destructors of a container. A `std::vector` stands for the rest of them.

Type	Creation and deletion of a container Example
------	---

Default	<code>std::vector<int> vec1</code>
Range	<code>std::vector<int> vec2(vec1.begin(), vec1.end())</code>
Copy	<code>std::vector<int> vec3(vec2)</code>
Copy	<code>std::vector<int> vec3= vec2</code>
Move	<code>std::vector<int> vec4(std::move(vec3))</code>
Move	<code>std::vector<int> vec4= std::move(vec3)</code>
Sequence (Initializer list)	<code>std::vector<int> vec5 {1, 2, 3, 4, 5}</code>
Sequence (Initializer list)	<code>std::vector<int> vec5= {1, 2, 3, 4, 5}</code>
Destructor	<code>vec5.~vector()</code>
Delete elements	<code>vec5.clear()</code>

Because `std::array` is generated at compile time, it is a special container. `std::array` has no move constructor and can neither be created with a range nor with an initialiser list. However, you can initialize a `std::array` with an aggregate initialisation. Also, `std::array` has no method for removing its elements.

Now I can use the different constructors on the different containers.

Various constructors

```
// containerConstructor.cpp
...
#include <map>
#include <unordered_map>
#include <vector>
...
using namespace std;
```

```

vector<int> vec= {1, 2, 3, 4, 5, 6, 7, 8, 9};
map<string, int> m= {{"bart", 12345}, {"jenne", 34929}, {"huber", 840284} };
unordered_map<string, int> um{m.begin(), m.end()};

for (auto v: vec) cout << v << " "; // 1 2 3 4 5 6 7 8 9
for (auto p: m) cout << p.first << "," << p.second << " ";
// bart,12345 huber,840284 jenne,34929
for (auto p: um) cout << p.first << "," << p.second << " ";
// bart,12345 jenne,34929 huber,840284

vector<int> vec2= vec;
cout << vec.size() << endl; // 9
cout << vec2.size() << endl; // 9

vector<int> vec3= move(vec);
cout << vec.size() << endl; // 0
cout << vec3.size() << endl; // 9

vec3.clear();
cout << vec3.size() << endl; // 0

```

Size

For a container `cont`, you can check with `cont.empty()` if the container is empty. `cont.size()` returns the current number of elements, and `cont.max_size()` returns the maximum number of elements `cont` can have. The maximum number of elements is implementation-defined.

Size of a container

```

// containerSize.cpp
...
#include <map>
#include <set>
#include <vector>
...
using namespace std;

vector<int> intVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
map<string, int> str2Int = {{"bart", 12345},
                           {"jenne", 34929}, {"huber", 840284}};
set<double> douSet{3.14, 2.5};

cout << intVec.empty() << endl; // false
cout << str2Int.empty() << endl; // false
cout << douSet.empty() << endl; // false

cout << intVec.size() << endl; // 9
cout << str2Int.size() << endl; // 3
cout << douSet.size() << endl; // 2

cout << intVec.max_size() << endl; // 4611686018427387903
cout << str2Int.max_size() << endl; // 384307168202282325
cout << douSet.max_size() << endl; // 461168601842738790

```



Use `cont.empty()` instead of `cont.size()`

For a container `cont`, use the method `cont.empty()` instead of `(cont.size() == 0)` to determine if the container is empty. First, `cont.empty()` is in general faster than `(const.size() == 0)`; second, `std::forward_list` has no method `size()`.

Access

To access the elements of a container, you can use an iterator. If you use a begin and end iterator, you have a range, which you can further process. For a container `cont`, you get with `cont.begin()` the begin iterator and with `cont.end()` the end iterator, which defines a *half-open* range. It is *half-open* because the begin iterator belongs to the range, the end iterator refers to a position past the range. With the iterator pair `cont.begin()` and `cont.end()` you can modify the elements.

Iterator	Creation and deletion of a container	Description
<code>cont.begin()</code> and <code>cont.end()</code>		Pair of iterators to iterate forward.
<code>cont.cbegin()</code> and <code>cont.cend()</code>		Pair of iterators to iterate const forward.
<code>cont.rbegin()</code> and <code>cont.rend()</code>		Pair of iterators to iterate backward.
<code>cont.crbegin()</code> and <code>cont.crend()</code>		Pair of iterators to iterate const backward.

Now I can modify the container.

Access the elements of a container

```
// containerAccess.cpp
...
#include <vector>
...
struct MyInt{
    MyInt(int i): myInt(i){};
    int myInt;
};

std::vector<MyInt> myIntVec;
myIntVec.push_back(MyInt(5));
```

```

myIntVec.emplace_back(1);
std::cout << myIntVec.size() << std::endl; // 2

std::vector<int> intVec;
intVec.assign({1, 2, 3});
for (auto v: intVec) std::cout << v << " "; // 1 2 3

intVec.insert(intVec.begin(), 0);
for (auto v: intVec) std::cout << v << " "; // 0 1 2 3

intVec.insert(intVec.begin()+4, 4);
for (auto v: intVec) std::cout << v << " "; // 0 1 2 3 4

intVec.insert(intVec.end(), {5, 6, 7, 8, 9, 10, 11});

for (auto v: intVec) std::cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10 11

for (auto revIt= intVec.rbegin(); revIt != intVec.rend(); ++revIt)
    std::cout << *revIt << " "; // 11 10 9 8 7 6 5 4 3 2 1 0

intVec.pop_back();
for (auto v: intVec ) std::cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10

```

Assign and Swap

You can assign new elements to existing containers or swap two containers. For the assignment of a container `cont2` to a container `cont`, there exists the copy assignment `cont = cont2` and the move assignment `cont = std::move(cont2)`. A special form of assignment is the one with an initialiser list: `cont = {1, 2, 3, 4, 5}`. That's not possible for `std::array`, but you can instead use the aggregate initialisation. The function `swap` exists in two forms. You have it as a method `cont(swap(cont2))` or as a function template `std::swap(cont, cont2)`.

Assignment and swap

```

// containerAssignmentAndSwap.cpp
...
#include <set>
...
std::set<int> set1{0, 1, 2, 3, 4, 5};
std::set<int> set2{6, 7, 8, 9};

for (auto s: set1) std::cout << s << " "; // 0 1 2 3 4 5
for (auto s: set2) std::cout << s << " "; // 6 7 8 9

set1= set2;
for (auto s: set1) std::cout << s << " "; // 6 7 8 9
for (auto s: set2) std::cout << s << " "; // 6 7 8 9

set1= std::move(set2);
for (auto s: set1) std::cout << s << " "; // 6 7 8 9
for (auto s: set2) std::cout << s << " "; //

set2= {60, 70, 80, 90};
for (auto s: set1) std::cout << s << " "; // 6 7 8 9
for (auto s: set2) std::cout << s << " "; // 60 70 80 90

std::swap(set1, set2);

```

```
for (auto s: set1) std::cout << s << " "; // 60 70 80 90
for (auto s: set2) std::cout << s << " "; // 6 7 8 9
```

Compare

Containers support the comparison operators ==, !=, <, >, <=, >=. The comparison of two containers happens on the elements of the containers. If you compare associative containers, their key is compared. Unordered associative containers support only the comparison operator == and !=.

Comparison of a container

```
// containerComparison.cpp
...
#include <array>
#include <set>
#include <unordered_map>
#include <vector>
...
using namespace std;

vector<int> vec1{1, 2, 3, 4};
vector<int> vec2{1, 2, 3, 4};
cout << (vec1 == vec2) << endl;           // true

array<int, 4> arr1{1, 2, 3, 4};
array<int, 4> arr2{1, 2, 3, 4};
cout << (arr1 == arr2) << endl;           // true

set<int> set1{1, 2, 3, 4};
set<int> set2{4, 3, 2, 1};
cout << (set1 == set2) << endl;           // true

set<int> set3{1, 2, 3, 4, 5};
cout << (set1 < set3) << endl;           // true

set<int> set4{1, 2, 3, -3};
cout << (set1 > set4) << endl;           // true

unordered_map<int, string> uSet1{{1, "one"}, {2, "two"}};
unordered_map<int, string> uSet2{{1, "one"}, {2, "Two"}};
cout << (uSet1 == uSet2) << endl;         // false
```

Erasure

The free functions std::erase(cont, val) and std::erase_if(cont, pred) erase all elements of the container cont which compares equal to val or fulfil the predicate pred. Both functions return the number of erased elements.

Consistent container erasure

```
// erase.cpp
...
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 << std::endl;
}

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

...

std::string str{"A sentence with e."};
std::cout << "str: " << str << std::endl;
std::erase(str, 'e');
std::cout << "str: " << str << std::endl;

std::cout << "\nstd::vector " << std::endl;
std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
doAll(vec);

std::cout << "\nstd::deque " << std::endl;
std::deque deq{1, 2, 3, 4, 5, 6, 7, 8, 9};
doAll(deq);

std::cout << "\nstd::list" << std::endl;
std::list lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
doAll(lst);

```

erase and erase_if can be applied to all containers of the STL and std::string.

```
str: A sentence with e.  
str: A sntnc with .  
  
std::vector  
1 2 3 4 5 6 7 8 9  
1 2 3 4 6 7 8 9  
1 2  
  
std::deque  
1 2 3 4 5 6 7 8 9  
1 2 3 4 6 7 8 9  
1 2  
  
std::list  
1 2 3 4 5 6 7 8 9  
1 2 3 4 6 7 8 9  
1 2
```

4. Sequence Containers

The [sequence container](#) have a lot in [common](#), but each container has its special domain. Before I dive into the details, I provide the overview of all five sequence containers of the std namespace.

Criteria	The sequence containers				
	array	vector	deque	list	for
Size	static	dynamic	dynamic	dynamic	dynamic
Implementation	static array	dynamic array	sequence of arrays	doubled linked list	single list
Access	random	random	random	forward and backward	for backward
Optimized for insert and delete at		end: O(1)	begin and end: O(1)	begin and end: O(1) arbitrary: O(1)	begin and end: O(1) arbitrary: O(1)
Memory reservation	yes		no	no	no
Release of memory		shrink_to_fit	shrink_to_fit	always	always
Strength	no memory allocation; minimal	95% solution	insertion and deletion at the	insertion and deletion	fast and minimal

	memory requirements	begin and end	at an arbitrary position	me req
Weakness	no dynamic memory allocation	Insertion and deletion at an arbitrary position: O(n)	Insertion and deletion at an arbitrary position: O(n)	no random access

A few additional remarks to the table.

$O(i)$ stands for the complexity (runtime) of an operation. So $O(1)$ means that the runtime of an operation on a container is constant and is independent of the size of the container. Opposite to that, $O(n)$ means, that the runtime depends linear on the number of the elements of the container. What does that mean for a `std::vector`. The access time on an element is independent of the size of the `std::vector`, but the insertion or deletion of an arbitrary element with k -times more elements is k -times slower.

Although the random access on the elements of a `std::vector` has the same complexity $O(1)$ as the random access on the element of a `std::deque`, that doesn't mean, that both operations are equally fast.

The complexity guarantee $O(1)$ for the insertion or deletion into a double (`std::list`) or single linked list (`std::forward_list`) is only guaranteed if the iterator points to the right element.



`std::string` is like `std::vector<char>`

Of course `std::string` is no container of the standard template library. But from a behavioural point of view, it is like a sequence container, especially like a `std::vector<char>`. Therefore I will treat `std::string` as a `std::vector<char>`.

Arrays

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

[`std::array`](#) is a homogeneous container of fixed length. It needs the header `<array>`. The `std::array` combines the memory and runtime characteristic of a C array with the interface of `std::vector`. This means in particular, the `std::array` knows its size. You can use `std::array` in the algorithms of the STL.

You have to keep a few special rules in your mind to initialise a `std::array`.

`std::array<int, 10> arr`

The 10 elements are not initialised.

`std::array<int, 10> arr{}`

The 10 elements are default initialised.

`std::array<int, 10> arr{1, 2, 3, 4, 5}`

The remaining elements are default initialised.

`std::array` supports three types of index access.

```
arr[n];
arr.at(n);
std::get<n>(arr);
```

The most often used first type form with angle brackets does not check the boundaries of the `arr`. This is in opposition to `arr.at(n)`. You will get eventually a `std::range_error` exception. The last type shows the relationship of the `std::array` with the [`std::tuple`](#), because both are containers of fixed length.

Here is a little bit of arithmetic with `std::array`.

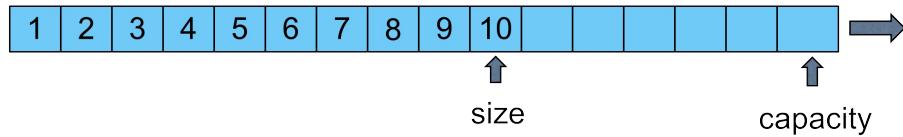
```
std::array
// array.cpp
...
#include <array>
...
std::array<int, 10> arr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
for (auto a: arr) std::cout << a << " " ;    // 1 2 3 4 5 6 7 8 9 10

double sum= std::accumulate(arr.begin(), arr.end(), 0);
std::cout << sum << std::endl;                // 55

double mean= sum / arr.size();
```

```
std::cout << mean << std::endl; // 5.5
std::cout << (arr[0] == std::get<0>(arr)); // true
```

Vectors



`std::vector` is a homogeneous container, for which its length can be adjusted at runtime. `std::vector` needs the header `<vector>`. As it stores its elements contiguously in memory, `std::vector` supports pointer arithmetic.

```
for (int i= 0; i < vec.size(); ++i){
    std::cout << vec[i] == *(vec + i) << std::endl; // true
}
```



Distinguish the round and curly braces by the creation of a `std::vector`

If you construct a `std::vector`, you have to keep a few specialities in mind. The constructor with round braces in the following example creates a `std::vector` with 10 elements, the constructor with curly braces a `std::vector` with the element 10.

```
std::vector<int> vec(10);
std::vector<int> vec{10};
```

The same rules hold for the expressions `std::vector<int>(10, 2011)` or `std::vector<int>{10, 2011}`. In the first case, you get a `std::vector` with 10 elements, initialised to 2011. In the second case, you get a `std::vector` with the elements 10 and 2011. The reason for the behaviour is, that curly braces are interpreted as initialiser lists and so, the [sequence constructor](#) is used.

Size versus Capacity

The number of elements a `std::vector` has is usually smaller than the number of elements for which space is already reserved. That is for a simple reason. The size of the `std::vector` can increase without an expensive allocation of new memory.

There are a few methods for the smart handling of memory:

Method	Memory management of <code>std::vector</code>	Description
<code>vec.size()</code>		Number of elements of vec.
<code>vec.capacity()</code>		Number of elements, which vec can have without reallocation.
<code>vec.resize(n)</code>		vec will be increased to n elements.
<code>vec.reserve(n)</code>		Reserve memory for at least n elements.
<code>vec.shrink_to_fit()</code>		Reduces capacity of vec to the size.

The call `vec.shrink_to_fit()` is not binding. That means the runtime can ignore it. But on popular platforms, I always observed the desired behaviour.

So let's see the methods in the application.

```
std::vector
// vector.cpp
...
#include <vector>
...
std::vector<int> intVec1(5, 2011);
intVec1.reserve(10);
std::cout << intVec1.size() << std::endl;      // 5
std::cout << intVec1.capacity() << std::endl; // 10

intVec1.shrink_to_fit();
std::cout << intVec1.capacity() << std::endl; // 5

std::vector<int> intVec2(10);
std::cout << intVec2.size() << std::endl;      // 10

std::vector<int> intVec3{10};
std::cout << intVec3.size() << std::endl;      // 1

std::vector<int> intVec4{5, 2011};
std::cout << intVec4.size() << std::endl;      // 2
```

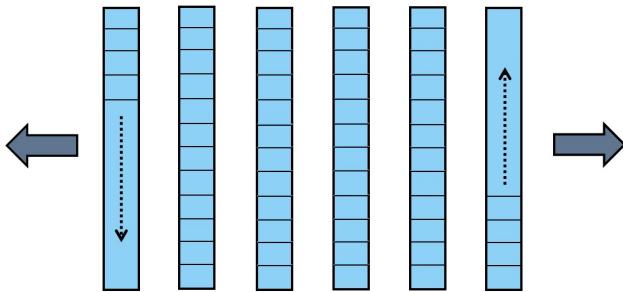
`std::vector` vec has a few methods to access its elements. With `vec.front()` you get the first element, with `vec.back()` you get the last element of vec. To read or write the $(n+1)$ -th element of vec, you can use the index operator `vec[n]`

or the method `vec.at(n)`. The second one checks the boundaries of `vec`, so that you eventually get a `std::out_of_range` exception.

Besides the index operator, `std::vector` offers additional methods to assign, insert, create or remove elements. See the following overview.

Method	Modify the elements of a <code>std::vector</code>
<code>vec.assign(...)</code>	Assigns one or more elements, a range or an initialiser list.
<code>vec.clear()</code>	Removes all elements from <code>vec</code> .
<code>vec.emplace(pos, args ...)</code>	Creates a new element before <code>pos</code> with the <code>args</code> in <code>vec</code> and returns the new position of the element.
<code>vec.emplace_back(args ...)</code>	Creates a new element in <code>vec</code> with <code>args ...</code> .
<code>vec.erase(...)</code>	Removes one element or a range and returns the next position.
<code>vec.insert(pos, ...)</code>	Inserts one or more elements, a range or an initialiser list and returns the new position of the element.
<code>vec.pop_back()</code>	Removes the last element.
<code>vec.push_back(elem)</code>	Adds a copy of <code>elem</code> at the end of <code>vec</code> .

Deques



[`std::deque`](#), which typically consists of a sequence of fixed-sized arrays, is quite similar to `std::vector`. `std::deque` need the header `<deque>`. The `std::deque` has three additional methods `deq.push_front(elem)`, `deq.pop_front()` and `deq.emplace_front(args...)` to add or remove elements at its beginning.

std::deque

```
// deque.cpp
...
#include <deque>
...
struct MyInt{
    MyInt(int i): myInt(i){};
    int myInt;
};

std::deque<MyInt> myIntDeq;

myIntDeq.push_back(MyInt(5));
myIntDeq.emplace_back(1);
std::cout << myIntDeq.size() << std::endl;      // 2

std::deque<MyInt> intDeq;

intDeq.assign({1, 2, 3});
for (auto v: intDeq) std::cout << v << " ";   // 1 2 3

intDeq.insert(intDeq.begin(), 0);
for (auto v: intDeq) std::cout << v << " ";   // 0 1 2 3

intDeq.insert(intDeq.begin() + 4, 4);
for (auto v: intDeq) std::cout << v << " ";   // 0 1 2 3 4

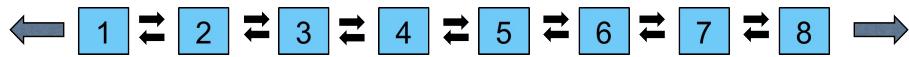
intDeq.insert(intDeq.end(), {5, 6, 7, 8, 9, 10, 11});
for (auto v: intDeq) std::cout << v << " ";   // 0 1 2 3 4 5 6 7 8 9 10 11

for (auto revIt= intDeq.rbegin(); revIt != intDeq.rend(); ++revIt)
    std::cout << *revIt << " ";           // 11 10 9 8 7 6 5 4 3 2 1 0

intDeq.pop_back();
for (auto v: intDeq) std::cout << v << " ";   // 0 1 2 3 4 5 6 7 8 9 10

intDeq.push_front(-1);
for (auto v: intDeq) std::cout << v << " ";   // -1 0 1 2 3 4 5 6 7 8 9 10
```

Lists



`std::list` is a doubled linked list. `std::list` needs the header `<list>`.

Although it has a similar interface to `std::vector` or `std::deque`, `std::list` is quite different to both of them. That's due to its structure.

`std::list` makes the following points unique:

- It supports no random access.
- The access of an arbitrary element is slow because you have to iterate in the worst case through the whole list.
- To add or remove an element is fast, if the iterator points to the right place.
- If you add or remove an element, the iterator keeps valid.

Because of its special structure, `std::list` has a few special methods.

Special methods of <code>std::list</code>	
Method	Description
<code>lis.merge(c)</code>	Merges the sorted list <code>c</code> into the sorted list <code>lis</code> , so that <code>lis</code> keeps sorted.
<code>lis.merge(c, op)</code>	Merges the sorted list <code>c</code> into the sorted list <code>lis</code> , so that <code>lis</code> keeps sorted. Uses <code>op</code> as sorting criteria.
<code>lis.remove(val)</code>	Removes all elements from <code>lis</code> with value <code>val</code> .
<code>lis.remove_if(pre)</code>	Removes all elements from <code>lis</code> , fulfilling the predicate <code>pre</code> .
<code>lis.splice(pos, ...)</code>	Splits the elements in <code>lis</code> before <code>pos</code> . The elements can be single elements, ranges or lists.
<code>lis.unique()</code>	Removes adjacent element with the same value.

`lis.unique(pre)` Removes adjacent elements, fulfilling the predicate `pre`.

Here are a few of the methods in a code snippet.

```
std::list  
// list.cpp  
...  
#include <list>  
...  
std::list<int> list1{15, 2, 18, 19, 4, 15, 1, 3, 18, 5,  
        4, 7, 17, 9, 16, 8, 6, 6, 17, 1, 2};  
  
list1.sort();  
for (auto l: list1) std::cout << l << " ";  
    // 1 1 2 2 3 4 4 5 6 6 7 8 9 15 15 16 17 17 18 18 19  
  
list1.unique();  
for (auto l: list1) std::cout << l << " ";  
    // 1 2 3 4 5 6 7 8 9 15 16 17 18 19  
  
std::list<int> list2{10, 11, 12, 13, 14};  
list1.splice(std::find(list1.begin(), list1.end(), 15), list2);  
for (auto l: list1) std::cout << l << " ";  
    // 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Forward Lists



[`std::forward_list`](#) is a singly linked list, which needs the header `<forward_list>`. `std::forward_list` has a drastically reduced interface and is optimised for minimal memory requirements.

`std::forward_list` has a lot in common with [`std::list`](#):

- It supports no random access.
- The access of an arbitrary element is slow because in the worst case you have to iterate forward through the whole list.
- To add or remove an element is fast, if the iterator points to the right place.
- If you add or remove an element, the iterator keeps valid.
- Operations always refer to the beginning of the `std::forward_list` or the position past the current element.

The characteristic that you can iterator a `std::forward_list` forward has a great impact. So the iterators cannot be decremented and therefore, operations like `It -`

- on iterators are not supported. For the same reason, `std::forward_list` has no backward iterator. `std::forward_list` is the only sequence container which doesn't know its size.



`std::forward_list` has a very special domain

`std::forward_list` is the replacement for single linked lists. It's optimised for minimal memory management and performance if the insertion, extraction or movement of elements only affect adjacent elements. This is typical for sort algorithm.

The special methods of `std::forward_list`.

Method	Special methods of <code>std::forward_list</code>	Description
<code>forw.before_begin()</code>		Returns an iterator before the first element.
<code>forw.emplace_after(pos, args...)</code>		Creates an element after pos with the arguments args....
<code>forw.emplace_front(args...)</code>		Creates an element at the beginning of <code>forw</code> with the arguments args....
<code>forw.erase_after(pos, ...)</code>		Removes from <code>forw</code> the element pos or a range of elements, starting with pos.
<code>forw.insert_after(pos, ...)</code>		Inserts after pos new elements. These elements can be single elements, ranges or initialiser lists.
<code>forw.merge(c)</code>		Merges the sorted forward list c into the sorted forward list <code>forw</code> , so that <code>forw</code> keeps sorted.
<code>forw.merge(c, op)</code>		Merges the forward sorted list c into the forward sorted list <code>forw</code> , so that <code>forw</code> keeps sorted. Uses os as sorting criteria.

<code>forw.splice_after(pos, ...)</code>	Splits the elements in <code>forw</code> before <code>pos</code> . The elements can be single elements, ranges or lists.
<code>forw.unique()</code>	Removes adjacent element with the same value.
<code>forw.unique(pre)</code>	Removes adjacent elements, fulfilling the predicate <code>pre</code> .

Let's have a look at the unique methods of `std::forward_list`.

```

std::forward_list
-----  

// forwardList.cpp
...
#include<forward_list>
...
using std::cout;

std::forward_list<int> forw;
std::cout << forw.empty() << std::endl; // true

forw.push_front(7);
forw.push_front(6);
forw.push_front(5);
forw.push_front(4);
forw.push_front(3);
forw.push_front(2);
forw.push_front(1);
for (auto i: forw) cout << i << " "; // 1 2 3 4 5 6 7

forw.erase_after(forw.before_begin());
cout << forw.front(); // 2

std::forward_list<int> forw2;
forw2.insert_after(forw2.before_begin(), 1);
forw2.insert_after(++forw2.before_begin(), 2);
forw2.insert_after(++(++forw2.before_begin()), 3);
forw2.push_front(1000);
for (auto i= forw2.cbegin(); i != forw2.cend(); ++i) cout << *i << " ";
// 1000 1 2 3

auto IteratorTo5= std::find(forw.begin(), forw.end(), 5);
forw.splice_after(IteratorTo5, std::move(forw));
for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
// 2 3 4 5 1000 1 2 3 6 7

forw.sort();
for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
// 1 2 2 3 3 4 5 6 7 1000

forw.reverse();
for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
// 1000 7 6 5 4 3 3 2 2 1

```

```
forw.unique();
for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
// 1000 7 6 5 4 3 2 1
```

5. Associative Containers

C++ has eight different [associative containers](#). Four of them are associative containers with sorted keys: `std::set`, `std::map`, `std::multiset` and `std::multimap`. The other four are associative containers with unsorted keys: `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset` and `std::unordered_multimap`. The associative containers are special containers. That means they support all of the operations described in the chapter [Interface of all containers](#).

Overview

All eight ordered and unordered containers have in common that they associate a key with a value. You can use the key to get the value. To get a classification of the associative containers, three simple questions need to be answered:

- Are the keys sorted?
- Does the key have an associated value?
- Can a key appear more than once?

The following table with $2^3 = 8$ rows gives the answers to the three questions. I answer a fourth question in the table. How fast is the access time of a key in the best case?

Associative container	Characteristics for associative containers				Access time
	Sorted	Associated value	More identical keys		
<code>std::set</code>	yes	no	no		logarithmic
<code>std::unordered_set</code>	no	no	no		constant
<code>std::map</code>	yes	yes	no		logarithmic
<code>std::unordered_map</code>	no	yes	no		constant

<code>std::multiset</code>	yes	no	yes	logarithmic
<code>std::unordered_multiset</code>	no	no	yes	constant
<code>std::multimap</code>	yes	yes	yes	logarithmic
<code>std::unordered_multimap</code>	no	yes	yes	constant

Since C++98, C++ has ordered associative containers; with C++11, C++ has in addition unordered associative containers. Both classes have a very similar interface. That's the reason that the following code sample is identical for `std::map` and `std::unordered_map`. To be more precise, the interface of `std::unordered_map` is a superset of the interface of `std::map`. The same holds for the remaining three unordered associative containers. So the porting of your code from the ordered to unordered containers is quite easy.

You can initialise the containers with an initialiser list and add new elements with the index operator. To access the first element of the key/value pair `p`, you have `p.first`, and for the second element, you have `p.second`. `p.first` is the key and `p.second` is the associated value of the pair.

std::map versus std::unordered_map

```
// orderedUnorderedComparison.cpp
...
#include <map>
#include <unordered_map>

// std::map

std::map<std::string, int> m {{"Dijkstra", 1972}, {"Scott", 1976}};
m["Ritchie"] = 1983;
std::cout << m["Ritchie"]; // 1983
for(auto p : m) std::cout << "{" << p.first << "," << p.second << "}";
// {Dijkstra,1972},{Ritchie,1983},{Scott,1976}
m.erase("Scott");
for(auto p : m) std::cout << "{" << p.first << "," << p.second << "}";
// {Dijkstra,1972},{Ritchie,1983}
m.clear();
std::cout << m.size() << std::endl; // 0

// std::unordered_map

std::unordered_map<std::string, int> um {{"Dijkstra", 1972}, {"Scott", 1976}};
um["Ritchie"] = 1983;
std::cout << um["Ritchie"]; // 1983
for(auto p : um) std::cout << "{" << p.first << "," << p.second << "}";
// {Ritchie,1983},{Scott,1976},{Dijkstra,1972}
```

```

um.erase("Scott");
for(auto p : um) std::cout << "{" << p.first << "," << p.second << "}";
    // {Ritchie, 1983}, {Dijkstra, 1972}
um.clear();
std::cout << um.size() << std::endl; // 0

```

There is a subtle difference between the two program executions: The keys of the `std::map` are ordered, the keys of the `std::unordered_map` are unordered. The question is: Why do we have such similar containers in C++? I already pointed it out in the [table](#). The reason is so often the same: performance. The access time to the keys of a unordered associative container is constant and therefore independent of the size of the container. If the containers are big enough, the performance difference is significant. Have a look at the section about the [performance](#).

Contains

The member function `associativeContainer.contains(ele)` checks if `associativeContainer` has the element `ele`.

Check if an associative container has an element

```

// containsElement.cpp
...
template <typename AssozCont>
bool containsElement5(const AssozCont& assozCont) {
    return assozCont.contains(5);
}

std::set<int> mySet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::cout << "containsElement5(mySet): "
    << containsElement5(mySet); // true

std::unordered_set<int> myUnordSet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::cout << "containsElement5(myUnordSet): "
    << containsElement5(myUnordSet); // true

std::map<int, std::string> myMap{ {1, "red"}, {2, "blue"}, {3, "green"} };
std::cout << "containsElement5(myMap): "
    << containsElement5(myMap); // false

std::unordered_map<int, std::string> myUnordMap{ {1, "red"}, {2, "blue"}, {3, "green"} };
std::cout << "containsElement5(myUnordMap): "
    << containsElement5(myUnordMap); // false

```

Insertion and Deletion

The insertion (`insert` and `emplace`) and deletion (`erase`) of elements in associative containers are similar to the rules of a [std::vector](#). For associative container which can have a key only once, the insertion fails if the key is already

in the container. Additionally, ordered associative containers support a special function `ordAssCont.erase(key)`, which removes all pairs with the key and returns their number. See the usage of the function.

Insertion and Deletion

```
// associativeContainerModify.cpp
...
#include <set>
...
std::multiset<int> mySet{3, 1, 5, 3, 4, 5, 1, 4, 4, 3, 2, 2, 7, 6, 4, 3, 6};
for (auto s: mySet) std::cout << s << " ";
// 1 1 2 2 3 3 3 3 4 4 4 5 5 6 6 7

mySet.insert(8);
std::array<int, 5> myArr{10, 11, 12, 13, 14};
mySet.insert(myArr.begin(), myArr.begin() + 3);
mySet.insert({22, 21, 20});
for (auto s: mySet) std::cout << s << " ";
// 1 1 2 2 3 3 3 3 4 4 4 4 5 5 6 6 7 10 11 12 20 21 22

std::cout << mySet.erase(4); // 4
mySet.erase(mySet.lower_bound(5), mySet.upper_bound(15));
for (auto s: mySet) std::cout << s << " ";
// 1 1 2 2 3 3 3 20 21 22
```

Ordered Associative Containers

Overview

The ordered associative container `std::map` and `std::multimap` associate their key with a value. Both are defined in the header `<map>`. `std::set` and `std::multiset` need the header `<set>`. This [Table](#) gives you the details.

All four ordered containers are parametrised by their type, their allocator and their comparison function. The containers have default values for the allocator and the comparison function, depending on the type. The declaration of `std::map` and `std::set` shows this very nicely.

```
template < class Key, class Val, class Comp = less<key>,
          class Alloc = allocator<pair<const Key, Val> >
class map;

template < class T, class Comp = less<T>,
          class Alloc = allocator<T> >
class set;
```

The declaration of both associative containers shows that `std::map` has an associated value. The key and the value are used for the default allocator: `allocator<pair<const Key, Val>>`. With a little bit more imagination, you can

derive more from the allocator. `std::map` has pairs of the type `std::pair<const key, val>`. The associated value `val` does not matter for the sort criteria: `less<key>`. All observations also hold for `std::multimap` and `std::multiset`.

Keys and Values

There are special rules for the key and the value of an ordered associative container.

The key has to be

- sortable (by default `<`),
- copyable and moveable.

The value has to be

- default constructible,
- copyable and moveable.

The with the key associated value builds a pair `p` so that you get with the member `p.first` the value `p.second`.

```
#include <map>
...
std::multimap<char, int> multiMap= {{'a', 10}, {'a', 20}, {'b', 30}};
for (auto p: multiMap) std::cout << "{" << p.first << "," << p.second << "}" " ;
// {a,10} {a,20} {b,30}
```

The Comparison Criterion

The default comparison criterion of the ordered associative containers is `std::less`. If you want to use a *user-defined* type as the key, you have to overload the operator `<`. It's sufficient to overload the operator `<` for your data type because the C++ runtime compares, with the help of the relation (`!(elem1<elem2 || elem2<elem1)`), two elements for equality.

You can specify the sorting criterion as a template argument. This sorting criterion must implement a *strict weak ordering*.



Strict weak ordering

Strict weak ordering for a sorting criterion on a set S is given if the following requirements are met.

- For s from S has to hold, that $s < s$ is not possible.
- For all s_1 and s_2 from S must hold: If $s_1 < s_2$, then $s_2 < s_1$ is not possible.
- For all s_1, s_2 and s_3 with $s_1 < s_2$ and $s_2 < s_3$ must hold: $s_1 < s_3$.
- For all s_1, s_2 and s_3 with s_1 not comparable with s_2 and s_2 not comparable with s_3 must hold: s_1 is not comparable with s_3 .

In opposite to the definition of the *strict weak ordering*, the usage of a comparison criterion with *strict weak ordering* is a lot simpler for a `std::map`.

```
#include <map>
...
std::map<int, std::string, std::greater<int>> int2Str{
    {5, "five"}, {1, "one"}, {4, "four"}, {3, "three"},  
    {2, "two"}, {7, "seven"}, {6, "six"} };
for (auto p: int2Str) std::cout << "{" << p.first << "," << p.second << "}" <<  
    // {7,seven} {6,six} {5,five} {4,four} {3,three} {2,two} {1,one}
```

Special Search Functions

Ordered associative containers are optimized for searching. So they offer unique search functions.

Special search functions of the ordered associative containers	
Search function	Description
<code>ordAssCont.count(key)</code>	Returns the number of values with the key.
<code>ordAssCont.find(key)</code>	Returns the iterator of key in <code>ordAssCont</code> . If there is no key in <code>ordAssCont</code> it returns <code>ordAssCont.end()</code> .
<code>ordAssCont.lower_bound(key)</code>	Returns the iterator to the first key in <code>ordAssCont</code> in which key would be inserted.
<code>ordAssCont.upper_bound(key)</code>	Returns the last position of key in

ordAssCont in which key would be inserted.

ordAssCont.equal_range(key)

Returns the range ordAssCont.lower
bound(key) and
ordAssCont.upper_bound(key) in a
std::pair.

Now, the application of the special search functions.

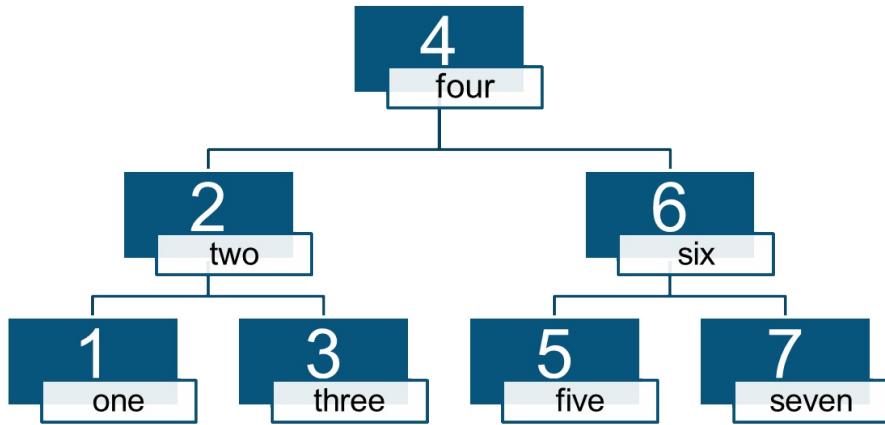
Search in an associative container

```
// associativeContainerSearch.cpp
...
#include <set>
...
std::multiset<int> mySet{3, 1, 5, 3, 4, 5, 1, 4, 4, 3, 2, 2, 7, 6, 4, 3, 6};

for (auto s: mySet) std::cout << s << " ";
// 1 1 2 2 3 3 3 3 4 4 4 5 5 6 6 7

mySet.erase(mySet.lower_bound(4), mySet.upper_bound(4));
for (auto s: mySet) std::cout << s << " ";
// 1 1 2 2 3 3 3 5 5 6 6 7
std::cout << mySet.count(3) << std::endl; // 4
std::cout << *mySet.find(3) << std::endl; // 3
std::cout << *mySet.lower_bound(3) << std::endl; // 3
std::cout << *mySet.upper_bound(3) << std::endl; // 5
auto pair= mySet.equal_range(3);
std::cout << "(" << *pair.first << "," << *pair.second << ")"; // (3,5)
```

std::map



`std::map` is by far the most frequently used associative container. The reason is simple. It combines often sufficient enough [performance](#) with a very convenient interface. You can access its elements via the index operator. If the key doesn't exist, `std::map` creates a key/value pair. For the value, the default constructor is used.



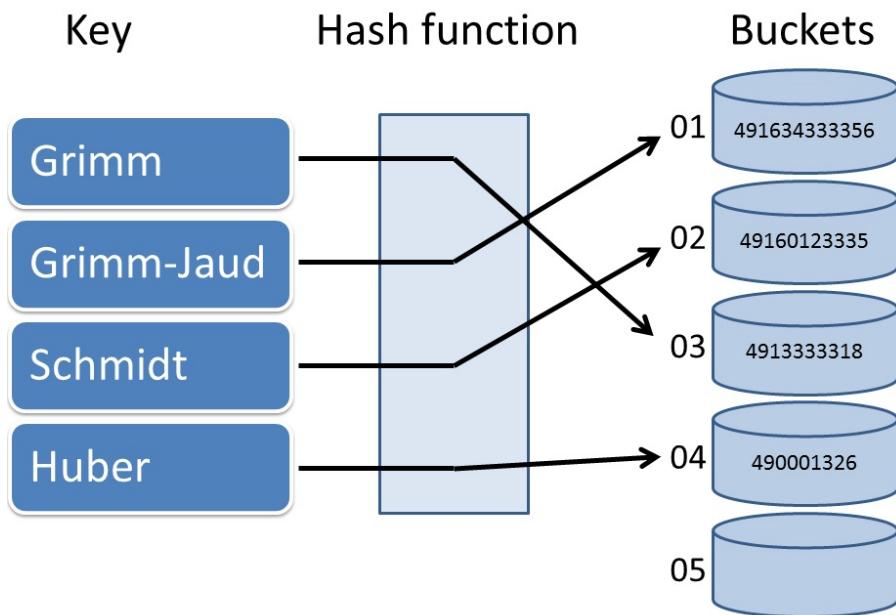
Consider `std::map` as a generalization of `std::vector`

Often `std::map` is called an associative array. Array, because `std::map` supports the index operator like a sequence container. The subtle difference is that its index is not restricted to a number like in the case of `std::vector`. Its index can be almost any arbitrary type.

The same observations hold for its namesake `std::unordered_map`.

In addition to the index operator, `std::map` supports the `at` method. The access via `at` is checked. So if the request key doesn't exist in the `std::map`, a `std::out_of_range` exception is thrown.

Unordered Associative Containers



Overview

With the new C++11 standard, C++ has four unordered associative containers: `std::unordered_map`, `std::unordered_multimap`, `std::unordered_set` and `std::unordered_multiset`. They have a lot in common with their namesakes, the [ordered associative containers](#). The difference is that the unordered ones have a richer interface and their keys are not sorted.

This shows the declaration of a `std::unordered_map`.

```
template< class key, class val, class Hash= std::hash<key>,
          class KeyEqual= std::equal_to<key>,
          class Alloc= std::allocator<std::pair<const key, val>>>
class unordered_map;
```

Like `std::map`, `std::unordered_map` has an allocator, but `std::unordered_map` needs no comparison function. Instead `std::unordered_map` needs two additional functions: One, to determine the hash value of its key: `std::hash<key>` and second, to compare the keys for equality: `std::equal_to<key>`. Because of the three default template parameters , you have only to provide the type of the key and the value of the `std::unordered_map`: `std::unordered_map<char, int> unordMap`.

Keys and Values

There are special rules for the key and the value of an unordered associative container.

The key has to be

- equal comparable,
- available as hash value,
- copyable or moveable.

The value has to be

- default constructible,
- copyable or moveable.

Performance

Performance - that's the simple reason - why the unordered associative containers were so long missed in C++. In the example below, one million randomly created values are read from a 10 million big std::map and std::unordered_map. The impressive result is that the linear access time of an unordered associative container is 20 times faster than the access time of an ordered associative container. That is just the difference between constant and logarithmic complexity $O(\log n)$ of these operations.

Performancecomparison

```
// associativeContainerPerformance.cpp
...
#include <map>
#include <unordered_map>
...
using std::chrono::duration;
static const long long mapSize= 100000000;
static const long long accSize= 1000000;
...
// read 1 million arbitrary values from a std::map
// with 10 million values from randValues
auto start= std::chrono::system_clock::now();
for (long long i= 0; i < accSize; ++i){myMap[randValues[i]];}
duration<double> dur= std::chrono::system_clock::now() - start;
std::cout << dur.count() << " sec"; // 9.18997 sec

// read 1 million arbitrary values from a std::unordered_map
// with 10 million values
auto start2= std::chrono::system_clock::now();
for (long long i= 0; i < accSize; ++i){ myUnorderedMap[randValues[i]];}
duration<double> dur2= std::chrono::system_clock::now() - start2;
std::cout << dur2.count() << " sec"; // 0.411334 sec
```

The Hash Function

The reason for the constant access time of the unordered associative container is the hash function, which is schematically shown [here](#). The hash function maps the key to its value the so-called has value. A hash function is good if it produces as few collisions as possible and equally distributes the keys onto the buckets. Because the execution of the hash function takes a constant amount of time, the access of the elements is in the base case also constant.

The hash function

- is already defined for the built-in types like boolean, natural numbers and floating point numbers,
- is available for `std::string` and `std::wstring`,
- generates for a C string `const char` a hash value of the pointer address,
- can be defined for *user-defined* data types.

For *user-defined* types which are used as a key for an unordered associative container, you have to keep two requirements to keep in mind. They need a hash function and have to be comparable to equal.

A custom hash function

```
// unorderedMapHash.cpp
...
#include <unordered_map>
...
struct MyInt{
    MyInt(int v):val(v){}
    bool operator== (const MyInt& other) const {
        return val == other.val;
    }
    int val;
};

struct MyHash{
    std::size_t operator()(MyInt m) const {
        std::hash<int> hashVal;
        return hashVal(m.val);
    }
};

std::ostream& operator << (std::ostream& st, const MyInt& myIn){
    st << myIn.val ;
    return st;
}

typedef std::unordered_map<MyInt, int, MyHash> MyIntMap;
MyIntMap myMap{{MyInt(-2), -2}, {MyInt(-1), -1}, {MyInt(0), 0}, {MyInt(1), 1}};

for(auto m : myMap) std::cout << "{" << m.first << "," << m.second << "}" " ;
// {MyInt(1),1} {MyInt(0),0} {MyInt(-1),-1} {MyInt(-2),-2}
```

```
std::cout << myMap[MyInt(-2)] << std::endl; // -2
```

The Details

The unordered associative containers store their indices in buckets. In which bucket the index goes is due to the hash function, which maps the key to the index. If different keys are mapped to the same index, it's called a collision. The hash function tries to avoid this.

Indices are typically stored in the bucket as a linked list. Since the access to the bucket is constant, the access in the bucket is linear. The number of buckets is called capacity, the average number of elements for each bucket is called the load factor. In general, the C++ runtime generates new buckets if the load factor is greater than 1. This process is called rehashing and can also explicitly be triggered:

Details to the hash function

```
// hashInfo.cpp
...
#include <unordered_set>
...
using namespace std;

void getInfo(const unordered_set<int>& hash){
    cout << "hash.bucket_count(): " << hash.bucket_count();
    cout << "hash.load_factor(): " << hash.load_factor();
}

unordered_set<int> hash;
cout << hash.max_load_factor() << endl; // 1

getInfo(hash);
// hash.bucket_count(): 1
// hash.load_factor(): 0

hash.insert(500);
cout << hash.bucket(500) << endl; // 5

// add 100 arbitrary values
fillHash(hash, 100);

getInfo(hash);
// hash.bucket_count(): 109
// hash.load_factor(): 0.88908

hash.rehash(500);

getInfo(hash);
// hash.bucket_count(): 541
// hash.load_factor(): 0.17298
cout << hash.bucket(500); // 500
```

With the method `max_load_factor`, you can read and set the load factor. So you can influence the probability of collisions and rehashing. I want to emphasise one point in the short example above. The key 500 is at first in the 5th bucket, but after rehashing is in the 500th bucket.

6. Adaptors for Containers

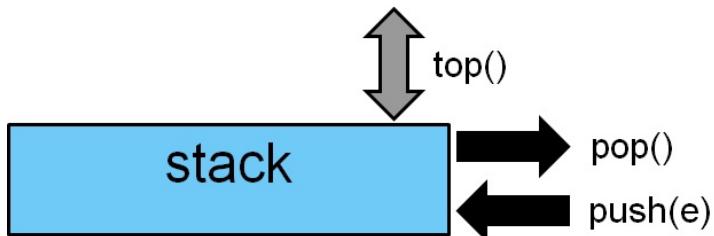
C++ has with `std::stack`, `std::queue` and `std::priority_queue` three special sequence containers. I guess, most of you know these classic data structures from your education.

The adaptors for containers

- support a reduced interface for existing sequence containers,
- can not be used with algorithms of the Standard Template Library,
- are class templates which are parametrised by their data type and their container (`std::vector`, `std::list` and `std::deque`),
- use by default `std::deque` as the internal sequence container:

```
template <typename T, typename Container= deque<T>>
class stack;
```

Stack



The `std::stack` follows the LIFO principle (**Last In First Out**). The stack `sta`, which needs the header `<stack>`, has three special methods.

With `sta.push(e)` you can insert a new element `e` at the top of the stack, remove it from the top with `sta.pop()` and reference it with `sta.top()`. The stack supports the comparison operators and knows its size. The operations of the stack have constant [complexity](#).

std::stack

```
// stack.cpp
...
#include <stack>
```

```

...
std::stack<int> myStack;

std::cout << myStack.empty() << std::endl;    // true
std::cout << myStack.size() << std::endl;      // 0

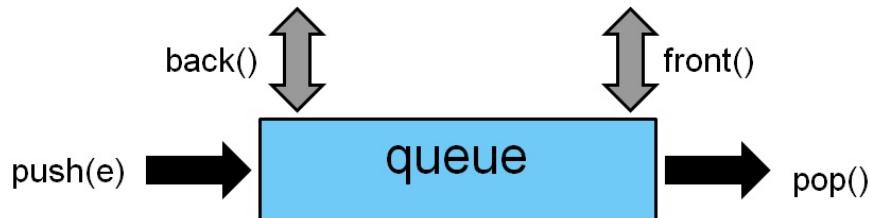
myStack.push(1);
myStack.push(2);
myStack.push(3);
std::cout << myStack.top() << std::endl;      // 3

while (!myStack.empty()){
    std::cout << myStack.top() << " ";
    myStack.pop();
}
                                         // 3 2 1

std::cout << myStack.empty() << std::endl;    // true
std::cout << myStack.size() << std::endl;      // 0

```

Queue



The [`std::queue`](#) follows the **FIFO principle (First In First Out)**. The queue `que`, which needs the header `<queue>`, has four special methods.

With `que.push(e)` you can insert an element `e` at the end of the queue and remove the first element from the queue with `que.pop()`. `que.back()` enables you to refer to the last element in the `que`, `que.front()` to the first element in the `que`. `std::queue` has similar characteristics as [`std::stack`](#). So you can compare `std::queue` instances and get their sizes. The operations of the queue have constant [complexity](#).

```

std::queue
// queue.cpp
...
#include <queue>
...
std::queue<int> myQueue;

std::cout << myQueue.empty() << std::endl;    // true
std::cout << myQueue.size() << std::endl;      // 0

myQueue.push(1);
myQueue.push(2);
myQueue.push(3);

```

```

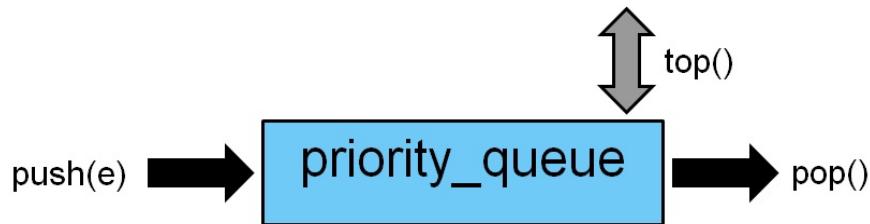
std::cout << myQueue.back() << std::endl;      // 3
std::cout << myQueue.front() << std::endl;      // 1

while (!myQueue.empty()){
    std::cout << myQueue.back() << " ";
    std::cout << myQueue.front() << " : ";
    myQueue.pop();
}                                                 // 3 1 : 3 2 : 3 3

std::cout << myQueue.empty() << std::endl;      // true
std::cout << myQueue.size() << std::endl;      // 0

```

Priority Queue



The [`std::priority_queue`](#) is a reduced `std::queue`. It needs the header `<queue>`.

The difference to the `std::queue` is, that their biggest element is always at the top of the priority queue. `std::priority_queue` `pri` uses by default the comparison operator `std::less`. Similar to `std::queue`, `pri.push(e)` inserts a new element `e` into the priority queue. `pri.pop()` removes the first element of the `pri`, but does that with logarithmic [complexity](#). With `pri.top()` you can reference the first element in the priority queue, which is the greatest one. The `std::priority_queue` knows its size, but didn't support the comparison operator on their instances.

```

std::priority_queue

```

```

// priorityQueue.cpp
...
#include <queue>
...
std::priority_queue<int> myPriorityQueue;

std::cout << myPriorityQueue.empty() << std::endl;      // true
std::cout << myPriorityQueue.size() << std::endl;      // 0

myPriorityQueue.push(3);
myPriorityQueue.push(1);
myPriorityQueue.push(2);
std::cout << myPriorityQueue.top() << std::endl;      // 3

while (!myPriorityQueue.empty()){
    std::cout << myPriorityQueue.top() << " ";

```

```
myPriorityQueue.pop(); // 3 2 1
}
std::cout << myPriorityQueue.empty() << std::endl; // true
std::cout << myPriorityQueue.size() << std::endl; // 0
std::priority_queue<std::string, std::vector<std::string>,
                    std::greater<std::string>> myPriorityQueue2;
myPriorityQueue2.push("Only");
myPriorityQueue2.push("for");
myPriorityQueue2.push("testing");
myPriorityQueue2.push("purpose");
myPriorityQueue2.push(".");
while (!myPriorityQueue2.empty()){
    std::cout << myPriorityQueue2.top() << " ";
    myPriorityQueue2.pop();
}
// . Only for purpose testing
```

7. Views on Contiguous Sequences

`std::span` is an object that refers to a contiguous sequence of objects. A C-array, a `std::array`, a `std::vector`, and a `std::string` store their objects in a contiguous memory block. `std::span` is often called view and is never an owner. The contiguous sequence of objects can be a C-array, a pointer and a length, or a `std::vector`. A typical implementation of `std::span` holds a pointer to its first element and its length. Thanks to `std::span` you can access subsequences.



`std::span` does not decay

When you invoke a function taking a C-array, decay occurs. The function takes the C-array via a pointer to its first element. The C-array to pointer conversion is very error-prone because all length information to the C-array is lost.

In contrast, a `std::span` knows its length.

```
// copySpanArray.cpp
...
#include <span>

template <typename T>
void copy_n(const T* p, T* q, int n){}

template <typename T>
void copy(std::span<const T> src, std::span<T> des){}

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

copy_n(arr1, arr2, 3);      // (1)
copy(arr1, arr2);          // (2)
```

In contrast to the C-array (1), a function taking a C-array via a `std::span` (2) does not need an explicit argument for its length.

`std::span` has various member functions to access the underlying contiguous sequence of objects.

Member functions of a std::span span

Member function	Description
span.front()	Accesses the first element
span.back()	Accesses the last element
span[i]	Accesses the i-th element
span.data()	Returns a pointer to the contiguous sequence
span.size()	Returns the number of elements
span.size_bytes()	Returns the size of the contiguous sequence
span.empty()	Returns if the sequence is empty
span<cnt>.first(), span.first(cnt)	Returns a subview of the first cnt objects
span<cnt>.last(), span.last(cnt)	Returns a subview of the last cnt objects
span<fir, cnt>.subspan(), span.subspan(fir, cnt)	Returns a subview of the first cnt objects, starting at fir

std::span taking different arguments

```
// quadSpan.cpp
...
#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";
}

std::cout << std::endl;

int arr[] = {1, 2, 3, 4}; // (1)
printMe(arr);
```

```
std::vector vec{1, 2, 3, 4, 5};           // (2)
printMe(vec);

std::array arr2{1, 2, 3, 4, 5, 6};       // (3)
printMe(arr2);
```

A `std::span` can be initialized with a C-array, , and a `std::vector` (1), and a `std::array` (2).



Prefer `std::string_view` to `std::span`

A `std::string` is a contiguous sequence of characters. Consequently, you could initialize a `std::span` with a `std::string`. You should prefer `std::string_view` to `std::span`, because a `std::string_view` represents a view of a sequence of characters and not a sequence of objects such as `std::span`. The interface of a `std::string_view` is string-like, but the interface of a `std::span` is quite generic.

8. Iterators

On the one hand [iterators](#) are generalisations of pointers which represents positions in a container. On the other hand, they provide powerful iteration and random access in a container.

Iterators are the glue between the generic containers and the generic algorithms of the Standard Template Library.

Iterators support the following operations:

- * Returns the element at the current position.
- ==, != Compares two positions.
- = Assigns a new value to an iterator.

The range-based for-loop uses the iterators implicitly.

Because iterators are not checked, they have the same issues as pointers.

```
std::vector<int> vec{1, 23, 3, 3, 3, 4, 5};  
std::deque<int> deq;  
  
// Start iterator bigger than end iterator  
std::copy(vec.begin() + 2, vec.begin(), deq.begin());  
  
// Target container too small  
std::copy(vec.begin(), vec.end(), deq.end());
```

Categories

Their capabilities can categorise iterator. C++ has forward, bidirectional and random access iterators. With the forward iterator, you can iterate the container forward, with the bidirectional iterator, in both directions. With the random access iterator, you can directly access an arbitrary element. In particular, this means for the last one, that you can use iterator arithmetic and ordering

comparisons (e.g.: `<`). The category of an iterator depends on the type of the container used.

In the table below is a representation of containers and their iterator categories. The bidirectional iterator includes the forward iterator functionalities, and the random access iterator includes the forward and the bidirectional iterator functionalities. `It` and `It2` are iterators, `n` is a natural number.

The iterator categories of the container		
Iterator category	Properties	Containers
Forward iterator	<code>++It, It++, *It</code> <code>It == It2, It != It2</code>	<code>std::unordered_set</code> <code>std::unordered_map</code> <code>std::unordered_multiset</code> <code>std::unordered_multimap</code> <code>std::forward_list</code>
Bidirectional iterator	<code>--It, It--</code>	<code>std::set</code> <code>std::map</code> <code>std::multiset</code> <code>std::multimap</code> <code>std::list</code>
Random access iterator	<code>It[i]</code> <code>It+= n, It-= n</code> <code>It+n, It-n</code> <code>n+It</code> <code>It-It2</code> <code>It < It2, It <= It2, It > It2</code> <code>It >= It2</code>	<code>std::deque</code>
Contiguous iterator		<code>std::array</code> <code>std::vector</code> <code>std::string</code>

The input iterator and the output iterator are special forward iterators: they can read and write their pointed element only once.

Iterator Creation

Each container generates its suitable iterator on request. For example an `std::unordered_map` generates constant and non-constant forward iterators.

```
std::unordered_map<std::string, int>::iterator unMapIt= unordMap.begin();
std::unordered_map<std::string, int>::iterator unMapIt= unordMap.end();

std::unordered_map<std::string, int>::const_iterator unMapIt= unordMap.cbegin();
std::unordered_map<std::string, int>::const_iterator unMapIt= unordMap.cend();
```

In addition, `std::map` supports the backward iterators:

```
std::map<std::string, int>::reverse_iterator mapIt= map.rbegin();
std::map<std::string, int>::reverse_iterator mapIt= map.rend();

std::map<std::string, int>::const_reverse_iterator mapIt= map.crbegin();
std::map<std::string, int>::const_reverse_iterator mapIt= map.crend();
```



Use auto for iterator definition

Iterator definition is very labour intensive. The automatic type deduction with `auto` reduces the writing to the bare minimum.

```
std::map<std::string, int>::const_reverse_iterator
mapIt= map.crbegin();
auto mapIt2= map.crbegin();
```

The final example:

Iterator creation

```
// iteratorCreation.cpp
...
using namespace std;
...
map<string, int> myMap{{"Rainer", 1966}, {"Beatrix", 1966}, {"Juliette", 1997},
                      {"Marius", 1999};

auto endIt= myMap.end();
for (auto mapIt= myMap.begin(); mapIt != endIt; ++mapIt)
    cout << "{" << mapIt->first << "," << mapIt->second << "}";
    // {Beatrix,1966}, {Juliette,1997}, {Marius,1999}, {Rainer,1966}

vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
vector<int>::const_iterator vecEndIt= myVec.end();
```

```

vector<int>::iterator vecIt;
for (vecIt= myVec.begin(); vecIt != vecEndIt; ++vecIt) cout << *vecIt << " ";
// 1 2 3 4 5 6 7 8 9

vector<int>::const_reverse_iterator vecEndRevIt= myVec.rend();
vector<int>::reverse_iterator revIt;
for (revIt= myVec.rbegin(); revIt != vecEndRevIt; ++revIt) cout << *revIt << " ";
// 9 8 7 6 5 4 3 2 1

```

Useful Functions

The global functions `std::begin`, `std::end`, `std::prev`, `std::next`, `std::distance` and `std::advance` make your handling of the iterators a lot easier. Only the function `std::prev` requires a bidirectional iterator. All functions need the header `<iterator>`. The table gives you the overview.

Global function	Description
Useful functions for iterators	
<code>std::begin(cont)</code>	Returns a begin iterator to the container cont.
<code>std::end(cont)</code>	Returns an end iterator to the container cont.
<code>std::rbegin(cont)</code>	Returns a reverse begin iterator to the container cont.
<code>std::rend(cont)</code>	Returns a reverse end iterator to the container cont.
<code>std::cbegin(cont)</code>	Returns a constant begin iterator to the container cont
<code>std::cend(cont)</code>	Returns a constant end iterator to the container cont.
<code>std::crbegin(cont)</code>	Returns a reverse constant begin iterator to the container cont.
<code>std::crend(cont)</code>	Returns a reverse constant end iterator to the container cont.
<code>std::prev(it)</code>	Returns an iterator, which points to a position before it
<code>std::next(it)</code>	Returns an iterator, which points to a position after it

```
std::distance(fir,  
sec)           Returns the number of elements between fir and sec
```

```
std::advance(it, n)    Puts the iterator it n positions further.
```

Now, the application of the useful functions.

Helper functions for iterators

```
// iteratorUtilities.cpp  
...  
#include <iterator>  
...  
using std::cout;  
  
std::unordered_map<std::string, int> myMap{{"Rainer", 1966}, {"Beatrix", 1966},  
                                         {"Juliette", 1997}, {"Marius", 1999}};  
  
for (auto m: myMap) cout << "{" << m.first << "," << m.second << "}" ";  
     // {Juliette,1997},{Marius,1999},{Beatrix,1966},{Rainer,1966}  
  
auto mapItBegin= std::begin(myMap);  
cout << mapItBegin->first << " " << mapItBegin->second; // Juliette 1997  
  
auto mapIt= std::next(mapItBegin);  
cout << mapIt->first << " " << mapIt->second;           // Marius 1999  
cout << std::distance(mapItBegin, mapIt);                  // 1  
  
std::array<int, 10> myArr{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
for (auto a: myArr) std::cout << a << " ";                // 0 1 2 3 4 5 6 7 8 9  
  
auto arrItEnd= std::end(myArr);  
auto arrIt= std::prev(arrItEnd);  
cout << *arrIt << std::endl;                                // 9  
  
std::advance(arrIt, -5);  
cout << *arrIt;                                              // 4
```

Adaptors

Iterator adaptors enable the use of iterators in insert mode or with streams. They need the header `<iterator>`.

Insert iterators

With the three insert iterators `std::front_inserter`, `std::back_inserter` and `std::inserter` you can insert an element into a container at the beginning, at the end or an arbitrary position respectively. The memory for the elements will automatically be provided. The three map their functionality on the underlying methods of the container cont.

The table below gives you two pieces of information: Which methods of the containers are internally used and which iterators can be used depends on the container's type.

Name	The three insert iterators	
	internally used method	Container
<code>std::front_inserter(val)</code>	<code>cont.push_front(val)</code>	<code>std::deque</code> <code>std::list</code>
<code>std::back_inserter(val)</code>	<code>cont.push_back(val)</code>	<code>std::vector</code> <code>std::deque</code> <code>std::list</code> <code>std::string</code>
<code>std::inserter(val, pos)</code>	<code>cont.insert(pos, val)</code>	<code>std::vector</code> <code>std::deque</code> <code>std::list</code> <code>std::string</code> <code>std::map</code> <code>std::set</code>

You can combine the algorithms in the STL with the three insert iterators.

```
#include <iterator>
...
std::deque<int> deq{5, 6, 7, 10, 11, 12};
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

std::copy(std::find(vec.begin(), vec.end(), 13),
          vec.end(), std::back_inserter(deq));

for (auto d: deq) std::cout << d << " ";
// 5 6 7 10 11 12 13 14 15

std::copy(std::find(vec.begin(), vec.end(), 8),
          std::find(vec.begin(), vec.end(), 10),
          std::inserter(deq,
                        std::find(deq.begin(), deq.end(), 10)));d
for (auto d: deq) std::cout << d << " ";
// 5 6 7 8 9 10 11 12 13 14 15

std::copy(vec.rbegin() + 11, vec.rend(),
          std::front_inserter(deq));
for (auto d: deq) std::cout << d << " ";
// 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Stream Iterators

Stream iterator adaptors can use streams as a data source or data sink. C++ offers two functions to create istream iterators and two to create ostream iterators. The created istream iterators behave like [input iterators](#), the ostream iterators like [insert iterators](#).

Function	The four stream iterators
Description	
<code>std::istream_iterator<T></code>	Creates an end-of-stream iterator.
<code>std::istream_iterator<T>(istream)</code>	Creates an istream iterator for <code>istream</code> .
<code>std::ostream_iterator<T>(ostream)</code>	Creates an ostream iterator for <code>ostream</code>
<code>std::ostream_iterator<T>(ostream, delim)</code>	Creates an ostream iterator for <code>ostream</code> with the delimiter <code>delim</code> .

Thanks to the stream iterator adapter you can directly read from or write to a stream.

The following interactive program fragment reads in an endless loop natural numbers from `std::cin` and pushes them onto the vector `myIntVec`. If the input is not a natural number, an error in the input stream will occur. All numbers in `myIntVec` are copied to `std::cout`, separated by `:`. Now you can see the result on the console.

```
#include <iostream>
...
std::vector<int> myIntVec;
std::istream_iterator<int> myIntStreamReader(std::cin);
std::istream_iterator<int> myEndIterator;

// Possible input
// 1
// 2
// 3
// 4
// z
while(myIntStreamReader != myEndIterator){
    myIntVec.push_back(*myIntStreamReader);
    ++myIntStreamReader;
}
```

```
std::copy(myIntVec.begin(), myIntVec.end(),
          std::ostream_iterator<int>(std::cout, ":"));  
// 1:2:3:4:
```

9. Callable Units



This chapter is intentionally not exhaustive

This book is about the C++ Standard library, therefore, will no go into the details of callable units. I provide only as much information as it's necessary to use them correctly in the algorithms of the Standard Template Library. An exhaustive discussion of callable units should be part of a book about the C++ core language.

Many of the STL algorithms and containers can be parametrised with callable units or short callables. A callable is something that behaves like a function. Not only are these functions but also function objects and lambda functions.

Predicates are special functions that return a boolean as 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. The same holds for functions. A function taking one argument is a unary function; a function taking two arguments is a binary function.



To change the elements of a container, your algorithm should take them by reference

Callables can receive their arguments by value or by reference from their container. To modify the elements of the container, they must address them directly, so it is necessary that the callable gets them by reference.

Functions

Functions are the simplest callables. They can have - apart from static variables - no state. Because the definition of a function is often widely separated from its use or even in a different translation unit, the compiler has fewer opportunities to optimise the resulting code.

```

void square(int& i){ i = i*i; }
std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::for_each(myVec.begin(), myVec.end(), square);
for (auto v: myVec) std::cout << v << " ";      // 1 4 9 16 25 36 49 64 81 100

```

Function Objects

At first, don't call them [functors](#). That's a *well-defined* term from the category theory.

[Function objects](#) are objects that behave like functions. They achieve this due to their call operator being implemented. 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 81 100

```



Instantiate function objects to use them

It's a common error that only the name of the function object (**Square**) and not the instance of function object (**Square()**) is used in an algorithm: `std::for_each(myVec.begin(), myVec.end(), Square)`. That's of course an error. You have to use the instance:
`std::for_each(myVec.begin(), myVec.end(), Square())`

Predefined Function Objects

C++ offers a bunch of predefined function objects. They need the header `<functional>`. These predefined function objects are very useful to change the default behaviour of the containers. For example, the keys of the ordered associative containers are by default sorted with the predefined function object `std::less`. But you may want to use `std::greater` instead:

```

std::map<int, std::string> myDefaultMap;           // std::less<int>
std::map<int, std::string> std::greater<int> mySpecialMap; // std::greater<int>

```

There are function objects in the Standard Template Library for arithmetic, logic and bitwise operations, and also for negation and comparison.

Predefined function objects

Function object for Representative

Negation	<code>std::negate<T>()</code>
Arithmetic	<code>std::plus<T>(), std::minus<T>()</code> <code>std::multiplies<T>(), std::divides<T>()</code> <code>std::modulus<T>()</code>
Comparison	<code>std::equal_to<T>(), std::not_equal_to<T>()</code> <code>std::less<T>(), std::greater<T>()</code> <code>std::less_equal<T>(), std::greater_equal<T>()</code>
Logical	<code>std::logical_not<T>()</code> <code>std::logical_and<T>(), std::logical_or<T>()</code>
Bitwise	<code>std::bit_and<T>(), std::bit_or<T>()</code> <code>std::bit_xor<T>()</code>

Lambda Functions

[Lambda functions](#) provide in-place functionality. The compiler gets a lot of insight and has therefore great optimisation potential. Lambda functions can receive their arguments by value or by reference. They can capture their environment by value, by reference and with C++14 by [move](#).

```
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
```



Lambda functions should be your first choice

If the functionality of your callable is short and self-explanatory, use a lambda function. A lambda function is, in general, faster and easier to understand.

10. Algorithms

The Standard Template Library has a large number of [algorithms](#) to work with containers and their elements. As the algorithms are function templates, they are independent of the type of the container elements. The glue between the containers and algorithms are the [iterators](#). If your container supports the interface of an STL container, you can apply the algorithms on your container.

Generic programming with the algorithmn

```
// algorithm.cpp
...
#include <algorithm>
...
template <typename Cont, typename T>
void doTheSame(Cont cont, T t){
    for (const auto c: cont) std::cout << c << " ";
    std::cout << cont.size() << std::endl;
    std::reverse(cont.begin(), cont.end());
    for (const auto c: cont) std::cout << c << " ";
    std::reverse(cont.begin(), cont.end());
    for (const auto c: cont) std::cout << c << " ";
    auto It= std::find(cont.begin(), cont.end(), t);
    std::reverse(It, cont.end());
    for (const auto c: cont) std::cout << c << " ";
}

std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::deque<std::string> myDeq({"A", "B", "C", "D", "E", "F", "G", "H", "I"});
std::list<char> myList({'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'});

doTheSame(myVec, 5);
// 1 2 3 4 5 6 7 8 9 10
// 10
// 10 9 8 7 6 5 4 3 2 1
// 1 2 3 4 5 6 7 8 9 10
// 1 2 3 4 10 9 8 7 6 5

doTheSame(myDeq, "D");
// A B C D E F G H I
// 9
// I H G F E D C B A
// A B C D E F G H I
// A B C I H G F E D

doTheSame(myList, 'd');
// a b c d e f g h
// 8
// h g f e d c b a
// a b c d e f g h
// a b c h g f e d
```

Conventions

To use the algorithms, you have to keep a few rules in your head.

The algorithms are defined in various headers.

`<algorithm>`

Contains the general algorithms.

`<numeric>`

Contains the numeric algorithms.

Many of the algorithms have the name suffix `_if` and `_copy`.

`_if`

The algorithm can be parametrized by a [predicate](#).

`_copy`

The algorithm copies its elements in another range.

Algorithms like `auto num= std::count(InpIt first, InpIt last, const T& val)` return the number of elements that are equal to `val`. `num` is of type `iterator_traits<InpIt>::difference_type`. You have the guarantee that `num` is sufficient to hold the result. Because of the automatic return type deduction with `auto`, the compiler will give you the right types.



If the container uses an additional range, it has to be valid

The algorithm `std::copy_if` uses an iterator to the beginning of its destination range. This destination range has to be valid.



Naming conventions for the algorithms

I use a few naming conventions for the type of arguments and the return type of the algorithms to make them easier to read.

Signature of the algorithms	
Name	Description
Init	Input iterator
FwdIt	Forward iterator
BiIt	Bidirectional iterator
UnFunc	Unary callable
BiFunc	Binary callable
UnPre	Unary predicate
BiPre	Binary predicate
Search	The searcher encapsulates the search algorithm.
ValType	From the input range automatically deduced value type.
ExePol	Execution policy

Iterators are the Glue

Iterators define the range of the container on which the algorithms work. They describe a *half-open* range. In a *half-open* range the begin iterator points to the beginning, and the end iterator points to one position after the range.

The iterators can be categorized based on their capabilities. See the [Categories section of the Iterators chapter](#). The algorithms provide conditions to the iterators. Like in the case of `std::rotate`, most of the times a [forward iterator](#) is sufficient. But that doesn't hold for `std::reverse`. `std::reverse` requires a [bidirectional iterator](#).

Sequential, Parallel, or Parallel Execution with Vectorisation

By using an execution policy in C++17, you can specify whether the algorithm should run sequentially, in parallel, or parallel with vectorisation.



Availability of the Parallel STL

On the end of 2020, only the Microsoft Compiler implements the parallel version of the STL algorithm. If you use a different compiler, you have to install third-party frameworks such as HPX. The [HPX \(High Performance ParalleX\)](#) is a framework that is a general-purpose C++ runtime system for parallel and distributed applications of any scale. HPX has already implemented the parallel STL in a different namespace.

Execution Policies

The policy tag specifies whether an algorithm should run sequentially, in parallel, or in parallel with vectorisation.

- `std::execution::seq`: runs the algorithm sequentially
- `std::execution::par`: runs the algorithm in parallel on multiple threads
- `std::execution::par_unseq`: runs the algorithm in parallel on multiple threads and allows the interleaving of individual loops; permits a vectorised version with [SIMD](#) (Single Instruction Multiple Data) extensions.

The following code snippet shows all execution policies.

The execution policy

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

// standard sequential sort
std::sort(v.begin(), v.end());

// sequential execution
std::sort(std::execution::seq, v.begin(), v.end());

// permitting parallel execution
std::sort(std::execution::par, v.begin(), v.end());

// permitting parallel and vectorised execution
std::sort(std::execution::par_unseq, v.begin(), v.end());
```

The example shows that you can still use the classic variant of `std::sort` without execution policy. Also, in C++17 you can specify explicitly whether the sequential, parallel, or the parallel and vectorised version should be used.



Parallel and Vectorised Execution

Whether an algorithm runs in a parallel and vectorised way depends on many factors. For example, it depends on whether the CPU and the operating system support SIMD instructions. Additionally, it depends on the compiler and the optimisation level that you use to translate your code.

The following example shows a simple loop for creating a new vector.

```
const int SIZE= 8;

int vec[] = {1, 2, 3, 4, 5, 6, 7, 8};
int res[] = {0, 0, 0, 0, 0, 0, 0, 0};

int main(){
    for (int i = 0; i < SIZE; ++i) {
        res[i] = vec[i] + 5;
    }
}
```

The expression `res[i] = vec[i] + 5` is the key line in this small example. Thanks to the [compiler Explorer](#) we can have a closer look at the assembler instructions generated by x86-64 clang 3.6.

Without Optimisation

Here are the assembler instructions. Each addition is done sequentially.

```
movslq -8(%rbp), %rax
movl   vec(,%rax,4), %ecx
addl   $5, %ecx
movslq -8(%rbp), %rax
movl   %ecx, res(,%rax,4)
```

With maximum Optimisation

By using the highest optimisation level, `-O3`, special registers such as `xmm0` that can hold 128 bits or 4 `ints` are used. This means that the addition takes place in parallel on four elements of the vector.

```
movdqa .LCPI0_0(%rip), %xmm0    # xmm0 = [5,5,5,5]
movdqa vec(%rip), %xmm1
paddd %xmm0, %xmm1
movdqa %xmm1, res(%rip)
paddd vec+16(%rip), %xmm0
movdqa %xmm0, res+16(%rip)
xorl  %eax, %eax
```

77 of the STL algorithms can be parametrised by an execution policy.

Algorithms with parallelised versions

Here are the 77 algorithms with parallelised versions.

The 77 algorithms with parallelised versions		
std::adjacent_difference	std::adjacent_find	std::all_of
std::any_of	std::copy	std::copy_if
std::copy_n	std::count	std::count_if
std::equal	std::exclusive_scan	std::fill
std::fill_n	std::find	std::find_end
std::find_first_of	std::find_if	std::find_if_not
std::for_each	std::for_each_n	std::generate
std::generate_n	std::includes	std::inclusive_scan
std::inner_product	std::inplace_merge	std::is_heap
std::is_heap_until	std::is_partitioned	std::is_sorted
std::is_sorted_until	std::lexicographical_compare	std::max_element
std::merge	std::min_element	std::minmax_element
std::mismatch	std::move	std::none_of
std::nth_element	std::partial_sort	std::partial_sort_copy
std::partition	std::partition_copy	std::reduce
std::remove	std::remove_copy	std::remove_copy_if
std::remove_if	std::replace	std::reverse
std::replace_copy_if	std::replace_if	std::rotate
std::reverse_copy	std::rotate	std::rotate_copy
std::search	std::search_n	std::set_difference
std::set_intersection	std::set_symmetric_difference	std::set_union
std::sort	std::stable_partition	std::stable_sort
std::swap_ranges	std::transform	std::transform_exclusive_scan
std::transform_inclusive_scan	std::transform_reduce	std::uninitialized_copy
std::uninitialized_copy_n	std::uninitialized_fill	std::uninitialized_fill_n
std::unique	std::unique_copy	



constexpr Container and Algorithms

C++20 supports the `constexpr` containers `std::vector` and `std::string`. `constexpr` means, that the member functions of both containers can be applied at compile-time. Additionally, the more than [100 algorithms](#) of the Standard Template Library are declared as `constexpr`.

for_each

`std::for_each` applies a unary callable to each element of its range. The range is given by the input iterators.

```
UnFunc std::for_each(InpIt first, InpIt second, UnFunc func)
void std::for_each(ExePol pol, FwdIt first, FwdIt second, UnFunc func)
```

`std::for_each`, when used without an explicit execution policy is a special algorithm because it returns its callable argument. If you invoke `std::for_each` with a function object, you can store the result of the function call directly in the function object.

```
InpIt std::for_each_n(InpIt first, Size n, UnFunc func)
FwdIt std::for_each_n(ExePol pol, FwdIt first, Size n, UnFunc func)
```

`std::for_each_n` is new with C++17 and applies a unary callable to the first `n` elements of its range. The range is given by an input iterator and a size.

std::for_each

```
// forEach.cpp
...
#include <algorithm>
...
template <typename T>
class ContInfo{
public:
    void operator()(T t){
        num++;
        sum+= t;
    }
    int getSum() const{ return sum; }
    int getSize() const{ return num; }
    double getMean() const{
        return static_cast<double>(sum)/static_cast<double>(num);
    }
private:
    T sum{0};
    int num{0};
};
```

```

std::vector<double> myVec{1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};
auto vecInfo= std::for_each(myVec.begin(), myVec.end(), ContInfo<double>());

std::cout << vecInfo.getSum() << std::endl;      // 49
std::cout << vecInfo.getSize() << std::endl;     // 9
std::cout << vecInfo.getMean() << std::endl;      // 5.5

std::array<int, 100> myArr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto arrInfo= std::for_each(myArr.begin(), myArr.end(), ContInfo<int>());

std::cout << arrInfo.getSum() << std::endl;      // 55
std::cout << arrInfo.getSize() << std::endl;     // 100
std::cout << arrInfo.getMean() << std::endl;      // 0.55

```

Non-Modifying Algorithms

Non-modifying algorithms are algorithms for searching and counting elements. However, you can also check properties on ranges, compare ranges or search for ranges within ranges.

Search Elements

You can search for elements in three different ways.

Returns an element in a range:

```

InpIt find(InpIt first, InpI last, const T& val)
InpIt find(ExePol pol, FwdIt first, FwdIt last, const T& val)

InpIt find_if(InpIt first, InpIt last, UnPred pred)
InpIt find_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred)

InpIt find_if_not(InpIt first, InpIt last, UnPred pre)
InpIt find_if_not(ExePol pol, FwdIt first, FwdIt last, UnPred pre)

```

Returns the first element of a range in a range:

```

FwdIt1 find_first_of(InpIt1 first1, InpIt1 last1,
                      FwdIt2 first2, FwdIt2 last2)
FwdIt1 find_first_of(ExePol pol, FwdIt1 first1, FwdIt1 last1,
                      FwdIt2 first2, FwdIt2 last2)

FwdIt1 find_first_of(InpIt1 first1, InpIt1 last1,
                      FwdIt2 first2, FwdIt2 last2, BiPre pre)
FwdIt1 find_first_of(ExePol pol, FwdIt1 first1, FwdIt1 last1,
                      FwdIt2 first2, FwdIt2 last2, BiPre pre)

```

Returns identical, adjacent elements in a range:

```

FwdIt adjacent_find(FwdIt first, FwdIt last)
FwdIt adjacent_find(ExePol pol, FwdIt first, FwdIt last)

```

```
FwdIt adjacent_find(FwdIt first, FwdIt last, BiPre pre)
FwdIt adjacent_find(ExePol pol, FwdIt first, FwdIt last, BiPre pre)
```

The algorithms require input or forward iterators as arguments and return an iterator on the element when successfully found. If the search is not successful, they return an end iterator.

```
std::find, std::find_if, std::find_if_not, std::find_of, and std::adjacent_fint
// find.cpp
...
#include <algorithm>
...
using namespace std;

bool isVowel(char c){
    string myVowels{"aeiouäöü"};
    set<char> vowels(myVowels.begin(), myVowels.end());
    return (vowels.find(c) != vowels.end());
}

list<char> myCha{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
int cha[] = {'A', 'B', 'C'};

cout << *find(myCha.begin(), myCha.end(), 'g');           // g
cout << *find_if(myCha.begin(), myCha.end(), isVowel);    // a
cout << *find_if_not(myCha.begin(), myCha.end(), isVowel); // b

auto iter= find_first_of(myCha.begin(), myCha.end(), cha, cha + 3);
if (iter == myCha.end()) cout << "None of A, B or C.";      // None of A, B or C.
auto iter2= find_first_of(myCha.begin(), myCha.end(), cha, cha+3,
    [] (char a, char b){ return toupper(a) == toupper(b); });

if (iter2 != myCha.end()) cout << *iter2;                  // a
auto iter3= adjacent_find(myCha.begin(), myCha.end());
if (iter3 == myCha.end()) cout << "No same adjacent chars."; // No same adjacent chars.

auto iter4= adjacent_find(myCha.begin(), myCha.end(),
    [] (char a, char b){ return isVowel(a) == isVowel(b); });
if (iter4 != myCha.end()) cout << *iter4;                  // b
```

Count Elements

You can count elements with the STL with and without a predicate.

Returns the number of elements:

```
Num count(InpIt first, InpIt last, const T& val)
Num count(ExePol pol, FwdIt first, FwdIt last, const T& val)

Num count_if(InpIt first, InpIt last, UnPred pre)
Num count_if(ExePol pol, FwdIt first, FwdIt last, UnPred pre)
```

Count algorithms take input iterators as arguments and return the number of elements matching val or the predicate.

std::count, and std::count_if

```
// count.cpp
...
#include <algorithm>
...
std::string str{"abcdabAAaefaBqeaBCQEaadsfdewAAQAAafbd"};
std::cout << std::count(str.begin(), str.end(), 'a'); // 9
std::cout << std::count_if(str.begin(), str.end(),
    [](char a){ return std::isupper(a); }); // 12
```

Check Conditions on Ranges

The three functions `std::all_of`, `std::any_of` and `std::none_of` answer the question, if all, at least one or no element of a range satisfies the condition. The functions need as arguments input iterators and a unary predicate and return a boolean.

Checks if all elements of the range satisfy the condition:

```
bool all_of(InIt first, InIt last, UnPre pre)
bool all_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

Checks if at least one element of the range satisfies the condition:

```
bool any_of(InIt first, InIt last, UnPre pre)
bool any_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

Checks if no element of the range satisfies the condition:

```
bool none_of(InIt first, InIt last, UnPre pre)
bool none_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

As promised, the example:

std::all_of, std::any_of, and std::none_of

```
// allAnyNone.cpp
...
#include <algorithm>
...
auto even= [](int i){ return i%2; };
std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::cout << std::any_of(myVec.begin(), myVec.end(), even); // true
std::cout << std::all_of(myVec.begin(), myVec.end(), even); // false
std::cout << std::none_of(myVec.begin(), myVec.end(), even); // false
```

Compare Ranges

With `std::equal` you can compare ranges on equality. With `std::lexicographical_compare` and `std::mismatch` you discover which range

is the smaller one.

Checks if both ranges are equal:

```
bool equal(InpIt first1, InpIt last1, InpIt first2)
bool equal(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2)

bool equal(InpIt first1, InpIt last1, InpIt first2, BiPre pred)
bool equal(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2, BiPre pred)

bool equal(InpIt first1, InpIt last1,
           InpIt first2, InpIt last2)
bool equal(ExePol pol, FwdIt first1, FwdIt last1,
           FwdIt first2, FwdIt last2)

bool equal(InpIt first1, InpIt last1,
           InpIt first2, InpIt last2, BiPre pred)
bool equal(ExePol pol, FwdIt first1, FwdIt last1,
           FwdIt first2, FwdIt last2, BiPre pred)
```

Checks if the first range is smaller than the second:

```
bool lexicographical_compare(InpIt first1, InpIt last1,
                             InpIt first2, InpIt last2)
bool lexicographical_compare(ExePol pol, FwdIt first1, FwdIt last1,
                           FwdIt first2, FwdIt last2)

bool lexicographical_compare(InpIt first1, InpIt last1,
                             InpIt first2, InpIt last2, BiPre pred)
bool lexicographical_compare(ExePol pol, FwdIt first1, FwdIt last1,
                           FwdIt first2, FwdIt last2, BiPre pred)
```

Finds the first position at which both ranges are not equal:

```
pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,
                           InpIt first2)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1,
                           FwdIt first2)

pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,
                           InpIt first2, BiPre pred)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last2,
                           FwdIt first2, BiPre pred)

pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,
                           InpIt first2, InpIt last2)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1,
                           FwdIt first2, FwdIt last2)

pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,
                           InpIt first2, InpIt last2, BiPre pred)
pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1,
                           FwdIt first2, FwdIt last2, BiPre pred)
```

The algorithms take input iterators and eventually a binary predicate.

`std::mismatch` returns as its result a pair `pa` of input iterators. `pa.first` holds

an input iterator for the first element that is not equal. `pa.second` holds the corresponding input iterator for the second range. If both ranges are identical, you get two end iterators.

```
std::equal, std::lexicographical_compare, and std::mismatch
// equalLexicographicalMismatch.cpp
...
#include <algorithm>
...
using namespace std;

string str1{"Only For Testing Purpose."};
string str2{"only for testing purpose."};
cout << equal(str1.begin(), str1.end(), str2.begin()); // false
cout << equal(str1.begin(), str1.end(), str2.begin(),
    [](char c1, char c2){ return toupper(c1) == toupper(c2); });
                                                // true

str1= {"Only for testing Purpose."};
str2= {"Only for testing purpose."};
auto pair= mismatch(str1.begin(), str1.end(), str2.begin());
if (pair.first != str1.end()){
    cout << distance(str1.begin(), pair.first)
        << " at (" << *pair.first << "," << *pair.second << ")";
    // 17 at (P,p)
}
auto pair2= mismatch(str1.begin(), str1.end(), str2.begin(),
    [] (char c1, char c2){ return toupper(c1) == toupper(c2); });
if (pair2.first == str1.end()){
    cout << "str1 and str2 are equal"; // str1 and str2 are equal
}
```

Search for Ranges within Ranges

`std::search` searches for a range in another range from the beginning, `std::find_end` from the end. `std::search_n` searches for n consecutive elements in the range.

All algorithms take a forward iterator, can be parametrised by a binary predicate and return an end an iterator for the first range, if the search was not successful.

Searches the second range in the first one and returns the position. Starts at the beginning:

```
FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2)
FwdIt1 search(ExePol pol, FwdIt1 first1, FwdIt1 last1,
    FwdIt2 first2, FwdIt2 last2)

FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
    FwdIt2 first2, FwdIt2 last2, BiPre pre)
FwdIt1 search(ExePol pol, FwdIt1 first1, FwdIt1 last1,
    FwdIt2 first2, FwdIt2 last2, BiPre pre)

FwdIt1 search(FwdIt1 first, FwdIt last1, Search search)
```

Searches the second range in the first one and returns the positions. Starts at the end:

```
FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2 FwdIt2 last2)
FwdIt1 find_end(ExePol pol, FwdIt1 first1, FwdIt1 last1,
                 FwdIt2 first2 FwdIt2 last2)

FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2,
                 BiPre pre)
FwdIt1 find_end(ExePol pol, FwdIt1 first1, FwdIt1 last1,
                 FwdIt2 first2, FwdIt2 last2, BiPre pre)
```

Searches count consecutive values in the first range:

```
FwdIt search_n(FwdIt first, FwdIt last, Size count, const T& value)
FwdIt search_n(ExePol pol, FwdIt first, FwdIt last, Size count, const T& value)

FwdIt search_n(FwdIt first, FwdIt last, Size count, const T& value, BiPre pre)
FwdIt search_n(ExePol pol, FwdIt first,
                 FwdIt last, Size count, const T& value, BiPre pre)
```



The algorithm `search_n` is very special

The algorithm `FwdIt search_n(FwdIt first, FwdIt last, Size count, const T& value, BiPre pre)` is very special. The binary predicate `BiPre` uses as first argument the values of the range and as second argument the value `value`.

```
std::find, std::find_end, and std::search_n
// search.cpp
...
#include <algorithm>
...

using std::search;

std::array<int, 10> arr1{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
std::array<int, 5> arr2{3, 4, -5, 6, 7};

auto fwdIt= search(arr1.begin(), arr1.end(), arr2.begin(), arr2.end());
if (fwdIt == arr1.end()) std::cout << "arr2 not in arr1." // arr2 not in arr1.

auto fwdIt2= search(arr1.begin(), arr1.end(), arr2.begin(), arr2.end(),
                   [] (int a, int b){ return std::abs(a) == std::abs(b); });
if (fwdIt2 != arr1.end()) std::cout << "arr2 at position "
                                         << std::distance(arr1.begin(), fwdIt2) << " in arr1.";
                                         // arr2 at position 3 in arr1.
```

Modifying Algorithms

C++ has many algorithms to modify elements and ranges.

Copy Elements and Ranges

You can copy ranges forward with `std::copy`, backward with `std::copy_backward` and conditionally with `std::copy_if`. If you want to copy `n` elements, you can use `std::copy_n`.

Copies the range:

```
OutIt copy(InpIt first, InpIt last, OutIt result)
FwdIt2 copy(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)
```

Copies `n` elements:

```
OutIt copy_n(InpIt first, Size n, OutIt result)
FwdIt2 copy_n(ExePol pol, FwdIt first, Size n, FwdIt2 result)
```

Copies the elements dependent on the predicate `pre`.

```
OutIt copy_if(InpIt first, InpIt last, OutIt result, UnPre pre)
FwdIt2 copy_if(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result, UnPre pre)
```

Copies the range backward:

```
BiIt copy_backward(BiIt first, BiIt last, BiIt result)
```

The algorithms need input iterators and copy their elements to `result`. They return an end iterator to the destination range.

Copy elements and ranges

```
// copy.cpp
...
#include <algorithm>
...

std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
std::vector<int> myVec2(10);

std::copy_if(myVec.begin(), myVec.end(), myVec2.begin() + 3,
            [] (int a){ return a % 2; });

for (auto v: myVec2) std::cout << v << " ";      // 0 0 0 1 3 5 7 9 00

std::string str{"abcdefghijklmnp"};
std::string str2{"-----"};

std::cout << str2;                                // -----
std::copy_backward(str.begin(), str.end(), str2.end());
std::cout << str2;                                // -----abcdefghijklmnp
std::cout << str;                                 // abcdefghijklmnp
```

```
std::copy_backward(str.begin(), str.begin() + 5, str.end());  
std::cout << str;  
// abcdefghijkabcde
```

Replace Elements and Ranges

You have with `std::replace`, `std::replace_if`, `std::replace_copy` and `std::replace_copy_if` four variations to replace elements in a range. The algorithms differ in two aspects. First, does the algorithm need a predicate? Second, does the algorithm copy the elements in the destination range?

Replaces the old elements in the range with `newValue`, if the old element has the value `old`.

```
void replace(FwdIt first, FwdIt last, const T& old, const T& newValue)  
void replace(ExePol pol, FwdIt first, FwdIt last, const T& old,  
            const T& newValue)
```

Replaces the old elements of the range with `newValue`, if the old element fulfils the predicate `pred`:

```
void replace_if(FwdIt first, FwdIt last, UnPred pred, const T& newValue)  
void replace_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred,  
                const T& newValue)
```

Replaces the old elements in the range with `newValue`, if the old element has the value `old`. Copies the result to `result`:

```
OutIt replace_copy(InpIt first, InpIt last, OutIt result, const T& old,  
                   const T& newValue)  
FwdIt2 replace_copy(ExePol pol, FwdIt first, FwdIt last,  
                    FwdIt2 result, const T& old, const T& newValue)
```

Replaces the old elements of the range with `newValue`, if the old element fulfils the predicate `pred`. Copies the result to `result`:

```
OutIt replace_copy_if(InpIt first, InpIt last, OutIt result, UnPre pred,  
                      const T& newValue)  
FwdIt2 replace_copy_if(ExePol pol, FwdIt first, FwdIt last,  
                       FwdIt2 result, UnPre pred, const T& newValue) \
```

The algorithms in action.

Replace elements and ranges

```
// replace.cpp  
...  
#include <algorithm>  
...
```

```

std::string str{"Only for testing purpose."};
std::replace(str.begin(), str.end(), ' ', '1');
std::cout << str; // Only1for1testing1purpose.

std::replace_if(str.begin(), str.end(), [](char c){ return c == '1'; }, '2');
std::cout << str; // Only2for2testing2purpose.

std::string str2;
std::replace_copy(str.begin(), str.end(), std::back_inserter(str2), '2', '3');
std::cout << str2; // Only3for3testing3purpose.

std::string str3;
std::replace_copy_if(str2.begin(), str2.end(),
std::back_inserter(str3), [](char c){ return c == '3'; }, '4');
std::cout << str3; // Only4for4testing4purpose.

```

Remove Elements and Ranges

The four variations `std::remove`, `std::remove_if`, `std::remove_copy` and `std::remove_copy_if` support two kinds of operations. On one hand, remove elements with and without a predicate from a range. On the other hand, copy the result of your modification to a new range.

Removes the elements from the range, having the value `val`:

```
FwdIt remove(FwdIt first, FwdIt last, const T& val)
FwdIt remove(ExePol pol, FwdIt first, FwdIt last, const T& val)
```

Removes the elements from the range, fulfilling the predicate `pred`:

```
FwdIt remove_if(FwdIt first, FwdIt last, UnPred pred)
FwdIt remove_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred)
```

Removes the elements from the range, having the value `val`. Copies the result to `result`:

```
OutIt remove_copy(InpIt first, InpIt last, OutIt result, const T& val)
FwdIt2 remove_copy(ExePol pol, FwdIt first, FwdIt last,
                  FwdIt2 result, const T& val)
```

Removes the elements from the range, which fulfil the predicate `pred`. Copies the result to `result`.

```
OutIt remove_copy_if(InpIt first, InpIt last, OutIt result, UnPre pred)
FwdIt2 remove_copy_if(ExePol pol, FwdIt first, FwdIt last,
                      FwdIt2 result, UnPre pred)
```

The algorithms need input iterators for the source range and an output iterator for the destination range. They return as a result an end iterator for the

destination range.



Apply the erase-remove idiom

The remove variations don't remove an element from the range. They return the new *logical* end of the range. You have to adjust the size of the container with the erase-remove idiom.

Remove elements and ranges

```
// remove.cpp
...
#include <algorithm>
...
std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

auto newIt= std::remove_if(myVec.begin(), myVec.end(),
    [](int a){ return a%2; });
for (auto v: myVec) std::cout << v << " "; // 0 2 4 6 8 5 6 7 8 9

myVec.erase(newIt, myVec.end());
for (auto v: myVec) std::cout << v << " "; // 0 2 4 6 8

std::string str{"Only for Testing Purpose."};
str.erase( std::remove_if(str.begin(), str.end(),
    [](char c){ return std::isupper(c); }, str.end() ) );
std::cout << str << std::endl; // nly for esting urpose.
```

Fill and Create Ranges

You can fill a range with `std::fill` and `std::fill_n`; you can generate new elements with `std::generate` and `std::generate_n`.

Fills a range with elements:

```
void fill(FwdIt first, FwdIt last, const T& val)
void fill(ExePol pol, FwdIt first, FwdIt last, const T& val)
```

Fills a range with n new elements:

```
OutIt fill_n(OutIt first, Size n, const T& val)
FwdIt fill_n(ExePol pol, FwdIt first, Size n, const T& val)
```

Generates a range with a generator gen:

```
void generate(FwdIt first, FwdIt last, Generator gen)
void generate(ExePol pol, FwdIt first, FwdIt last, Generator gen)
```

Generates n elements of a range with the generator gen:

```
OutIt generate_n(OutIt first, Size n, Generator gen)
FwdIt generate_n(ExePol pol, FwdIt first, Size n, Generator gen)
```

The algorithms expect the value val or a generator gen as an argument. gen has to be a function taking no argument and returning the new value. The return value of the algorithms std::fill_n and std::generate_n is an output iterator, pointing to the last created element.

Fill and create ranges

```
// fillAndCreate.cpp
...
#include <algorithm>
...

int getNext(){
    static int next{0};
    return ++next;
}

std::vector<int> vec(10);
std::fill(vec.begin(), vec.end(), 2011);
for (auto v: vec) std::cout << v << " ";
// 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011

std::generate_n(vec.begin(), 5, getNext);
for (auto v: vec) std::cout << v << " ";
// 1 2 3 4 5 2011 2011 2011 2011 2011
```

Move Ranges

std::move moves the ranges forward; std::move_backward moves the ranges backwards.

Moves the range forward:

```
OutIt move(InIt first, InIt last, OutIt result)
FwdIt2 move(ExePol pol, FwdIt first, FwdIt last, Fwd2It result)
```

Moves the range backward:

```
BiIt move_backward(BiIt first, BiIt last, BiIt result)
```

Both algorithms need a destination iterator result, to which the range is moved. In the case of the std::move algorithm this is an output iterator, and in the case of the std::move_backward algorithm this is a bidirectional iterator. The algorithms return an output or a bidirectional iterator, pointing to the initial position in the destination range.



The source range may be changed

`std::move` and `std::move_backward` apply move semantics. Therefore the source range is valid, but have not necessarily the same elements afterwards.

Move ranges

```
// move.cpp
...
#include <algorithm>
...

std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
std::vector<int> myVec2(myVec.size());
std::move(myVec.begin(), myVec.end(), myVec2.begin());
for (auto v: myVec2) std::cout << v << " "; // 0 1 2 3 4 5 6 7 9 0

std::string str{"abcdefghijklmnp"};
std::string str2{"-----"};
std::move_backward(str.begin(), str.end(), str2.end());
std::cout << str2; // -----abcdefghijklmnp
```

Swap Ranges

`std::swap` and `std::swap_ranges` can swap objects and ranges.

Swaps objects:

```
void swap(T& a, T& b)
```

Swaps ranges:

```
FwdIt swap_ranges(FwdIt1 first1, FwdIt1 last1, FwdIt first2)
FwdIt swap_ranges(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt first2)
```

The returned iterator points to the last swapped element in the destination range.



The ranges must not overlap

Swap algorithms

```
// swap.cpp
...
#include <algorithm>
```

```

...
std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
std::vector<int> myVec2(9);
std::swap(myVec, myVec2);
for (auto v: myVec) std::cout << v << " ";    // 0 0 0 0 0 0 0 0 0
for (auto v: myVec2) std::cout << v << " ";    // 0 1 2 3 4 5 6 7 9

std::string str{"abcdefghijklmnp"};
std::string str2{"-----"};
std::swap_ranges(str.begin(), str.begin() + 5, str2.begin() + 5);
std::cout << str << std::endl;                      // -----fghijklmnp
std::cout << str2 << std::endl;                      // -----abcde-----

```

Transform Ranges

The `std::transform` algorithm applies a unary or binary callable to a range and copies the modified elements to the destination range.

Applies the unary callable `fun` to the elements of the input range and copies the result to `result`:

```

OutIt transform(InpIt first1, InpIt last1, OutIt result, UnFun fun)
FwdIt2 transform(ExePol pol, FwdIt first1, FwdIt last1, FwdIt2 result, UnFun fun)

```

Applies the binary callable `fun` to both input ranges and copies the result to `result`:

```

OutIt transform(InpIt1 first1, InpIt1 last1, InpIt2 first2, OutIt result,
                BiFun fun)
FwdIt3 transform(ExePol pol, FwdIt1 first1, FwdIt1 last1,
                 FwdIt2 first2, FwdIt3 result, BiFun fun)

```

The difference between the two versions is that the first version applies the callable to each element of the range; the second version applies the callable to pairs of both ranges in parallel. The returned iterator points to one position after the last transformed element.

Transform algorithms

```

// transform.cpp
...
#include <algorithm>
...

std::string str{"abcdefghijklmnpqrstuvwxyz"};
std::transform(str.begin(), str.end(), str.begin(),
              [](char c){ return std::toupper(c); });
std::cout << str;                                // ABCDEFGHIJKLMNOPQRSTUVWXYZ

std::vector<std::string> vecStr{"Only", "for", "testing", "purpose", "."};
std::vector<std::string> vecStr2(5, "-");
std::vector<std::string> vecRes;
std::transform(vecStr.begin(), vecStr.end(),

```

```
vecStr2.begin(), std::back_inserter(vecRes),
[]](std::string a, std::string b){ return std::string(b)+a+b; });
for (auto str: vecRes) std::cout << str << " ";
// -Only- -for- -testing- -purpose- -.-
```

Reverse Ranges

`std::reverse` and `std::reverse_copy` invert the order of the elements in their range.

Reverses the order of the elements in the range:

```
void reverse(BiIt first, BiIt last)
void reverse(ExePol pol, BiIt first, BiIt last)
```

Reverses the order of the elements in the range and copies the result to `result`:

```
OutIt reverse_copy(BiIt first, BiIt last, OutIt result)
FwdIt reverse_copy(ExePol pol, BiIt first, BiIt last, FwdIt result)
```

Both algorithms require bidirectional iterators. The returned iterator points to the position of the output range `result` before the elements were copied.

Reverse range algorithms

```
// algorithmen.cpp
...
#include <algorithm>
...

std::string str{"123456789"};
std::reverse(str.begin(), str.begin() + 5);
std::cout << str;           // 543216789
```

Rotate Ranges

`std::rotate` and `std::rotate_copy` rotate their elements.

Rotates the elements in such a way that `middle` becomes the new first element:

```
FwdIt rotate(FwdIt first, FwdIt middle, FwdIt last)
FwdIt rotate(ExePol pol, FwdIt first, FwdIt middle, FwdIt last)
```

Rotates the elements in such a way that `middle` becomes the new first element. Copies the result to `result`:

```
OutIt rotate_copy(FwdIt first, FwdIt middle, FwdIt last, OutIt result)
FwdIt2 rotate_copy(ExePol pol, FwdIt first, FwdIt middle, FwdIt last,
                  FwdIt2 result)
```

Both algorithms need forward iterators. The returned iterator is an end iterator for the copied range.

Rotate algorithms

```
// rotate.cpp
...
#include <algorithm>
...

std::string str{"12345"};
for (auto i= 0; i < str.size(); ++i){
    std::string tmp{str};
    std::rotate(tmp.begin(), tmp.begin() + i, tmp.end());
    std::cout << tmp << " ";
}
                                // 12345 23451 34512 45123 51234
```

Randomly Shuffle Ranges

You can randomly shuffle ranges with `std::random_shuffle` and `std::shuffle`.

Randomly shuffles the elements in a range:

```
void random_shuffle(RanIt first, RanIt last)
```

Randomly shuffles the elements in the range, by using the random number generator `gen`:

```
void random_shuffle(RanIt first, RanIt last, RanNumGen&& gen)
```

Randomly shuffles the elements in a range, using the uniform random number generator `gen`:

```
void shuffle(RanIt first, RanIt last, URNG&& gen)
```

The algorithms need random access iterators. `RanNumGen&& gen` has to be a callable, taking an argument and returning a value within its arguments. `URNG&& gen` has to be a *uniform random number generator*.



Prefer std::shuffle

Use `std::shuffle` instead of `std::random_shuffle`. `std::random_shuffle` has been *deprecated* since C++14 and removed in C++17, because it uses the C function `rand` internally.

Randomly shuffle algorithms

```
// shuffle.cpp
...
#include <algorithm>
...

using std::chrono::system_clock;
using std::default_random_engine;
std::vector<int> vec1{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> vec2(vec1);

std::random_shuffle(vec1.begin(), vec1.end());
for (auto v: vec1) std::cout << v << " ";      // 4 3 7 8 0 5 2 1 6 9

unsigned seed= system_clock::now().time_since_epoch().count();
std::shuffle(vec2.begin(), vec2.end(), default_random_engine(seed));
for (auto v: vec2) std::cout << v << " ";      // 4 0 2 3 9 6 5 1 8 7
```

seed initialises the random number generator.

Remove Duplicates

With the algorithms `std::unique` and `std::unique_copy` you have more opportunities to remove adjacent duplicates. This can be done with and without a binary predicate.

Removes adjacent duplicates:

```
FwdIt unique(FwdIt first, FwdIt last)
FwdIt unique(ExePol pol, FwdIt first, FwdIt last)
```

Removes adjacent duplicates, satisfying the binary predicate:

```
FwdIt unique(FwdIt first, FwdIt last, BiPred pre)
FwdIt unique(ExePol pol, FwdIt first, FwdIt last, BiPred pre)
```

Removes adjacent duplicates and copies the result to result:

```
OutIt unique_copy(InpIt first, InpIt last, OutIt result)
FwdIt2 unique_copy(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)
```

Removes adjacent duplicates, satisfying the binary predicate and copies the result to `result`:

```
OutIt unique_copy(InIt first, InIt last, OutIt result, BiPred pre)
FwdIt2 unique_copy(ExePol pol, FwdIt first, FwdIt last,
                    FwdIt2 result, BiPred pre)
```



The unique algorithms return the new logical end iterator

The unique algorithms return the logical end iterator of the range. The elements have to be removed with the [erase-remove idiom](#).

Remove duplicates algorithms

```
// removeDuplicates.cpp
...
#include <algorithm>
...

std::vector<int> myVec{0, 0, 1, 1, 2, 2, 3, 4, 4, 5,
                      3, 6, 7, 8, 1, 3, 3, 8, 8, 9};

myVec.erase(std::unique(myVec.begin(), myVec.end()), myVec.end());
for (auto v: myVec) std::cout << v << " ";      // 0 1 2 3 4 5 3 6 7 8 1 3 8 9

std::vector<int> myVec2{1, 4, 3, 3, 3, 5, 7, 9, 2, 4, 1, 6, 8,
                        0, 3, 5, 7, 9, 2, 4, 1, 6, 8, 0, 3, 5, 7, 8, 7, 3, 9, 2, 4, 2, 5, 7, 3};
std::vector<int> resVec;
resVec.reserve(myVec2.size());
std::unique_copy(myVec2.begin(), myVec2.end(), std::back_inserter(resVec),
                [] (int a, int b){ return (a%2) == (b%2); } );
for (auto v: myVec2) std::cout << v << " ";
                           // 1 4 3 3 3 5 7 9 2 4 1 6 8 0 3 5 7 8 7 3 9 2 4 2 5 7 3
for (auto v: resVec) std::cout << v << " ";      // 1 4 3 2 1 6 3 8 7 2 5
```

Partition



What is a partition?

A partition of a set is a decomposition of a set in subsets so that each element of the set is precisely in one subset. The subsets are defined in C++ by a unary predicate so that the members of the first subset fulfil the predicate. The remaining elements are in the second subset.

C++ offers a few functions for dealing with partitions. All of them need a unary predicate `pre`. `std::partition` and `std::stable_partition` partition a range and returns the partition point. With `std::partition_point` you can get the partition point of a partition. Afterwards, you can check the partition with `std::is_partitioned` or copy it with `std::partition_copy`.

Checks if the range is partitioned:

```
bool is_partitioned(InIt first, InIt last, UnPre pre)
bool is_partitioned(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

Partitions the range:

```
FwdIt partition(FwdIt first, FwdIt last, UnPre pre)
FwdIt partition(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

Partitions the range ([stable](#)):

```
BiIt stable_partition(FwdIt first, FwdIt last, UnPre pre)
BiIt stable_partition(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

Copies a partition in two ranges:

```
pair<OutIt1, OutIt2> partition_copy(InIt first, InIt last,
                                      OutIt1 result_true, OutIt2 result_false, UnPre pre)
pair<FwdIt1, FwdIt2> partition_copy(ExePol pol, FwdIt1 first, FwdIt1 last,
                                      FwdIt2 result_true, FwdIt3 result_false, UnPre pre) \
```

Returns the partition point:

```
FwdIt partition_point(FwdIt first, FwdIt last, UnPre pre)
```

A `std::stable_partition` guarantees, in contrary to a `std::partition`, that the elements preserve their relative order. The returned iterator `FwdIt` and `BiIt` point to the initial position in the second subset of the partition. The pair `std::pair<OutIt, OutIt>` of the algorithm `std::partition_copy` contains the end iterator of the subsets `result_true` and `result_false`. The behaviour of `std::partition_point` is undefined if the range is not partitioned.

Partition algorithms

```
// partition.cpp
...
#include <algorithm>
...
using namespace std;
```

```

bool isOdd(int i){ return (i%2) == 1; }
vector<int> vec{1, 4, 3, 4, 5, 6, 7, 3, 4, 5, 6, 0, 4,
                8, 4, 6, 6, 5, 8, 8, 3, 9, 3, 7, 6, 4, 8};

auto parPoint= partition(vec.begin(), vec.end(), isOdd);
for (auto v: vec) cout << v << " ";
// 1 7 3 3 5 9 7 3 3 5 5 0 4 8 4 6 6 6 8 8 4 6 4 4 6 4 8

for (auto v= vec.begin(); v != parPoint; ++v) cout << *v << " ";
// 1 7 3 3 5 9 7 3 3 5 5
for (auto v= parPoint; v != vec.end(); ++v) cout << *v << " ";
// 4 8 4 6 6 6 8 8 4 6 4 4 6 4 8

cout << is_partitioned(vec.begin(), vec.end(), isOdd); // true
list<int> le;
list<int> ri;
partition_copy(vec.begin(), vec.end(), back_inserter(le), back_inserter(ri),
               [] (int i) { return i < 5; });
for (auto v: le) cout << v << " "; // 1 3 3 3 3 0 4 4 4 4 4 4
for (auto v: ri) cout << v << " "; // 7 5 9 7 5 5 8 6 6 6 8 8 6 6 8

```

Sort

You can sort a range with `std::sort` or `std::stable_sort` or `sort until a position` with `std::partial_sort`. In addition `std::partial_sort_copy` copies the partially sorted range. With `std::nth_element` you can assign an element the *sorted* position in the range. You can check with `std::is_sorted` if a range is sorted. If you want to know until which position a range is sorted, use `std::is_sorted_until`.

Per default the predefined function object `std::less` is used as a sorting criterion. However, you can use your sorting criterion. This has to obey the [strict weak ordering](#).

Sorts the elements in the range:

```

void sort(RaIt first, RaIt last)
void sort(ExePol pol, RaIt first, RaIt last)

void sort(RaIt first, RaIt last, BiPre pre)
void sort(ExePol pol, RaIt first, RaIt last, BiPre pre)

```

Sorts the elements in the range ([stable](#)):

```

void stable_sort(RaIt first, RaIt last)
void stable_sort(ExePol pol, RaIt first, RaIt last)

void stable_sort(RaIt first, RaIt last, BiPre pre)
void stable_sort(ExePol pol, RaIt first, RaIt last, BiPre pre)

```

Sorts partially the elements in the range until `middle`:

```

void partial_sort(RaIt first, RaIt middle, RaIt last)
void partial_sort(ExePol pol, RaIt first, RaIt middle, RaIt last)

void partial_sort(RaIt first, RaIt middle, RaIt last, BiPre pre)
void partial_sort(ExePol pol, RaIt first, RaIt middle, RaIt last, BiPre pre)

```

Sorts partially the elements in the range and copies them in the destination ranges `result_first` and `result_last`:

```

RaIt partial_sort_copy(Init first, Init last,
                      RaIt result_first, RaIt result_last)
RaIt partial_sort_copy(ExePol pol, FwdIt first, FwdIt last,
                      RaIt result_first, RaIt result_last)

RaIt partial_sort_copy(Init first, Init last,
                      RaIt result_first, RaIt result_last, BiPre pre)
RaIt partial_sort_copy(ExePol pol, FwdIt first, FwdIt last,
                      RaIt result_first, RaIt result_last, BiPre pre)

```

Checks if a range is sorted:

```

bool is_sorted(FwdIt first, FwdIt last)
bool is_sorted(ExePol pol, FwdIt first, FwdIt last)

bool is_sorted(FwdIt first, FwdIt last, BiPre pre)
bool is_sorted(ExePol pol, FwdIt first, FwdIt last, BiPre pre)

```

Returns the position to the first element that doesn't satisfy the sorting criterion:

```

FwdIt is_sorted_until(FwdIt first, FwdIt last)
FwdIt is_sorted_until(ExePol pol, FwdIt first, FwdIt last)

FwdIt is_sorted_until(FwdIt first, FwdIt last, BiPre pre)
FwdIt is_sorted_until(ExePol pol, FwdIt first, FwdIt last, BiPre pre)

```

Reorders the range, so that the n-th element has the right (sorted) position:

```

void nth_element(RaIt first, RaIt nth, RaIt last)
void nth_element(ExePol pol, RaIt first, RaIt nth, RaIt last)

void nth_element(RaIt first, RaIt nth, RaIt last, BiPre pre)
void nth_element(ExePol pol, RaIt first, RaIt nth, RaIt last, BiPre pre)

```

Here is a code snippet.

Sort algorithms

```

// sort.cpp
...
#include <algorithm>
...

std::string str{"RUDAjDkaACsdfjwldXmnEiVSEZTiepfg0Ikue"};
std::cout << std::is_sorted(str.begin(), str.end());           // false

std::partial_sort(str.begin(), str.begin() + 30, str.end());

```

```

std::cout << str;           // AACDEEIORSTUVXZaddddeeffgijjkwspluk

auto sortUntil= std::is_sorted_until(str.begin(), str.end());
std::cout << *sortUntil;      // s
for (auto charIt= str.begin(); charIt != sortUntil; ++charIt)
    std::cout << *charIt;       // AACDEEIORSTUVXZaddddeeffgijjkw

std::vector<int> vec{1, 0, 4, 3, 5};
auto vecIt= vec.begin();
while(vecIt != vec.end()){
    std::nth_element(vec.begin(), vecIt++, vec.end());
    std::cout << std::distance(vec.begin(), vecIt) << "-th ";
    for (auto v: vec) std::cout << v << "/";
}
// 1-th 01435/2-th 01435/3-th 10345/4-th 30145/5-th 10345

```

Binary Search

The binary search algorithms use the fact that the ranges are already sorted. To search for an element, use `std::binary_search`. With `std::lower_bound` you get an iterator for the first element, being no smaller than the given value. With `std::upper_bound` you get an iterator back for the first element, which is bigger than the given value. `std::equal_range` combines both algorithms.

If the container has n elements, you need on average $\log_2(n)$ comparisons for the search. The binary search requires that you use the same comparison criterion that you used for sorting the container. Per default the comparison criterion is `std::less`, but you can adjust it. Your sorting criterion has to obey the [strict weak ordering](#). If not, the program is undefined.

If you have an unordered associative container, the methods of the [unordered associative container](#) are in general faster.

Searches the element `val` in the range:

```

bool binary_search(FwdIt first, FwdIt last, const T& val)
bool binary_search(FwdIt first, FwdIt last, const T& val, BiPre pre)

```

Returns the position of the first element of the range, being not smaller than `val`:

```

FwdIt lower_bound(FwdIt first, FwdIt last, const T& val)
FwdIt lower_bound(FwdIt first, FwdIt last, const T& val, BiPre pre)

```

Returns the position of the first element of the range, being bigger than `val`:

```

FwdIt upper_bound(FwdIt first, FwdIt last, const T& val)
FwdIt upper_bound(FwdIt first, FwdIt last, const T& val, BiPre pre)

```

Returns the pair `std::lower_bound` and `std::upper_bound` for the element `val`:

```
pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T& val)
pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T& val, BiPre pre)
```

Finally, the code snippet.

Binary search algorithms

```
// binarySearch.cpp
...
#include <algorithm>
...
using namespace std;

bool isLessAbs(int a, int b){
    return abs(a) < abs(b);
}
vector<int> vec{-3, 0, -3, 2, -3, 5, -3, 7, -0, 6, -3, 5,
                 -6, 8, 9, 0, 8, 7, -7, 8, 9, -6, 3, -3, 2};

sort(vec.begin(), vec.end(), isLessAbs);
for (auto v: vec) cout << v << " ";
    // 0 0 2 2 -3 -3 -3 3 -3 5 5 -6 -6 6 7 -7 7 8 8 8 9 9
cout << binary_search(vec.begin(), vec.end(), -5, isLessAbs); // true
cout << binary_search(vec.begin(), vec.end(), 5, isLessAbs); // true

auto pair= equal_range(vec.begin(), vec.end(), 3, isLessAbs);
cout << distance(vec.begin(), pair.first); // 5
cout << distance(vec.begin(), pair.second)-1; // 11

for (auto threeIt= pair.first;threeIt != pair.second; ++threeIt)
    cout << *threeIt << " "; // -3 -3 -3 -3 -3 3 -3
```

Merge Operations

Merge operations empower you to merge sorted ranges in a new sorted range. The algorithm requires that the ranges and the algorithm use the same sorting criterion. If not, the program is undefined. Per default the predefined sorting criterion `std::less` is used. If you use your sorting criterion, it has to obey the [strict weak ordering](#). If not, the program is undefined.

You can merge two sorted ranges with `std::inplace_merge` and `std::merge`. You can check with `std::includes` if one sorted range is in another sorted range. You can merge with `std::set_difference`, `std::set_intersection`, `std::set_symmetric_difference` and `std::set_union` two sorted ranges in a new sorted range.

Merges in place two sorted sub-ranges `[first, mid)` and `[mid, last)`:

```
void inplace_merge(BiIt first, BiIt mid, BiIt last)
void inplace_merge(ExePol pol, BiIt first, BiIt mid, BiIt last)
```

```
void inplace_merge(BiIt first, BiIt mid, BiIt last, BiPre pre)
void inplace_merge(ExePol pol, BiIt first, BiIt mid, BiIt last, BiPre pre)
```

Merges two sorted ranges and copies the result to result:

```
OutIt merge(InpIt first1, InpIt last1, InpIt first2, InpIt last2, OutIt result)
FwdIt3 merge(ExePol pol, FwdIt1 first1, FwdIt1 last1,
             FwdIt2 first2, FwdIt2 last2, FwdIt3 result)

OutIt merge(InpIt first1, InpIt last1, InpIt first2, InpIt last2, OutIt result,
            BiPre pre)
FwdIt3 merge(ExePol pol, FwdIt1 first1, FwdIt1 last1,
             FwdIt2 first2, FwdIt2 last2, FwdIt3 result, BiPre pre)
```

Checks if all elements of the second range are in the first range:

```
bool includes(InpIt first1, InpIt last1, InpIt1 first2, InpIt1 last2)
bool includes(ExePol pol, FwdIt first1, FwdIt last1, FwdIt1 first2, FwdIt1 last2)

bool includes(InpIt first1, InpIt last1, InpIt first2, InpIt last2, BinPre pre)
bool includes(ExePol pol, FwdIt first1, FwdIt last1,
              FwdIt1 first2, FwdIt1 last2, BinPre pre)
```

Copies these elements of the first range to result, being not in the second range:

```
OutIt set_difference(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,
                      OutIt result)
FwdIt2 set_difference(ExePol pol, FwdIt first1, FwdIt last1,
                      FwdIt1 first2, FwdIt1 last2, FwdIt2 result)

OutIt set_difference(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,
                      OutIt result, BiPre pre)
FwdIt2 set_difference(ExePol pol, FwdIt first1, FwdIt last1,
                      FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)
```

Determines the intersection of the first with the second range and copies the result to result:

```
OutIt set_intersection(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,
                       OutIt result)
FwdIt2 set_intersection(ExePol pol, FwdIt first1, FwdIt last1,
                       FwdIt1 first2, FwdIt1 last2, FwdIt2 result)

OutIt set_intersection(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,
                       OutIt result, BiPre pre)
FwdIt2 set_intersection(ExePol pol, FwdIt first1, FwdIt last1,
                       FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)
```

Determines the symmetric difference of the first with the second range and copies the result to result:

```
OutIt set_symmetric_difference(InpIt first1, InpIt last1,
                               InpIt1 first2, InpIt2 last2, OutIt result)
```

```

FwdIt2 set_symmetric_difference(ExePol pol, FwdIt first1, FwdIt last1,
                                FwdIt1 first2, FwdIt1 last2, FwdIt2 result)

OutIt set_symmetric_difference(InpIt first1, InpIt last1,
                               InpIt1 first2, InpIt2 last2, OutIt result,
                               BiPre pre)
FwdIt2 set_symmetric_difference(ExePol pol, FwdIt first1, FwdIt last1,
                                FwdIt1 first2, FwdIt1 last2, FwdIt2 result,
                                BiPre pre)

```

Determines the union of the first with the second range and copies the result to result:

```

OutIt set_union(InpIt first1, InpIt last1,
                 InpIt1 first2, InpIt2 last2, OutIt result)
FwdIt2 set_union(ExePol pol, FwdIt first1, FwdIt last1,
                  FwdIt1 first2, FwdIt1 last2, FwdIt2 result)

OutIt set_union(InpIt first1, InpIt last1,
                 InpIt1 first2, InpIt2 last2, OutIt result, BiPre pre)
FwdIt2 set_union(ExePol pol, FwdIt first1, FwdIt last1,
                  FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)

```

The returned iterator is an end iterator for the destination range. The destination range of std::set_difference has all the elements in the first, but not the second range. On the contrary, the destination range of std::symmetric_difference has only the elements that are elements of one range, but not both. std::union determines the union of both sorted ranges.

Merge algorithms

```

// merge.cpp
...
#include <algorithm>
...

std::vector<int> vec1{1, 1, 4, 3, 5, 8, 6, 7, 9, 2};
std::vector<int> vec2{1, 2, 3};

std::sort(vec1.begin(), vec1.end());
std::vector<int> vec(vec1);

vec1.reserve(vec1.size() + vec2.size());
vec1.insert(vec1.end(), vec2.begin(), vec2.end());
for (auto v: vec1) std::cout << v << " ";           // 1 1 2 3 4 5 6 7 8 9 1 2 3

std::inplace_merge(vec1.begin(), vec1.end()-vec2.size(), vec1.end());
for (auto v: vec1) std::cout << v << " ";           // 1 1 1 2 2 3 3 4 5 6 7 8 9

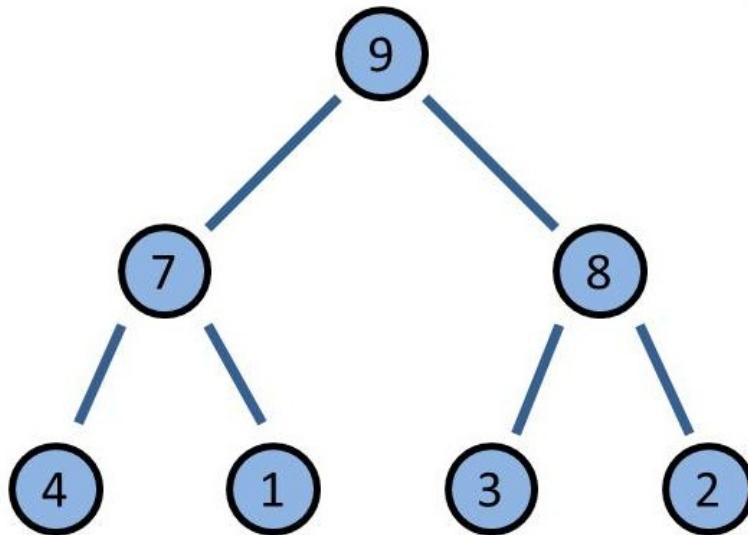
vec2.push_back(10);
for (auto v: vec) std::cout << v << " ";           // 1 1 2 3 4 5 6 7 8 9
for (auto v: vec2) std::cout << v << " ";           // 1 2 3 10

std::vector<int> res;
std::set_symmetric_difference(vec.begin(), vec.end(), vec2.begin(), vec2.end(),
                             std::back_inserter(res));
for (auto v : res) std::cout << v << " ";           // 1 4 5 6 7 8 9 10

```

```
res= {};
std::set_union(vec.begin(), vec.end(), vec2.begin(), vec2.end(),
std::back_inserter(res));
for (auto v : res) std::cout << v << " ";
// 1 1 2 3 4 5 6 7 8 9 10
```

Heaps



What is a heap?

A heap is a binary search tree in which parent elements are always bigger than its child elements. Heap trees are optimised for the efficient sorting of elements.

You can create with `std::make_heap` a heap. You can push with `std::push_heap` new elements on the heap. On the contrary, you can pop the largest element with `std::pop_heap` from the heap. Both operations respect the heap characteristics. `std::push_heap` moves the last element of the range on the heap; `std::pop_heap` moves the biggest element of the heap to the last position in the range. You can check with `std::is_heap` if a range is a heap. You can determine with `std::is_heap_until` until which position the range is a heap. `std::sort_heap` sorts the heap.

The heap algorithms require that the ranges and the algorithm use the same sorting criterion. If not, the program is undefined. Per default the predefined

sorting criterion `std::less` is used. If you use your sorting criterion, it has to obey the [strict weak ordering](#). If not, the program is undefined.

Creates a heap from the range:

```
void make_heap(RaIt first, RaIt last)
void make_heap(RaIt first, RaIt last, BiPre pre)
```

Checks if the range is a heap:

```
bool is_heap(RaIt first, RaIt last)
bool is_heap(ExePol pol, RaIt first, RaIt last)

bool is_heap(RaIt first, RaIt last, BiPre pre)
bool is_heap(ExePol pol, RaIt first, RaIt last, BiPre pre)
```

Determines until which position the range is a heap:

```
RaIt is_heap_until(RaIt first, RaIt last)
RaIt is_heap_until(ExePol pol, RaIt first, RaIt last)

RaIt is_heap_until(RaIt first, RaIt last, BiPre pre)
RaIt is_heap_until(ExePol pol, RaIt first, RaIt last, BiPre pre)
```

Sorts the heap:

```
void sort_heap(RaIt first, RaIt last)
void sort_heap(RaIt first, RaIt last, BiPre pre)
```

Pushes the last element of the range onto the heap. `[first, last-1]` has to be a heap.

```
void push_heap(RaIt first, RaIt last)
void push_heap(RaIt first, RaIt last, BiPre pre)
```

Removes the biggest element from the heap and puts it to the end of the range:

```
void pop_heap(RaIt first, RaIt last)
void pop_heap(RaIt first, RaIt last, BiPre pre)
```

With `std::pop_heap` you can remove the biggest element from the heap. Afterwards, the biggest element is the last element of the range. To remove it from the heap `h`, use `h.pop_back`.

Heap algorithms

```
// heap.cpp
...
#include <algorithm>
...
```

```

std::vector<int> vec{4, 3, 2, 1, 5, 6, 7, 9, 10};
std::make_heap(vec.begin(), vec.end());
for (auto v: vec) std::cout << v << " ";
std::cout << std::is_heap(vec.begin(), vec.end()); // 10 9 7 4 5 6 2 3 1 // true

vec.push_back(100);
std::cout << std::is_heap(vec.begin(), vec.end()); // false
std::cout << *std::is_heap_until(vec.begin(), vec.end()); // 100
for (auto v: vec) std::cout << v << " ";
// 10 9 7 4 5 6 2 3 1 100

std::push_heap(vec.begin(), vec.end());
std::cout << std::is_heap(vec.begin(), vec.end()); // true
for (auto v: vec) std::cout << v << " ";
// 100 10 7 4 9 6 2 3 1 5

std::pop_heap(vec.begin(), vec.end());
for (auto v: vec) std::cout << v << " ";
std::cout << *std::is_heap_until(vec.begin(), vec.end()); // 100

vec.resize(vec.size()-1);
std::cout << std::is_heap(vec.begin(), vec.end()); // true
std::cout << vec.front() << std::endl; // 10

```

Min and Max

You can determine the minimum, the maximum and the minimum and maximum pair of a range with the algorithms `std::min_element`, `std::max_element` and `std::minmax_element`. Each algorithm can be configured with a binary predicate.

Returns the minimum element of the range:

```

constexpr FwdIt min_element(FwdIt first, FwdIt last)
FwdIt min_element(ExePol pol, FwdIt first, FwdIt last)

constexpr FwdIt min_element(FwdIt first, FwdIt last, BinPre pre)
FwdIt min_element(ExePol pol, FwdIt first, FwdIt last, BinPre pre)

```

Returns the maximum element of the range:

```

constexpr FwdIt max_element(FwdIt first, FwdIt last)
FwdIt max_element(ExePol pol, FwdIt first, FwdIt last)

constexpr FwdIt max_element(FwdIt first, FwdIt last, BinPre pre)
FwdIt max_element(ExePol pol, FwdIt first, FwdIt last, BinPre pre)

```

Returns the pair `std::min_element` and `std::max_element` of the range:

```

constexpr pair<FwdIt, FwdIt> minmax_element(FwdIt first, FwdIt last)
pair<FwdIt, FwdIt> minmax_element(ExePol pol, FwdIt first, FwdIt last)

constexpr pair<FwdIt, FwdIt> minmax_element(FwdIt first, FwdIt last, BinPre pre)
pair<FwdIt, FwdIt> minmax_element(ExePol pol, FwdIt first, FwdIt last,
BinPre pre)

```

If the range has more than one minimum or maximum element, the first one is returned.

Minimum and maximum algorithms

```
// minMax.cpp
...
#include <algorithm>
...

int.toInt(const std::string& s){
    std::stringstream buff;
    buff.str("");
    buff << s;
    int value;
    buff >> value;
    return value;
}

std::vector<std::string> myStrings{"94", "5", "39", "-4", "-49", "1001", "-77",
                                    "23", "0", "84", "59", "96", "6", "-94"};
auto str= std::minmax_element(myStrings.begin(), myStrings.end());
std::cout << *str.first << ":" << *str.second;           // -4:96

auto asInt= std::minmax_element(myStrings.begin(), myStrings.end(),
                                [] (std::string a, std::string b){ return toInt(a) < toInt(b); });
std::cout << *asInt.first << ":" << *asInt.second;        // -94:1001
```

Permutations

`std::prev_permutation` and `std::next_permutation` return the previous smaller or next bigger permutation of the newly ordered range. If a smaller or bigger permutation is not available, the algorithms return `false`. Both algorithms need bidirectional iterators. Per default the predefined sorting criterion `std::less` is used. If you use your sorting criterion, it has to obey the [strict weak ordering](#). If not, the program is undefined.

Applies the previous permutation to the range:

```
bool prev_permutation(BiIt first, BiIt last)
bool prev_permutation(BiIt first, BiIt last, BiPred pre)
```

Applies the next permutation to the range:

```
bool next_permutation(BiIt first, BiIt last)
bool next_permutation(BiIt first, BiIt last, BiPred pre)
```

You can easily generate with both algorithms all permutations of the range.

Permutation algorithms

```
// permutation.cpp
...
```

```

#include <algorithm>
...
std::vector<int> myInts{1, 2, 3};
do{
    for (auto i: myInts) std::cout << i;
    std::cout << " ";
} while(std::next_permutation(myInts.begin(), myInts.end()));
// 123 132 213 231 312 321

std::reverse(myInts.begin(), myInts.end());
do{
    for (auto i: myInts) std::cout << i;
    std::cout << " ";
} while(std::prev_permutation(myInts.begin(), myInts.end()));
// 321 312 231 213 132 123

```

Numeric

The numeric algorithms `std::accumulate`, `std::adjacent_difference`, `std::partial_sum`, `std::inner_product` and `std::iota` and the six additional C++17 algorithms `std::exclusive_scan`, `std::inclusive_scan`, `std::transform_exclusive_scan`, `std::transform_inclusive_scan`, `std::reduce`, and `std::transform_reduce` are special. All of them are defined in the header `<numeric>`. They are widely applicable, because they can be configured with a callable.

Accumulates the elements of the range. `init` is the start value:

```

T accumulate(InpIt first, InpIt last, T init)
T accumulate(InpIt first, InpIt last, T init, BiFun fun)

```

Calculates the difference between adjacent elements of the range and stores the result in `result`:

```

OutIt adjacent_difference(InpIt first, InpIt last, OutIt result)
FwdIt2 adjacent_difference(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)

OutIt adjacent_difference(InpIt first, InpIt last, OutIt result, BiFun fun)
FwdIt2 adjacent_difference(ExePol pol, FwdIt first, FwdIt last,
                           FwdIt2 result, BiFun fun)

```

Calculates the partial sum of the range:

```

OutIt partial_sum(InpIt first, InpIt last, OutIt result)
OutIt partial_sum(InpIt first, InpIt last, OutIt result, BiFun fun)

```

Calculates the inner product (scalar product) of the two ranges and returns the result:

```

T inner_product(InpIt first1, InpIt last1, OutIt first2, T init)
T inner_product(InpIt first1, InpIt last1, OutIt first2, T init,
               BiFun fun1, BiFun fun2)

```

Assigns each element of the range a by 1 sequentially increasing value. The start value is val:

```
void iota(FwdIt first, FwdIt last, T val)
```

The algorithms are not so easy to get.

`std::accumulate` without callable uses the following strategy:

```

result = init;
result += *(first+0);
result += *(first+1);
...

```

`std::adjacent_difference` without callable uses the following strategy:

```

*(result) = *first;
*(result+1) = *(first+1) - *(first);
*(result+2) = *(first+2) - *(first+1);
...

```

`std::partial_sum` without callable uses the following strategy:

```

*(result) = *first;
*(result+1) = *first + *(first+1);
*(result+2) = *first + *(first+1) + *(first+2)
...

```

The challenging algorithm variation `inner_product` (`InpIt`, `InpIt`, `OutIt`, `T`, `BiFun fun1`, `BiFun fun2`) with two binary callables uses the following strategy: The second callable `fun2` is applied to each pair of the ranges to generate the temporary destination range `tmp`, and the first callable is applied to each element of the destination range `tmp` for accumulating them and therefore generating the final result.

Numeric algorithms

```

// numeric.cpp
...
#include <numeric>
...

std::array<int, 9> arr{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::cout << std::accumulate(arr.begin(), arr.end(), 0); // 45
std::cout << std::accumulate(arr.begin(), arr.end(), 1,
                           [] (int a, int b){ return a*b; } ); // 362880

```

```

std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> myVec;
std::adjacent_difference(vec.begin(), vec.end(),
    std::back_inserter(myVec), [](int a, int b){ return a*b; });
for (auto v: myVec) std::cout << v << " "; // 1 2 6 12 20 30 42 56 72
std::cout << std::inner_product(vec.begin(), vec.end(), arr.begin(), 0); // 285

myVec= {};
std::partial_sum(vec.begin(), vec.end(), std::back_inserter(myVec));
for (auto v: myVec) std::cout << v << " "; // 1 3 6 10 15 21 28 36 45

std::vector<int> myLongVec(10);
std::iota(myLongVec.begin(), myLongVec.end(), 2000);
for (auto v: myLongVec) std::cout << v << " ";
// 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009

```

New Algorithms with C++17

The six new algorithms that are typically used for parallel execution are also known under the name [prefix sum](#). If the given binary callables are not associative and commutative, the behaviour of the algorithms is undefined.

`reduce`: reduces the elements of the range. `init` is the start value.

- Behaves the same as [`std::accumulate`](#) but the range may be rearranged.

```

ValType reduce(InpIt first, InpIt last)
ValType reduce(ExePol pol, InpIt first, InpIt last)

T reduce(InpIt first, InpIt last, T init)
T reduce(ExePol pol, InpIt first, InpIt last, T init)

T reduce(InpIt first, InpIt last, T init, BiFun fun)
T reduce(ExePol pol, InpIt first, InpIt last, T init, BiFun fun)

```

`transform_reduce`: transforms and reduces the elements of one or two ranges. `init` is the start value.

- Behaves similar to [`std::inner_product`](#) but the range may be rearranged.
- If applied to two ranges
 - if not provided, multiplication is used for transforming the ranges into one range and the addition is used to reduce the intermediate range into the result
 - if provided, `fun1` is used for the transforming step and `fun2` is used for the reducing step
- If applied to a single range
 - `fun2` is used for transforming the given range

```

T transform_reduce(InpIt first, InpIt last, InpIt first2, T init)
T transform_reduce(InpIt first, InpIt last,
                  InpIt first2, T init, BiFun fun1, BiFun fun2)

T transform_reduce(FwdIt first, FwdIt last, FwdIt first2, T init)
T transform_reduce(ExePol pol, FwdIt first, FwdIt last,
                  FwdIt first2, T init, BiFun fun1, BiFun fun2)

T transform_reduce(InpIt first, InpIt last, T init, BiFun fun1, UnFun fun2)
T transform_reduce(ExePol pol, FwdIt first, FwdIt last,
                  T init, BiFun fun1, UnFun fun2)

```



MapReduce in C++17

The [Haskell](#) function `map` is called `std::transform` in C++. When you substitute `transform` with `map` in the name `std::transform_reduce`, you will get `std::map_reduce`. [MapReduce](#) is the well-known parallel framework that first maps each value to a new value, then reduces in the second phase all values to the result.

The algorithm is directly applicable in C++17. In the following example, in the map phase, each word is mapped to its length, and the lengths of all words are then reduced to their sum during the reduce phase. The result is the sum of the length of all words.

```

std::vector<std::string> str{"Only", "for", "testing", "purpose"};

std::size_t result= std::transform_reduce(std::execution::par,
                                         str.begin(), str.end(),
                                         0, [](std::size_t a, std::size_t b){ return a + b; },
                                         [](std::string s){ return s.length(); });

std::cout << result << std::endl;      //    21

```

`exclusive_scan`: computes the exclusive prefix sum using a binary operation

- Behaves similar to [std::reduce](#), but provides a range of all prefix sums
- excludes the last element in each iteration

```

OutIt exclusive_scan(InpIt first, InpIt last, OutIt first, T init)
FwdIt2 exclusive_scan(ExePol pol, FwdIt first, FwdIt last,
                      FwdIt2 first2, T init)

OutIt exclusive_scan(InpIt first, InpIt last, OutIt first, T init, BiFun fun)
FwdIt2 exclusive_scan(ExePol pol, FwdIt first, FwdIt last,
                      FwdIt2 first2, T init, BiFun fun)

```

`inclusive_scan`: computes the inclusive prefix sum using a binary operation

- Behaves similar to [std::reduce](#), but provides a range of all prefix sums
- includes the last element in each iteration

```
OutIt inclusive_scan(InpIt first, InpIt last, OutIt first2)
FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last, FwdIt2 first2)

OutIt inclusive_scan(InpIt first, InpIt last, OutIt first, BiFun fun)
FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last,
                      FwdIt2 first2, BiFun fun)

OutIt inclusive_scan(InpIt first, InpIt last, OutIt firs2t, BiFun fun, T init)
FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last,
                      FwdIt2 first2, BiFun fun, T init)
```

transform_exclusive_scan: first transforms each element and then computes the exclusive prefix sums

```
OutIt transform_exclusive_scan(InpIt first, InpIt last, OutIt first2, T init,
                               BiFun fun, UnFun fun2)

FwdIt2 transform_exclusive_scan(ExePol pol, FwdIt first, FwdIt last,
                                 FwdIt2 first2, T init,
                                 BiFun fun, UnFun fun2)
```

transform_inclusive_scan: first transforms each element of the input range and then computes the inclusive prefix sums

```
OutIt transform_inclusive_scan(InpIt first, InpIt last, OutIt first2,
                               BiFun fun, UnFun fun2)

FwdIt2 transform_inclusive_scan(ExePol pol, FwdIt first, FwdIt last,
                                 FwdIt2 first2,
                                 BiFun fun, UnFun fun2)

OutIt transform_inclusive_scan(InpIt first, InpIt last, OutIt first2,
                               BiFun fun, UnFun fun2,
                               T init)

FwdIt2 transform_inclusive_scan(ExePol pol, FwdIt first, FwdIt last,
                                 FwdIt2 first2,
                                 BiFun fun, UnFun fun2,
                                 T init)
```

The following example illustrates the usage of the six algorithms using the parallel [execution policy](#).

The new algorithms

```
// newAlgorithms.cpp
...
#include <numeric>
...

std::vector<int> resVec{1, 2, 3, 4, 5, 6, 7, 8};
std::exclusive_scan(std::execution::par,
```

```

        resVec.begin(), resVec.end(), resVec.begin(), 1,
        [](int fir, int sec){ return fir * sec; });

for (auto v: resVec) std::cout << v << " "; // 1 1 2 6 24 120 720 5040

std::vector<int> resVec2{1, 2, 3, 4, 5, 6, 7, 8};

std::inclusive_scan(std::execution::par,
                    resVec2.begin(), resVec2.end(), resVec2.begin(),
                    [](int fir, int sec){ return fir * sec; }, 1);

for (auto v: resVec2) std::cout << v << " "; // 1 2 6 24 120 720 5040 40320

std::vector<int> resVec3{1, 2, 3, 4, 5, 6, 7, 8};
std::vector<int> resVec4(resVec3.size());
std::transform_exclusive_scan(std::execution::par,
                             resVec3.begin(), resVec3.end(),
                             resVec4.begin(), 0,
                             [](int fir, int sec){ return fir + sec; },
                             [](int arg){ return arg *= arg; });

for (auto v: resVec4) std::cout << v << " "; // 0 1 5 14 30 55 91 140

std::vector<std::string> strVec{"Only", "for", "testing", "purpose"};
std::vector<int> resVec5(strVec.size());

std::transform_inclusive_scan(std::execution::par,
                            strVec.begin(), strVec.end(),
                            resVec5.begin(),
                            [](auto fir, auto sec){ return fir + sec; },
                            [](auto s){ return s.length(); }, 0);

for (auto v: resVec5) std::cout << v << " "; // 4 7 14 21

std::vector<std::string> strVec2{"Only", "for", "testing", "purpose"};

std::string res = std::reduce(std::execution::par,
                           strVec2.begin() + 1, strVec2.end(), strVec2[0],
                           [](auto fir, auto sec){ return fir + ":" + sec; });

std::cout << res;      // Only:for:testing:purpose

std::size_t res7 = std::transform_reduce(std::execution::par,
                                       strVec2.begin(), strVec2.end(), 0,
                                       [](std::size_t a, std::size_t b){ return a + b; },
                                       [](std::string s){ return s.length(); });

std::cout << res7;    // 21

```

11. Ranges

The algorithms of the ranges library are lazy, operate directly on the container, and can be composed.

Composing of ranges

```
// rangesFilterTransform.cpp
...
#include <ranges>

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 can pass which are even (std::views::filter([](int n){ return n % 2 == 0; })). Afterwards, each remaining number is mapped to its double (std::views::transform([](int n){ return n * 2; })). T

Range

Ranges and views are [concepts](#). C++20 supports various kinds of ranges:

std::range

A range is a group of items that you can iterator over. It provides a begin iterator and an end sentinel.

There are further refinements of std::range:

Refinements of std::range

Range (namespace std omitted)

Description

ranges::input_range

Specifies a range whose iterator type satisfies [input iterator](#)

<code>ranges::output_range</code>	Specifies a range whose iterator type satisfies <u>output iterator</u>
<code>ranges::forward_range</code>	Specifies a range whose iterator type satisfies <u>forward iterator</u>
<code>ranges::bidirectional_range</code>	Specifies a range whose iterator type satisfies <u>bidirectional iterator</u>
<code>ranges::random_access_range</code>	Specifies a range whose iterator type satisfies <u>random access iterator</u>
<code>ranges::contiguous_range</code>	Specifies a range whose iterator type satisfies <u>contiguous iterator</u>

A `random_access_iterator` provides random-access on its elements and is implicit a `bidirectional_iterator`; a `bidirectional_iterator` enables iterating in both directions and is implicit a `forward_iterator`; a `forward_iterator` iteration in one direction. The interface of a `contiguous_iterator` is the same such as the interface of a `bidirectional_iterator`. The `contiguous_iterator` guarantees that the iterator refers contiguous storage.

View

A View is something that you apply on a range and performs some operation. A view does not own data and it's time to copy, to move, or to assigne is constant.

```
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.

The ranges library has a rich set of views.

Views in C++20

View	Description
<code>std::views::all_t</code>	Takes all elements.

<code>std::views::all</code>	
<code>std::ranges::ref_view</code>	Takes all elements of another range.
<code>std::ranges::filter_view</code> <code>std::views::filter</code>	Takes the elements which satisfy the predicate.
<code>std::ranges::transform_view</code> <code>std::views::transform</code>	Transforms each element.
<code>std::ranges::take_view</code> <code>std::views::take</code>	Takes the first n elements of another view.
<code>std::ranges::take_while_view</code> <code>std::views::take_while</code>	Takes the elements of another view as long as the predicate returns true.
<code>std::ranges::drop_view</code> <code>std::views::drop</code>	Skips the first n elements of another view.
<code>std::ranges::drop_while_view</code> <code>std::views::drop_while</code>	Skips the initial elements of another view until the predicate returns false.
<code>std::ranges::join_view</code> <code>std::views::join</code>	Joins a view of ranges.
<code>std::ranges::split_view</code> <code>std::views::split</code>	Splits a view by using a delimiter.
<code>std::ranges::common_view</code> <code>std::views::common</code>	Converts a view into a <code>std::ranges::common_range</code> .
<code>std::ranges::reverse_view</code> <code>std::views::reverse</code>	Iterates in reverse order.

<code>std::ranges::basic_istream_view</code>	Applies operator>> on the view
<code>std::ranges::istream_view</code>	
<code>std::ranges::elements_view</code>	Creates a view on the n-th element of tuple
<code>std::views::elements</code>	
<code>std::ranges::keys_view</code>	Creates a view on the first element of pair-values.
<code>std::views::keys</code>	
<code>std::ranges::values_view</code>	Creates a view on the second elements of pair-like values.
<code>std::views::values</code>	

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

Thanks to the ranges library, algorithms can directly be applied to the containers, can be composed, and are lazy.

Direct on the Containers

The algorithms of the Standard Template Library need a begin and an end iterator.

```
std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto res = std::accumulate(std::begin(myVec), std::end(myVec), 0);
std::cout << res << std::endl; // 45
```

The ranges library allows it to directly create a view on the keys (1) or values (3) on the a `std::unordered_map`.

Ranges work directly on the container

```
// rangesEntireContainer.cpp
...
#include <ranges>

std::unordered_map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33},
                                              {"tale", 45}, {"dog", 4},
                                              {"cat", 34}, {"fish", 23} };
std::cout << "Keys" << std::endl;
auto names = std::views::keys(freqWord); // (1)
```

```

for (const auto& name : names){ std::cout << name << " "; }

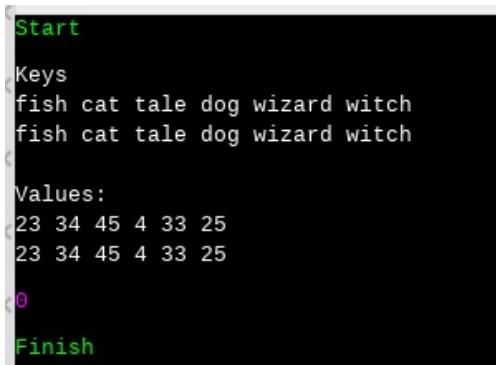
for (const auto& na : std::views::keys(freqWord)){ std::cout << na << " "; } // (2)

std::cout << "Values: " << std::endl;
auto values = std::views::values(freqWord); // (3)
for (const auto& value : values){ std::cout << value << " "; }

for (const auto& value : std::views::values(freqWord)){ // (4)
    std::cout << value << " ";
}

```

Of course, the keys and values can be displayed directly (marker (2) and (4)).
The output is identical.



```

Start
Keys
fish cat tale dog wizard witch
fish cat tale dog wizard witch

Values:
23 34 45 4 33 25
23 34 45 4 33 25

Finish

```

Function Composition

The ranges library supports function composition using the | symbol.

Ranges work directly on the container

```

// rangesComposition.cpp
...
#include <ranges>

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

std::cout << "All words: " // (1)
for (const auto& name : std::views::keys(freqWord)) { std::cout << name << " "; } \

std::cout << "All words reverse: " // (2)
for (const auto& name : std::views::keys(freqWord) | std::views::reverse) {
    std::cout << name << " ";
}

std::cout << "The first 4 words: " // (3)
for (const auto& name : std::views::keys(freqWord) | std::views::take(4)) {
    std::cout << name << " ";
}

```

```

std::cout << "All words starting with w: " // (4)
auto firstw = [] (const std::string& name){ return name[0] == 'w'; };
for (const auto& name : std::views::keys(freqWord) | std::views::filter(firstw)) {
    std::cout << name << " ";
}

```

In this case, I'm only interested in the keys. I display all of them (1), all of them reversed (2), the first four (3), and the keys starting with the letter 'w' (4).

The pipe symbol | is syntactic sugar for function composition. Instead of $C(R)$ you can write $R \mid C$. Consequentially, the next three lines are equivalent.

```

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;

```

Finally, here is the output of the program:

```

Start
All words: cat dog fish tale witch wizard
All words reverse: wizard witch tale fish dog cat
The first 4 words: cat dog fish tale
All words starting with w: witch wizard

0

Finish

```

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. Thanks to the function, I can find the first 20 prime numbers starting with 1000000.

Finding 20 prime numbers starting with 1000000

```

// rangesLazy.cpp
...
#include <ranges>

bool isPrime(int i) {
    for (int j=2; j*j <= i; ++j){
        if (i % j == 0) return false;
    }
    return true;
} // (1)
std::cout << "Numbers from 1000000 to 1001000 (displayed each 100th): " << "\n";

```

```

for (int i: std::views::iota(1000000, 1001000)) {
    if (i % 100 == 0) std::cout << i << " ";
}
                                            // (2)
auto odd = [] (int i){ return i % 2 == 1; };
std::cout << "Odd numbers from 1000000 to 1001000 (displayed each 100th): " << "\n";
for (int i: std::views::iota(1000000, 1001000) | std::views::filter(odd)) {
    if (i % 100 == 1) std::cout << i << " ";
}
                                            // (3)
std::cout << "Prime numbers from 1000000 to 1001000: " << std::endl;
for (int i: std::views::iota(1000000, 1001000) | std::views::filter(odd)
                                             | std::views::filter(isPrime)) {
    std::cout << i << " ";
}
                                            // (4)
std::cout << "20 prime numbers starting with 1000000: " << std::endl;
for (int i: std::views::iota(1000000) | std::views::filter(odd)
                                             | std::views::filter(isPrime)
                                             | std::views::take(20)) {
    std::cout << i << " ";
}

```

Here is my iterative strategy:

1. I don't know when I have 20 primes greater than 1000000. To be on the safe side, I create 1000 numbers. I displayed only each 100th.
2. A prime is an odd number; therefore, I remove the even numbers.
3. The predicate `isPrime` returns if a number is a prime. I get 75 primes, but I only want to have 20.
4. I use `std::iota` as an infinite number factory, starting with 1000000 and ask precisely for 20 primes.

```

Numbers from 1000000 to 1001000 (displayed each 100th):
1000000 1000100 1000200 1000300 1000400 1000500 1000600 1000700 1000800 1000900

Odd numbers from 1000000 to 1001000 (displayed each 100th):
1000001 1000101 1000201 1000301 1000401 1000501 1000601 1000701 1000801 1000901

Prime numbers from 1000000 to 1001000:
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 1000000:
1000003 1000033 1000037 1000039 1000081 1000099 1000117 1000121 1000133 1000151 1000159 1000171 1000183 1000187 1000193 1000199 1000211 1000213
1000231 1000249

```

12. Numeric

C++ inherits the numeric functions from C and has a random number library.

Random Numbers

[Random numbers](#) are necessary for many domains, e.g., to test software, to generate cryptographic keys or for computer games. The random number facility of C++ consists of two components. There is the generation of the random numbers, and there is the distribution of these random numbers. Both components need the header <random>.

Random Number Generator

The random number generator generates a random number stream between a minimum and maximum value. This stream is initialized by a “so-called” *seed*, guaranteeing different sequences of random numbers.

```
#include <random>
...
std::random_device seed;
std::mt19937 generator(seed);
```

A random number generator gen of type Generator supports four different requests:

Generator::result_type

 Data type of the generated random number.

gen()

 Returns a random number.

gen.min()

 Returns the minimum random number that can be returned by gen().

gen.max()

 Returns the maximum random number that can be returned by gen.

The random number library supports several random number generators. The best known are the Mersenne Twister, the std::default_random_engine that is chosen by the implementation and std::random_device. std::random_device is the only true random number generator, but not all platforms offer it.

Random Number Distribution

The random number distribution maps the random number with the help of the random number generator gen to the selected distribution.

```
#include <random>
...
std::random_device seed;
std::mt19937 gen(seed());
std::uniform_int_distribution<> unDis(0, 20); // distribution between 0 and 20
unDis(gen); // generates a random number
```

C++ has several discrete and continuous random number distributions. The discrete random number distribution generates integers, the continuous random number distribution generates floating-point numbers.

```
class bernoulli_distribution;
template<class T = int> class uniform_int_distribution;
template<class T = int> class binomial_distribution;
template<class T = int> class geometric_distribution;
template<class T = int> class negative_binomial_distribution;
template<class T = int> class poisson_distribution;
template<class T = int> class discrete_distribution;
template<class T = double> class exponential_distribution;
template<class T = double> class gamma_distribution;
template<class T = double> class weibull_distribution;
template<class T = double> class extreme_value_distribution;
template<class T = double> class normal_distribution;
template<class T = double> class lognormal_distribution;
template<class T = double> class chi_squared_distribution;
template<class T = double> class cauchy_distribution;
template<class T = double> class fisher_f_distribution;
template<class T = double> class student_t_distribution;
template<class T = double> class piecewise_constant_distribution;
template<class T = double> class piecewise_linear_distribution;
template<class T = double> class uniform_real_distribution;
```

Class templates with a default template argument `int` are discrete. The Bernoulli distribution generates booleans.

Here is an example using the Mersenne Twister `std::mt19937` as the pseudo random-number generator for generating 1 million random numbers. The random number stream is mapped to the uniform and normal (or Gaussian) distribution.

Random numbers

```
// random.cpp
...
#include <random>
...
static const int NUM= 1000000;
```

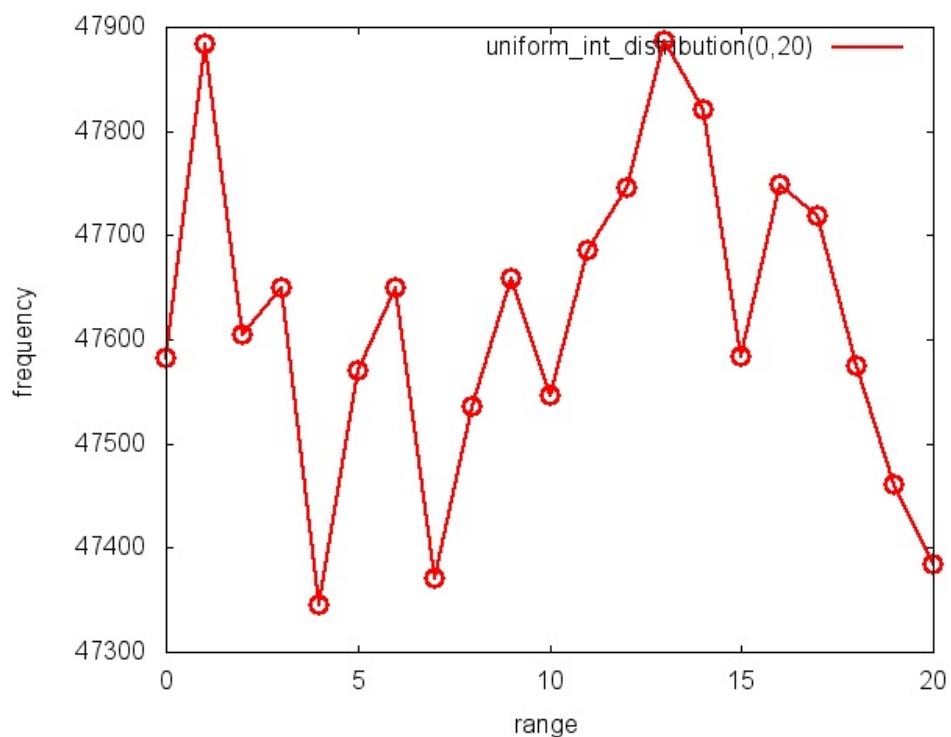
```

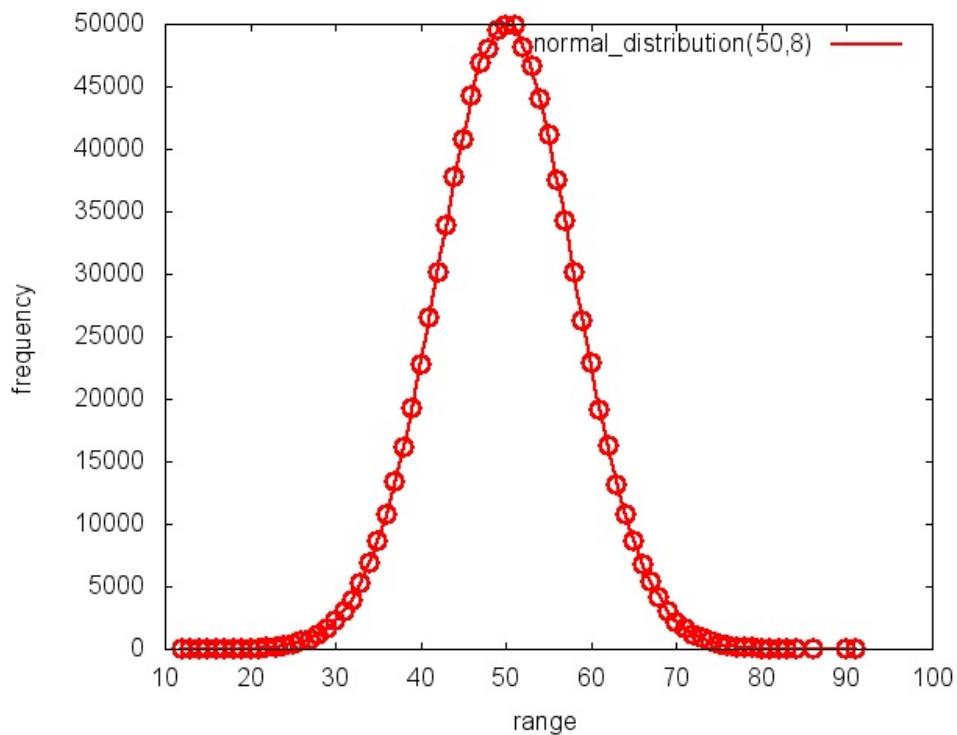
std::random_device seed;
std::mt19937 gen(seed());
std::uniform_int_distribution<> uniformDist(0, 20); // min= 0; max= 20
std::normal_distribution<> normDist(50, 8);           // mean= 50; sigma= 8

std::map<int, int> uniformFrequency;
std::map<int, int> normFrequency;
for (int i= 1; i <= NUM; ++i){
    ++uniformFrequency[uniformDist(gen)];
    ++normFrequency[round(normDist(gen))];
}

```

The following pictures show the uniform and the normal distribution of the 1 million random numbers as a plot.





Numeric Functions Inherited from C

C++ inherited many numeric functions from C. They need the header [`<cmath>`](#). The table below shows the names of these functions.

Mathematical functions in <code><cmath></code>				
<code>pow</code>	<code>sin</code>	<code>tanh</code>	<code>asinh</code>	<code>fabs</code>
<code>exp</code>	<code>cos</code>	<code>asin</code>	<code>acosh</code>	<code>fmod</code>
<code>sqrt</code>	<code>tan</code>	<code>acos</code>	<code>atanh</code>	<code>frexp</code>
<code>log</code>	<code>sinh</code>	<code>atan</code>	<code>ceil</code>	<code>ldexp</code>
<code>log10</code>	<code>cosh</code>	<code>atan2</code>	<code>floor</code>	<code>modf</code>

Additionally, C++ inherits further mathematical functions from C. They are defined in the header [`<cstdlib>`](#). Once more, the names.

Mathematical functions in <code><cstdlib></code>				
<code>abs</code>	<code>llabs</code>	<code>ldiv</code>	<code>srand</code>	
<code>labs</code>	<code>div</code>	<code>lldiv</code>	<code>rand</code>	

All functions for integers are available for the types `int`, `long` and `long long`; all functions for floating point numbers are available for the types `float`, `double` and `long double`.

The numeric functions need to be qualified with the namespace `std`.

Mathematic functions

```
// mathFunctions.cpp
...
#include <cmath>
#include <cstdlib>
...

std::cout << std::pow(2, 10);      // 1024
std::cout << std::pow(2, 0.5);    // 1.41421
std::cout << std::exp(1);        // 2.71828
std::cout << std::ceil(5.5);     // 6
std::cout << std::floor(5.5);   // 5
std::cout << std::fmod(5.5, 2); // 1.5

double intPart;
auto fracPart= std::modf(5.7, &intPart);
std::cout << intPart << " + " << fracPart;           // 5 + 0.7
std::div_t divresult= std::div(14, 5);
std::cout << divresult.quot << " " << divresult.rem; // 2 4

// seed
std::srand(time(nullptr));
for (int i= 0;i < 10; ++i) std::cout << (rand()%6 + 1) << " ";
// 3 6 5 3 6 5 6 3 1 5
```

Mathematical Constants

C++ supports basic and advanced mathematical constants. They are in the namespace `std::numbers` and part of the header `<numbers>`. The mathematical constants have the data type `double`.

The mathematical constants	
Mathematical Constant	Meaning
<code>std::numbers::e</code>	e
<code>std::numbers::log2e</code>	$\log_2 e$
<code>std::numbers::log10e</code>	$\log_{10} e$
<code>std::numbers::pi</code>	π

<code>std::numbers::inv_pi</code>	$\frac{1}{\pi}$
<code>std::numbers::inv_sqrt(pi)</code>	$\frac{1}{\sqrt{\pi}}$
<code>std::numbers::ln2</code>	$\ln 2$
<code>std::numbers::ln10</code>	$\ln 10$
<code>std::numbers::sqrt2</code>	$\sqrt{2}$
<code>std::numbers::sqrt3</code>	$\sqrt{3}$
<code>std::numbers::inv_sqrt3</code>	$\frac{1}{\sqrt{3}}$
<code>std::numbers::egamma</code>	Euler-Mascheroni constant
<code>std::numbers::phi</code>	ϕ

The following code snippet displays all mathematical constants.

Mathematic functions

```
// mathematicalConstants.cpp
#include <numbers>
...
std::cout << std::setprecision(10);

std::cout << "std::numbers::e: " << std::numbers::e << std::endl;
std::cout << "std::numbers::log2e: " << std::numbers::log2e << std::endl;
std::cout << "std::numbers::log10e: " << std::numbers::log10e << std::endl;
std::cout << "std::numbers::pi: " << std::numbers::pi << std::endl;
std::cout << "std::numbers::inv_pi: " << std::numbers::inv_pi << std::endl;
std::cout << "std::numbers::inv_sqrt(pi): " << std::numbers::inv_sqrt(pi) << std::endl;
std::cout << "std::numbers::ln2: " << std::numbers::ln2 << std::endl;
std::cout << "std::numbers::sqrt2: " << std::numbers::sqrt2 << std::endl;
std::cout << "std::numbers::sqrt3: " << std::numbers::sqrt3 << std::endl;
std::cout << "std::numbers::inv_sqrt3: " << std::numbers::inv_sqrt3 << std::endl;
std::cout << "std::numbers::egamma: " << std::numbers::egamma << std::endl;
std::cout << "std::numbers::phi: " << std::numbers::phi << std::endl;
```

```
Windows PowerShell

C:\Users\rainer>mathematicalConstants.exe

std::numbers::e: 2.718281828
std::numbers::log2e: 1.442695041
std::numbers::log10e: 0.4342944819
std::numbers::pi: 3.141592654
std::numbers::inv_pi: 0.3183098862
std::numbers::inv_sqrt(pi): 0.5641895835
std::numbers::ln2: 0.6931471806
std::numbers::sqrt2: 1.414213562
std::numbers::sqrt3: 1.732050808
std::numbers::inv_sqrt3: 0.5773502692
std::numbers::egamma: 0.5772156649
std::numbers::phi: 1.618033989

C:\Users\rainer>
```

13. Strings

A [string](#) is a sequence of characters. C++ has many methods to analyse or to change a string. C++-strings are the safe replacement for C Strings: `const char*`. Strings need the header `<string>`.



A string is similar to a `std::vector`

A string feels like a `std::vector` containing characters. It supports a very similar interface. This means that in addition to the methods of the string class, you have the [algorithms of the Standard Template Library](#) to work with the string.

The following code snippet has the `std::string` `name` with the value `RainerGrimm`. I use the STL algorithm `std::find_if` to get the upper letter and then extract my first and last name into the variables `firstName` and `lastName`. The expression `name.begin() + 1` shows, that strings support [random access iterators](#):

string versus vector

```
// string.cpp
...
#include <algorithm>
#include <string>

std::string name{"RainerGrimm"};
auto strIt= std::find_if(name.begin() + 1, name.end(),
                         [] (char c){ return std::isupper(c); });
if (strIt != name.end()){
    firstName= std::string(name.begin(), strIt);
    lastName= std::string(strIt, name.end());
}
```

Strings are class templates parametrised by their character, their character trait and their allocator. The character trait and the allocator have defaults.

```
template <typename charT,
          typename traits= char_traits<charT>,
```

```
    typename Allocator= allocator<charT> >
class basic_string;
```

C++ has synonyms for the character types `char`, `wchar_t`, `char16_t` and `char32_t`

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
```



std::string is the string

If we speak in C++ about a string, we refer with 99 % probability to the specialisation `std::basic_string` for the character type `char`. This statement is also true for this book.

Create and Delete

C++ offers many methods to create strings from C or C++-strings. Under the hood there is always a C string involved for creating a C++-string. That changes with C++14, because the new C++ standard support C++-string literals: `std::string str{"string"s}`. The C string literals "string literal" becomes with the suffix `s` a C++-string literal: "string literal"s.

The table gives you an overview of the methods to create and delete a C++-string.

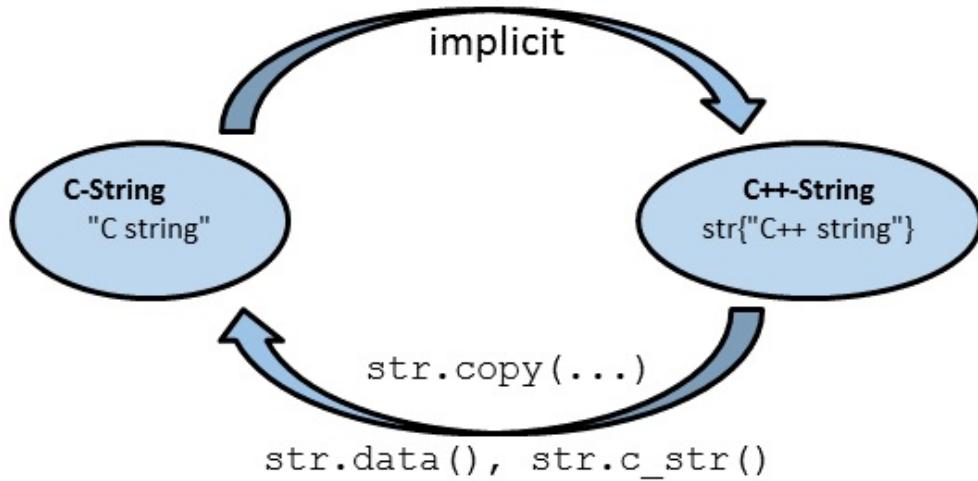
Methods	Methods to create and delete a string
Default	<code>std::string str</code>
Copies from a C++-string	<code>std::string str(oth)</code>
Moves from a C++-string	<code>std::string str(std::move(oth))</code>
From the range of a C++-string	<code>std::string(oth.begin(), oth.end())</code>

From a substring of a C++-string	<code>std::string(oth, otherIndex)</code>
From a substring of a C++-string	<code>std::string(oth, otherIndex, strlen)</code>
From a C string	<code>std::string str("c-string")</code>
From a C array	<code>std::string str("c-array", len)</code>
From characters	<code>std::string str(num, 'c')</code>
From a initializer list	<code>std::string str({'a', 'b', 'c', 'd'})</code>
From a substring	<code>str= other.substring(3, 10)</code>
Destructor	<code>str~string()</code>

Creation of a string

```
// stringConstructor.cpp
...
#include <string>
...
std::string defaultString;
std::string other{"123456789"};
std::string str1(other);                                // 123456789
std::string tmp(other);                                // 123456789
std::string str2(std::move(tmp));                      // 123456789
std::string str3(other.begin(), other.end());          // 123456789
std::string str4(other, 2);                            // 3456789
std::string str5(other, 2, 5);                          // 34567
std::string str6("123456789", 5);                     // 12345
std::string str7(5, '1');                             // 11111
std::string str8({'1', '2', '3', '4', '5'});          // 12345
std::cout << str6.substr();                           // 12345
std::cout << str6.substr(1);                          // 2345
std::cout << str6.substr(1, 2);                      // 23
```

Conversion Between C++ and C Strings



While the conversion of a C string in a C++-string is done implicitly, you must explicitly request the conversion from a C++-string into a C string. `str.copy()` copies the content of a C++-string without the terminating \0 character. `str.data()` and `str.c_str()` includes the terminating null character.



Be careful with `str.data()` and `str.c_str()`

The return value of the two methods `str.data()` and `std::c_str()` is invalid, if you modify the `str`.

C versus C++-strings

```
// stringCversusC++.cpp
...
#include<string>
...

std::string str{"C++-String"};
str += " C-String";
std::cout << str;                                // C++-String C-String
const char* cString= str.c_str();
char buffer[10];
str.copy(buffer, 10);
str+= "works";
// const char* cString2= cString; // ERROR
std::string str2(buffer, buffer+10);
std::cout<< str2;                                // C++-String
```

Size versus Capacity

The number of elements a string has (`str.size()`) is in general smaller than the number of elements, for which space is reserved: `str.capacity()`. Therefore if you add elements to a string, there will not automatically be new memory allocated. `std::max_size()` return how many elements a string can maximal have. For the three methods the following relation holds: `str.size() <= str.capacity() <= str.max_size()`.

The following table shows the methods for dealing with the memory management of the string.

Methods	Methods to create and delete a string Description
<code>str.empty()</code>	Checks if <code>str</code> has elements.
<code>str.size(), str.length()</code>	Number of elements of the <code>str</code> .
<code>str.capacity()</code>	Number of elements <code>str</code> can have without reallocation.
<code>str.max_size()</code>	Number of elements <code>str</code> can maximal have.
<code>str.resize(n)</code>	Increases <code>str</code> to <code>n</code> elements.
<code>str.reserve(n)</code>	Reserve memory for a least <code>n</code> elements.
<code>str.shrink_to_fit()</code>	Adjusts the capacity of the string to it's size.

The request `str.shrink_to_fit()` is as in the case of [`std::vector`](#) non-binding.

Size versus capacity

```
// stringSizeCapacity.cpp
...
#include <string>
...

void showStringInfo(const std::string& s){
    std::cout << s << ":";
```

```
    std::cout << s.size() << " ";
    std::cout << s.capacity() << " ";
    std::cout << s.max_size() << " ";
}

std::string str;
showStringInfo(str); // "": 0 0 4611686018427387897

str += "12345";
showStringInfo(str); // "12345": 5 5 4611686018427387897

str.resize(30);
showStringInfo(str); // "12345": 30 30 4611686018427387897

str.reserve(1000);
showStringInfo(str); // "12345": 30 1000 4611686018427387897

str.shrink_to_fit();
showStringInfo(str); // "12345": 30 30 4611686018427387897
```

Comparison

Strings support the well-known comparison operators ==, !=, <, >, >=. The comparison of two strings takes place on their elements.

String comparison

```
// stringComparisonAndConcatenation.cpp
...
#include <string>
...

std::string first{"aaa"};
std::string second{"aaaa"};

std::cout << (first < first) << std::endl; // false
std::cout << (first <= first) << std::endl; // true
std::cout << (first < second) << std::endl; // true
```

String Concatenation

The + operator is overloaded for strings, so you can *add* strings.



The + operator is only overloaded for C++-strings

The C++ type system permits it to concatenate C++ and C strings to C++-strings, but not to concatenate C++ and C strings to C strings. The reason is that the + operator is overloaded for C++-strings. Therefore only the second line is valid C++, because the C is implicitly converted to a C++-string:

String Concatenation

```
// stringComparisonAndConcatenation.cpp
...
#include <string>
...
std::string wrong= "1" + "1"; // ERROR
std::string right= std::string("1") + "1"; // 11
```

Element Access

The access to the elements of a string str is very convenient because the string supports [random access iterators](#). You can access with str.front() the first character and with str.back() the last character of the string. With str[n] and str.at(n) you get the n-th element by index.

The following table provides an overview.

Access the elements of the string	
Methods	Example
str.front()	Returns the first character of str.
str.back()	Returns the last character of str.
str[n]	Returns the n-th character of str. The string boundaries will not be checked .
str.at(n)	Returns the n-th character of str. The string boundaries will be checked . If the boundaries are violated a std::out_of_range exception is thrown.

```

// stringAccess.cpp
...
#include <string>
...

std::string str= {"0123456789"};
std::cout << str.front() << std::endl;           // 0
std::cout << str.back() << std::endl;           // 9
for (int i= 0; i <= 3; ++i){
    std::cout << "str[" << i << "]:" << str[i] << ";" ;
} // str[0]: 0; str[1]: 1; str[2]: 2; str[3]: 3;

std::cout << str[10] << std::endl;           // undefined behaviour
try{
    str.at(10);
}
catch (const std::out_of_range& e){
    std::cerr << "Exception: " << e.what() << std::endl;
} // Exception: basic_string::at

std::cout << *(&str[0]+5) << std::endl;           // 5
std::cout << *(&str[5]) << std::endl;           // 5
std::cout << str[5] << std::endl;           // 5

```

It is particularly interesting to see in the example that the compiler performs the invocation `str[10]`. The access outside the string boundaries is *undefined behaviour*. On the contrary, the compiler complains the call `str.at(10)`.

Input and Output

A string can read from an input stream via `>>` and write to an output stream via `<<`.

The global function `getline` empowers you to read from an input stream line by line until the *end-of-file* character.

There are four variations of the `getline` function available. The first two arguments are the input stream `is` and the string `line` holding the line read. Optionally you can specify a special line separator. The function returns by reference to the input stream.

```

istream& getline (istream& is, string& line, char delim);
istream& getline (istream&& is, string& line, char delim);
istream& getline (istream& is, string& line);
istream& getline (istream&& is, string& line);

```

`getline` consumes the whole line including empty spaces. Only the line separator is ignored. The function needs the header `<string>`.

```
// stringInputOutput.cpp
...
#include <string>
...

std::vector<std::string> readFromFile(const char* fileName){
    std::ifstream file(fileName);
    if (!file){
        std::cerr << "Could not open the file " << fileName << ".";
        exit(EXIT_FAILURE);
    }
    std::vector<std::string> lines;
    std::string line;
    while (getline(file, line)) lines.push_back(line);
    return lines;
}

std::string fileName;
std::cout << "Your filename: ";
std::cin >> fileName;
std::vector<std::string> lines= readFromFile(fileName.c_str());
int num{0};
for (auto line: lines) std::cout << ++num << ": " << line << std::endl;
```

The program displays the lines of an arbitrary file including their line number. The expression `std::cin >> fileName` reads the file name. The function `readFromFile` reads with `getline` all file lines and pushes them onto the vector.

Search

C++ offers the ability to search in a string in many variations. Each variation exists in various overloaded forms.



Search is called find

Odd enough the algorithms for search in a string starts with the name `find`. If the search was successful, you get the index of type `std::string::size_type`, if not, you get the constant `std::string::npos`. The first character has the index 0.

The `find` algorithms support to:

- search for a character, a C String or a C++-string,
- search for a character from a C or C++-string,
- search forward and backward,

- search positive (does contain) or negative (does not contain) for characters from a C or C++-string,
- start the search at an arbitrary position in the string.

The arguments of all six variations of the find functions follow a similar pattern. The first argument is the text you are searching for. The second argument holds the start position of the search and the third the number of characters starting from the second argument.

Here are the six variations.

Methods	Find variations of the string
	Description
<code>str.find(...)</code>	Returns the first position of a character, a C or C++-string in str.
<code>str.rfind(...)</code>	Returns the last position of a character, a C or C++-string in str.
<code>str.find_first_of(...)</code>	Returns the first position of a character from a C or C++-string in str.
<code>str.find_last_of(...)</code>	Returns the last position of a character from a C or C++-string in str.
<code>str.find_first_not_of(...)</code>	Returns the first position of a character in str, which is not from a C or C++-string.
<code>str.find_last_not_of(...)</code>	Returns the last position of a character in str, which is not from a C or C++-string.

Find(search) in a string

```
// stringFind.cpp
...
#include <string>
...

std::string str;

auto idx= str.find("no");
if (idx == std::string::npos) std::cout << "not found"; // not found
```

```

str= {"dkeu84kf8k48kdj39kdj74945du942"};
std::string str2{"84"};

std::cout << str.find('8');                                // 4
std::cout << str.rfind('8');                             // 11
std::cout << str.find('8', 10);                          // 11
std::cout << str.find(str2);                            // 4
std::cout << str.rfind(str2);                           // 4
std::cout << str.find(str2, 10);                         // 18446744073709551615

str2="0123456789";

std::cout << str.find_first_of("678");                  // 4
std::cout << str.find_last_of("678");                 // 20
std::cout << str.find_first_of("678", 10);             // 11
std::cout << str.find_first_of(str2);                  // 4
std::cout << str.find_last_of(str2);                  // 29
std::cout << str.find_first_of(str2, 10);              // 10
std::cout << str.find_first_not_of("678");            // 0
std::cout << str.find_last_not_of("678");             // 29
std::cout << str.find_first_not_of("678", 10);        // 10
std::cout << str.find_first_not_of(str2);              // 0
std::cout << str.find_last_not_of(str2);              // 26
std::cout << str.find_first_not_of(str2, 10);         // 12

```

The call `std::find(str2, 10)` returns `std::string::npos`. If I display that value I get on my platform 18446744073709551615.

Check for a Prefix or a Suffix

The member function `str.starts_with(prefix)` and `str.end_with(suffix)` check for a given string `str` if it begins with a prefix or ends with a suffix.

Check if the string starts with or ends with a prefix or suffix

```

// startWithEndsWith.cpp
...
#include <string>
...
std::string helloWorld = "hello world";

std::cout << helloWorld.starts_with("hello") << std::endl; // true
std::cout << helloWorld.starts_with("llo") << std::endl;   // false

std::cout << helloWorld.ends_with("world") << std::endl;  // true
std::cout << helloWorld.ends_with("wo") << std::endl;    // false

```

Modifying Operations

Strings have many operations to modify them. `str.assign` assigns a new string to the string `str`. With `str.swap` you can swap two strings. To remove a character from a string use `str.pop_back` or `str.erase`. In contrary `str.clear` or `str.erase` deletes the whole string. To append new characters to a string use

`+=`, `std.append` or `str.push_back`. You can use `str.insert` to insert new characters or `str.replace` to replace characters.

Methods	Methods for modifying a string
	Description
str= str2	Assigns str2 to str.
str.assign(...)	Assigns to str a new string.
str.swap(str2)	Swaps str and str2.
str.pop_back()	Removes the last character from str.
str.erase(...)	Removes characters from str.
str.clear()	Clears the str.
str.append(...)	Appends characters to str.
str.push_back(s)	Appends the character s to str.
str.insert(pos, ...)	Inserts characters in str starting at pos.
str.replace(pos, len, ...)	Replaces the len characters from str starting at pos

The operations are available in many overloaded versions. The methods `str.assign`, `str.append`, `str.insert` and `str.replace` are very similar. All four can be invoked with C++-strings and substrings, but also characters, C strings, C string arrays, ranges and initialiser lists. `str.erase` can erase a single character, ranges, but also many characters starting at a given position.

The following code snippet shows many of the variations. For simplicity reasons only the effects of the strings modifications are displayed:

Modifying strings

```
// stringModification.cpp
...
```

```

#include <string>
...
std::string str{"New String"};
std::string str2{"Other String"};

str.assign(str2, 4, std::string::npos); // r String
str.assign(5, '-'); // ----

str= {"0123456789"};
str.erase(7, 2); // 01234569
str.erase(str.begin()+2, str.end()-2); // 012
str.erase(str.begin()+2, str.end()); // 01
str.pop_back(); // 0
str.erase(); // 

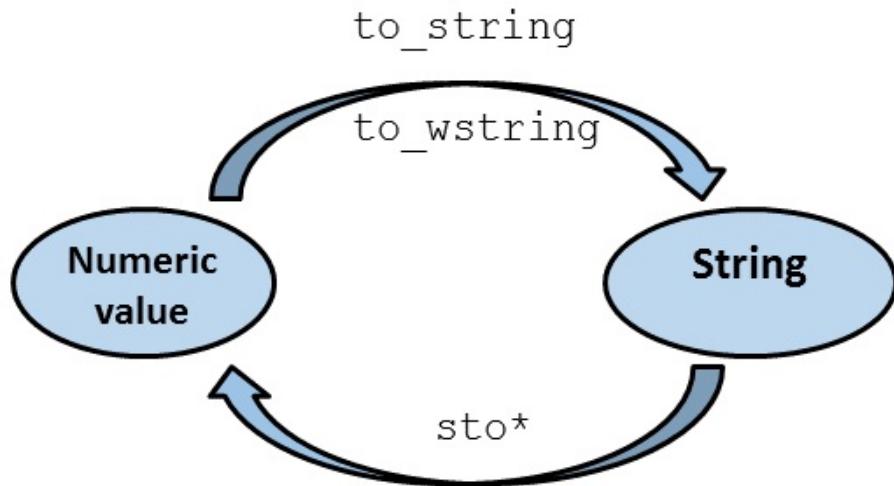
str= "01234";
str+= "56"; // 0123456
str+= '7'; // 01234567
str+= {'8', '9'}; // 0123456789
str.append(str); // 01234567890123456789
str.append(str, 2, 4); // 012345678901234567892345
str.append(3, '0'); // 012345678901234567892345000
str.append(str, 10, 10); // 01234567890123456789234500001234567989
str.push_back('9'); // 012345678901234567892345000012345679899

str= {"345"};
str.insert(3, "6789"); // 3456789
str.insert(0, "012"); // 0123456789

str= {"only for testing purpose."};
str.replace(0, 0, "0"); // Only for testing purpose.
str.replace(0, 5, "Only", 0, 4); // Only for testing purpose.
str.replace(16, 8, ""); // Only for testing.
str.replace(4, 0, 5, 'y'); // Onlyyyyyy for testing.
str.replace(str.begin(), str.end(), "Only for testing purpose."); // Only for testing purpose.
str.replace(str.begin() + 4, str.end() - 8, 10, '#'); // Only#####purpose.

```

Numeric Conversions



You can convert with `std::to_string(val)` and `std::to_wstring(val)` numbers or floating point numbers to the corresponding `std::string` or `std::wstring`. For the opposite direction for the numbers or floating point numbers, you have the function family of the `sto*` functions. All functions need the header `<string>`.



Read `sto*` as string to

The seven ways to convert a string into a natural or floating point number follow a simple pattern. All functions start with `sto` and add further characters, denoting the type to which the strings should be converted to. E.g. `stoi` stands for string to long or `stod` for string to double.

The `sto` functions all have the same interface. The example shows it for the type `long`.

```
std::stol(str, idx= nullptr, base= 10)
```

The function takes a string and determines the `long` representation to the base `base`. `stol` ignores leading spaces and optionally returns the index of the first invalid character in `idx`. By default, the base is 10. Valid values for the base are 0 and 2 until 36. If you use base 0 the compiler automatically determines the

type based on the format of the string. If the base is bigger than 10 the compiler encodes them in the characters a until z. The representation is analogous to the representation of hexadecimal numbers.

The table gives an overview of all functions.

Method	Numeric conversion of strings
Description	
std::to_string(val)	Converts val into a std::string.
std::to_wstring(val)	Converts val into a std::wstring.
std::stoi(str)	Returns an int value.
std::stol(str)	Returns a long value.
std::stoll(str)	Returns a long long value.
std::stoul(str)	Returns an unsigned long value.
std::stoull(str)	Returns an unsigned long long value.
std::stof(str)	Returns a float value.
std::stod(str)	Returns a double value.
std::stold(str)	Returns an long double value.



Where is the stou function?

In case you're curious, the C++ sto functions are thin wrappers around the C `strto*` functions, but there is no `strtou` function in C. Therefore C++ has no `stou` function.

The functions throw a `std::invalid_argument` exception if the conversion is not possible. If the determined value is too big for the destination type you get a `std::out_of_range` exception.

Numeric conversion

```
stringNumericConversion.cpp
...
#include <string>
...

std::string maxLongLongString=
    std::to_string(std::numeric_limits<long long>::max());
std::wstring maxLongLongWstring=
    std::to_wstring(std::numeric_limits<long long>::max());

std::cout << std::numeric_limits<long long>::max();           // 9223372036854775807
std::cout << maxLongLongString;                                // 9223372036854775807
std::wcout << maxLongLongWstring;                             // 9223372036854775807

std::string str("10010101");
std::cout << std::stoi(str);                                    // 10010101
std::cout << std::stoi(str, nullptr, 16);                     // 268501249
std::cout << std::stoi(str, nullptr, 8);                      // 2101313
std::cout << std::stoi(str, nullptr, 2);                      // 149

std::size_t idx;
std::cout << std::stod(" 3.5 km", &idx);                  // 3.5
std::cout << idx;                                         // 6

try{
    std::cout << std::stoi(" 3.5 km") << std::endl;        // 3
    std::cout << std::stoi(" 3.5 km", nullptr, 2) << std::endl;
}
catch (const std::exception& e){
    std::cerr << e.what() << std::endl;                    // stoi
}
```

14. String Views

A [string view](#) is a non-owning reference to a string. It represents a view of a sequence of characters. This sequence of characters can be a C++-string or a C-string. A string view needs the header `<string_view>`.



A string view is a for copying optimised string

From a birds-eye perspective the purpose of `std::string_view` is to avoid copying data which is already owned by someone else and to allow immutable access to a `std::string` like object. The string view is a kind of a restricted string that supports only the immutable operations.

Additionally, a string view `sv` has two additional mutating operations: `sv.remove_prefix` and `sv.remove_suffix`.

String views are class templates parameterised by their character and their character trait. The character trait has a default. In contrast to a [string](#), a string view is non-owner and, therefore, needs no allocator.

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>
> class basic_string_view;
```

According to strings, there exist for string views four synonyms for the underlying character types `char`, `wchar_t`, `char16_t` and `char32_t`.

```
typedef std::string_view std::basic_string_view<char>
typedef std::wstring_view std::basic_string_view<wchar_t>
typedef std::u16string_view std::basic_string_view<char16_t>
typedef std::u32string_view std::basic_string_view<char32_t>
```



std::string_view is the string view

If we speak in C++ about a string view, we refer with 99% probability to the specialisation `std::basic_string_view` for the character type `char`. This statement is also true for this book.

Create and Initialise

You can create an empty string view. You can also create a string view from an existing string, an existing character-array, or an existing string view.

The table below gives you an overview of the various ways of creating a string view.

Methods	Methods to create and set a string view
Example	
Empty string view	<code>std::string_view str_view</code>
From a C-string	<code>std::string_view str_view2("C-string")</code>
From a string view	<code>std::string_view str_view3(str_view2)</code>
From a C array	<code>std::string_view str_view4(arr, sizeof arr)</code>
From a string_view	<code>str_view4= str_view3.substring(2, 3)</code>
From a string view	<code>std::string_view str_view5 = str_view4</code>

Non-modifying operations

To make this chapter concise and not repeat the detailed descriptions from the chapter on strings, I only mention the non-modifying operations of the string view. For further details, please use the link to the associated documentation in the [string](#) chapter.

- *Element access*: operator[], at, front, back, data (see [string: element access](#))
- *Capacity*: size, length, max_size, empty (see [string: size versus capacity](#))
- *Find*: find, rfind, find_first_of, find_last_of, find_first_not_of, find_last_not_of (see [string: search](#))
- *Copy*: copy (see [string: conversion between a C++-string and a C-String](#))

Modifying operations

The call `stringView.swap(stringView2)` swaps the content of the two string views. The methods `remove_prefix` and `remove_suffix` are unique to a string view because a string supports neither. `remove_prefix` shrinks its start forward; `remove_suffix` shrinks its end backwards.

Non-modifying operations

```
// string_view.cpp
...
#include <string_view>
...

using namespace std;
string str = " A lot of space";
string_view strView = str;
strView.remove_prefix(min(strView.find_first_not_of(" "), strView.size()));
cout << str << endl // " A lot of space
<< strView << endl; // "A lot of space"

char arr[] = {'A', ' ', 'l', 'o', 't', ' ', 'o', 'f', ' ', 
's', 'p', 'a', 'c', 'e', '\0', '\0', '\0'};
string_view strView2(arr, sizeof arr);
auto trimPos = strView2.find('\0');
if(trimPos != strView2npos) strView2.remove_suffix(strView2.size() - trimPos);
cout << arr << ":" << sizeof arr << endl // A lot of space: 17
<< strView2 << ":" << strView2.size() << endl; // A lot of space: 14
```



No memory allocation with a string view

If you create a string view or copy a string view, there is no memory allocation necessary. This is in contrast to a string; creating a string or copying a string requires memory allocation.

Memory allocation

```
// stringView.cpp
...
#include <string_view>
...

void* operator new(std::size_t count){
    std::cout << " " << count << " bytes" << std::endl;
    return malloc(count);
}

void getString(const std::string&){}
void getStringView(std::string_view){}

std::string large = "012345678901234567890"
                    "1234567890123456789"; // 41 bytes allocated
std::string substr = large.substr(10); // 31 bytes allocated

std::string_view largeStringView{large.c_str(), // 0 bytes allocated
                                large.size()};
largeStringView.remove_prefix(10); // 0 bytes allocated

getString(large);
getString("012345678901234567890"
          "1234567890123456789"); // 41 bytes allocated
const char message []= "0123456789012345678901234567890123456789";
getString(message); // 41 bytes allocated

getStringView(large); // 0 bytes allocated
getStringView("012345678901234567890"
              "1234567890123456789"); // 0 bytes allocated
getStringView(message); // 0 bytes allocated
```

Thanks to the global overload operator `new` I can observe each memory allocation.

15. Regular Expressions

[Regular expression](#) is a language for describing text patterns. They need the header <regex>.

Regular expressions are a powerful tool for the following tasks:

- Check if a text matches a text pattern: std::regex_match
- Search for a text pattern in a text: std::regex_search
- Replace a text pattern with a text: std::regex_replace
- Iterate through all text patterns in a text: std::regex_iterator and std::regex_token_iterator

C++ supports six different grammars for regular expressions. By default, the ECMAScript grammar is used. This one is the most powerful grammar of the six grammars and is quite similar to the grammar used in Perl 5. The other five grammars are the basic, extended, awk, grep and egrep grammars.



Use raw strings

Use raw string literals in regular expressions. The regular expression for the text C++ is quite ugly: C\\+\\.+. You have to use two backslashes for each + sign. First, the + sign is a special character in a regular expression. Second, the backslash is a special character in a string. Therefore one backslash escapes the + sign, the other backslash escapes the backslash. By using a raw string literal the second backslash is not necessary any more, because the backslash is not be interpreted in the string.

```
#include <regex>
...
std::string regExpr("C\\+\\.+");
std::string regExprRaw(R"(C\+\.+)");
```

Dealing with regular expressions is typically done in three steps:

I. Define the regular expression:

```
std::string text="C++ or c++.";
std::string regExpr(R"(C\+\+)");
std::regex rgx(regExpr);
```

II. Store the result of the search:

```
std::smatch result;
std::regex_search(text, result, rgx);
```

III. Process the result:

```
std::cout << result[0] << std::endl;
```

Character Types

The type of the text determines the character type of the regular expression, and the type of the search result.

The table shows the four different combinations.

Combinations of type of text, regular expression, search result and action		
Text type	Regular expression type	Result type
const char*	std::regex	std::cmatch
std::string	std::regex	std::smatch
const wchar_t*	std::wregex	std::wcmatch
std::wstring	std::wregex	std::wsmatch

The program shown in the [Search](#) section of this chapter provides the four combinations in detail.

Regular Expression Objects

Objects of type regular expression are instances of the class template `template <class charT, class traits= regex_traits <charT>> class basic_regex` parametrized by their character type and traits class. The traits class defines the interpretation of the properties of regular grammar. There are two type synonyms in C++:

```
typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;
```

You can further customise the object of type regular expression. Therefore you can specify the used grammar or adapt the syntax. As said before, C++ supports the basic, extended, awk, grep and egrep grammars. A regular expression qualified by the `std::regex_constants::icase` flag is case insensitive. If you want to adopt the syntax, you have to specify the grammar explicitly.

Specify the grammar

```
// regexGrammar.cpp
...
#include <regex>
...
using std::regex_constants::ECMAScript;
using std::regex_constants::icase;

std::string theQuestion="C++ or c++, that's the question.";
std::string regExprStr(R"(c\+\+|c++)");

std::regex rgx(regExprStr);
std::smatch smatch;

if (std::regex_search(theQuestion, smatch, rgx)){
    std::cout << "case sensitive: " << smatch[0];           // c++
}

std::regex rgxIn(regExprStr, ECMAScript|icase);
if (std::regex_search(theQuestion, smatch, rgxIn)){
    std::cout << "case insensitive: " << smatch[0];        // C++
}
```

If you use the case-sensitive regular expression `rgx` the result of the search in the text `theQuestion` is `c++`. That's not the case if your case-insensitive regular expression `rgxIn` is applied. Now you get the match string `C++`.

The Search Result `match_results`

The object of type `std::match_results` is the result of a `std::regex_match` or `std::regex_search`. `std::match_results` is a sequence container having at least one capture group of a `std::sub_match` object. The `std::sub_match` objects are sequences of characters.



What is a capture group?

Capture groups allow it to further analyse the search result in a regular expression. They are defined by a pair of parentheses (). The regular expression `((a+)(b+)(c+))` has four capture groups: `((a+)(b+)(c+))`, `(a+)`, `(b+)` and `(c+)`. The total result is the 0th capture group.

C++ has four types of synonyms of type `std::match_results`:

```
typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t*> wcmatch;
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;
```

The search result `std::smatch` `smatch` has a powerful interface.

Method	Interface of <code>std::smatch</code>	Description
<code>smatch.size()</code>		Returns the number of capture groups.
<code>smatch.empty()</code>		Returns if the search result has a capture group.
<code>smatch[i]</code>		Returns the ith capture group.
<code>smatch.length(i)</code>		Returns the length of the ith capture group.
<code>smatch.position(i)</code>		Returns the position of the ith capture group.
<code>smatch.str(i)</code>		Returns the ith capture group as string.
<code>smatch.prefix()</code> and <code>smatch.suffix()</code>		Returns the string before and after the capture group.
<code>smatch.begin()</code> and <code>smatch.end()</code>		Returns the begin and end iterator for the capture groups.
<code>smatch.format(...)</code>		Formats <code>std::smatch</code> objects for the output.

The following program shows the output of the first four capture groups for different regular expressions.

```
Capture groups


---


// captureGroups.cpp
...
#include<regex>
...
using namespace std;
```

```

void showCaptureGroups(const string& regEx, const string& text){
    regex rgx(regEx);
    smatch smatch;
    if (regex_search(text, smatch, rgx)){
        cout << regEx << text << smatch[0] << " " << smatch[1]
        << " " << smatch[2] << " " << smatch[3] << endl;
    }
}

showCaptureGroups("abc+", "abcccc");
showCaptureGroups("(a+)(b+)", "aaabccc");
showCaptureGroups("((a+)(b+))", "aaabccc");
showCaptureGroups("(ab)(abc)+", "ababcabc");
...

```

reg Expr	text	smatch[0]	smatch[1]	smatch[2]	smatch[3]
abc+	abcccc	abcccc			
(a+)(b+)(c+)	aaabccc	aaabccc	aaa	b	ccc
((a+)(b+))(c+)	aaabccc	aaabccc	aaabccc	aaa	b
(ab)(abc)+	ababcabc	ababcabc	ab	abc	

std::sub_match

The capture groups are of type `std::sub_match`. As with [std::match_results](#) C++ defines the following four type synonyms.

```

typedef sub_match<const char*> csub_match;
typedef sub_match<const wchar_t*> wsub_match;
typedef sub_match<string::const_iterator> ssub_match;
typedef sub_match<wstring::const_iterator> wssub_match;

```

You can further analyze the capture group `cap`.

Method	The std::sub_match object
<code>cap.matched()</code>	Indicates if this match was successful.
<code>cap.first()</code> and <code>cap.end()</code>	Returns the begin and end iterator of the character sequence.
<code>cap.length()</code>	Returns the length of the capture group.
<code>cap.str()</code>	Returns the capture group as string.
<code>cap.compare(other)</code>	Compares the current capture group with another capture group.

Here is a code snippet showing the interplay between the search result `std::match_results` and its capture groups `std::sub_match`.

```
std::sub_match

---

// subMatch.cpp
...
#include <regex>
...

using std::cout;

std::string privateAddress="192.168.178.21";
std::string regEx(R"((\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3}))");
std::regex rgx(regEx);
std::smatch smatch;

if (std::regex_match(privateAddress, smatch, rgx)){
    for (auto cap: smatch){
        cout << "capture group: " << cap << std::endl;
        if (cap.matched){
            std::for_each(cap.first, cap.second, [](int v){
                cout << std::hex << v << " ";
            });
            cout << std::endl;
        }
    }
}
...

capture group: 192.168.178.21
31 39 32 2e 31 36 38 2e 31 37 38 2e 32 31

capture group: 192
31 39 32

capture group: 168
31 36 38

capture group: 178
31 37 38

capture group: 21
32 31

---


```

The regular expression `regEx` stands for an IPv4 address. `regEx` is used to extract the components of the address using capture groups. Finally, the capture groups and the characters in ASCII are displayed in hexadecimal values.

Match

`std::regex_match` determines if text matches a text pattern. You can further analyse the search result of type `std::match_results`.

The code snippet below shows three simple applications of `std::regex_match`: a C string, a C++ string and a range returning only a boolean. The three variants

are available for `std::match_results` objects respectively.

```
std::match
```

```
// match.cpp
...
#include <regex>
...

std::string numberRegEx(R"([-+]?([0-9]*\.[0-9]+|[0-9]+))");
std::regex rgx(numberRegEx);
const char* numChar{"2011"};

if (std::regex_match(numChar, rgx)){
    std::cout << numChar << " is a number." << std::endl;
} // 2011 is a number.

const std::string numStr{"3.14159265359"};
if (std::regex_match(numStr, rgx)){
    std::cout << numStr << " is a number." << std::endl;
} // 3.14159265359 is a number.

const std::vector<char> numVec{{'-'}, '2', '.', '7', '1', '8', '2',
                                '8', '1', '8', '2', '8'};
if (std::regex_match(numVec.begin(), numVec.end(), rgx)){
    for (auto c: numVec){ std::cout << c ;};
    std::cout << " is a number." << std::endl;
} // -2.718281828 is a number.
```

Search

`std::regex_search` checks if text contains a text pattern. You can use the function with and without a `std::match_results` object and apply it to a C string, a C++ string or a range.

The example below shows how to use `std::regex_search` with texts of type `const char*`, `std::string`, `const wchar_t*` and `std::wstring`.

```
std::search
```

```
// search.cpp
...
#include <regex>
...

// regular expression holder for time
std::regex crgx("[01]?[0-9]|2[0-3]):[0-5][0-9]");

// const char*
std::cmatch cmatch;

const char* ctime{"Now it is 23:10."};
if (std::regex_search(ctime, cmatch, crgx)){
    std::cout << ctime << std::endl; // Now it is 23:10.
    std::cout << "Time: " << cmatch[0] << std::endl; // Time: 23:10
}

// std::string
std::smatch smatch;
```

```

std::string stime{"Now it is 23:25."};
if (std::regex_search(stime, smatch, crgx)){
    std::cout << stime << std::endl;           // Now it is 23:25.
    std::cout << "Time: " << smatch[0] << std::endl; // Time: 23:25
}

// regular expression holder for time
std::wregex wrgx(L"([01]?[0-9]|2[0-3]):[0-5][0-9]");

// const wchar_t*
std::wcmatch wcmatch;

const wchar_t* wctime{L"Now it is 23:47."};
if (std::regex_search(wctime, wcmatch, wrgx)){
    std::wcout << wctime << std::endl;           // Now it is 23:47.
    std::wcout << "Time: " << wcmatch[0] << std::endl; // Time: 23:47
}

// std::wstring
std::wsmatch wsmatch;

std::wstring wstime{L"Now it is 00:03."};
if (std::regex_search(wstime, wsmatch, wrgx)){
    std::wcout << wstime << std::endl;           // Now it is 00:03.
    std::wcout << "Time: " << wsmatch[0] << std::endl; // Time: 00:03
}

```

Replace

`std::regex_replace` replaces sequences in a text matching a text pattern. It returns in the simple form `std::regex_replace(text, regex, replString)` its result as string. The function replaces an occurrence of `regex` in `text` with `replString`.

```

std::replace


---


// replace.cpp
...
#include <regex>
...
using namespace std;

string future{"Future"};
string unofficialName{"The unofficial name of the new C++ standard is C++0x."};

regex rgxCpp{R"(C\+\+0x)"};
string newCppName{"C++11"};
string newName{regex_replace(unofficialName, rgxCpp, newCppName)};

regex rgxOff{"unofficial"};
string makeOfficial{"official"};
string officialName{regex_replace(newName, rgxOff, makeOfficial)};

cout << officialName << endl;
// The official name of the new C++ standard is C++11.

```

In addition to the simple version, C++ has a version of `std::regex_replace` working on ranges. It enables you to push the modified string directly into another string:

```
typedef basic_regex<char> regex;
std::string str2;
std::regex_replace(std::back_inserter(str2),
                  text.begin(), text.end(), regex, replString);
```

All variants of `std::regex_replace` have an additional optional parameter. If you set the parameter to `std::regex_constants::format_no_copy` you will get the part of the text matching the regular expression, the unmatched text is not copied. If you set the parameter to `std::regex_constants::format_first_only` `std::regex_replace` will only be applied once.

Format

`std::regex_replace` and `std::match_results::format` in combination with capture groups enables you to format text. You can use a format string together with a placeholder to insert the value.

Here are both possibilities:

Formatting with regex

```
// format.cpp
...
#include <regex>
...

std::string future{"Future"};
const std::string unofficial{"unofficial, C++0x"};
const std::string official{"official, C++11"};

std::regex regValues{"(.*),(.*)"};
std::string standardText{"The $1 name of the new C++ standard is $2."};
std::string textNow= std::regex_replace(unofficial, regValues, standardText);
std::cout << textNow << std::endl;
// The unofficial name of the new C++ standard is C++0x.

std::smatch smatch;
if (std::regex_match(official, smatch, regValues)){
    std::cout << smatch.str();                                // official, C++11
    std::string textFuture= smatch.format(standardText);
    std::cout << textFuture << std::endl;
}                                                               // The official name of the new C++ standard is C++11.
```

In the function call `std::regex_replace(unofficial, regValues, standardText)` the text matching the first and second capture group of the

regular expression `regValues` is extracted from the string `unofficial`. The placeholders `$1` and `$2` in the text `standardText` are then replaced by the extracted values. The strategy of `smatch.format(standardTest)` is similar but there is a difference:

The creation of the search results `smatch` is separated from their usage when formatting the string.

In addition to capture groups, C++ supports additional format escape sequences. You can use them in format strings:

Format escape sequence	Format escape sequences
<hr/>	
<code>\$&</code>	Returns the total match (0th capture group).
<code>\$\$</code>	Returns <code>\$</code> .
<code>\$`</code> (backward tic)	Returns the text before the total match.
<code>\$'`</code> (forward tic)	Returns the text after the total match.
<code>\$i</code>	Returns the <code>i</code> th capture group.

Repeated Search

It's quite convenient to iterate with `std::regex_iterator` and `std::regex_token_iterator` over the matched texts. `std::regex_iterator` supports the matches and their capture groups. `std::regex_token_iterator` supports more. You can address the components of each capture and by using a negative index, you can access the text between the matches.

`std::regex_iterator`

C++ defines the following four type synonyms for `std::regex_iterator`.

```
typedef cregex_iterator    regex_iterator<const char*>
typedef wcregex_iterator  regex_iterator<const wchar_t*>
typedef sregex_iterator   regex_iterator<std::string::const_iterator>
typedef wsregex_iterator regex_iterator<std::wstring::const_iterator>
```

You can use `std::regex_iterator` to count the occurrences of the words in a text:

```
std::regex_iterator  
// regexIterator.cpp  
...  
#include <regex>  
...  
using std::cout;  
std::string text{"That's a (to me) amazingly frequent question. It may be the most frequently asked question. Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever."};  
  
std::regex wordReg{R"(\w+)"};  
std::sregex_iterator wordItBegin(text.begin(), text.end(), wordReg);  
const std::sregex_iterator wordItEnd;  
std::unordered_map<std::string, std::size_t> allWords;  
for (; wordItBegin != wordItEnd; ++wordItBegin){  
    ++allWords[wordItBegin->str()];  
}  
for (auto wordIt: allWords) cout << "(" << wordIt.first << ":"  
    << wordIt.second << ")";  
    // (as:2)(of:1)(level:1)(find:1)(ever:1)(and:2)(natural:1) ...
```

A word consists of at least one character (`\w+`). This regular expression is used to define the begin iterator `wordItBegin` and the end iterator `wordItEnd`. The iteration through the matches happens in the for loop. Each word increments the counter: `++allWords[wordItBegin->str()]`. A word with counter equals to 1 is created if it is not already in `allWords`.

std::regex_token_iterator

C++ defines the following four type synonyms for `std::regex_token_iterator`.

typedef cregex_iterator	<code>regex_iterator<const char*></code>
typedef wcregex_iterator	<code>regex_iterator<const wchar_t*></code>
typedef sregex_iterator	<code>regex_iterator<std::string::const_iterator></code>
typedef wsregex_iterator	<code>regex_iterator<std::wstring::const_iterator></code>

`std::regex_token_iterator` enables you by using indexes to specify which capture groups you are interested in explicitly. If you don't specify the index, you will get all capture groups, but you can also request specific capture groups using its respective index. The -1 index is special: You can use -1 to address the text between the matches:

```
std::regex_token_iterator  
// tokenIterator.cpp  
...
```

```

using namespace std;

string text = "Pete Becker,The C++ Standard Library Extensions,2006:"
    + "Nicolai Josuttis,The C++ Standard Library,1999";

regex regBook(R"((\w+)\s(\w+),([\w\s\+]*) ,(\d{4}))");
sregex_token_iterator bookItBegin(text.begin(), text.end(), regBook);

const sregex_token_iterator bookItEnd;
while (bookItBegin != bookItEnd){
    cout << *bookItBegin++ << endl;
}                                // Pete Becker,The C++ Standard Library Extensions,2006
                                // Nicolai Josuttis,The C++ Standard Library,1999

sregex_token_iterator bookItNameIssueBegin(text.begin(), text.end(),
                                         regBook, {{2,4}});
const sregex_token_iterator bookItNameIssueEnd;

while (bookItNameIssueBegin != bookItNameIssueEnd){
    cout << *bookItNameIssueBegin++ << ", ";
    cout << *bookItNameIssueBegin++ << endl;
}                                // Becker, 1999
                                // Josuttis, 2001

regex regBookNeg(":");
sregex_token_iterator bookItNegBegin(text.begin(), text.end(), regBookNeg, -1);

const sregex_token_iterator bookItNegEnd;
while (bookItNegBegin != bookItNegEnd){
    cout << *bookItNegBegin++ << endl;
}                                // Pete Becker,The C++ Standard Library Extensions,2006
                                // Nicolai Josuttis,The C++ Standard Library,1999

```

bookItBegin using no indices and bookItNegbegin using the negative index returns both the total capture group, but bookNameIssueBegin only the second and fourth capture group {{2,4}}.

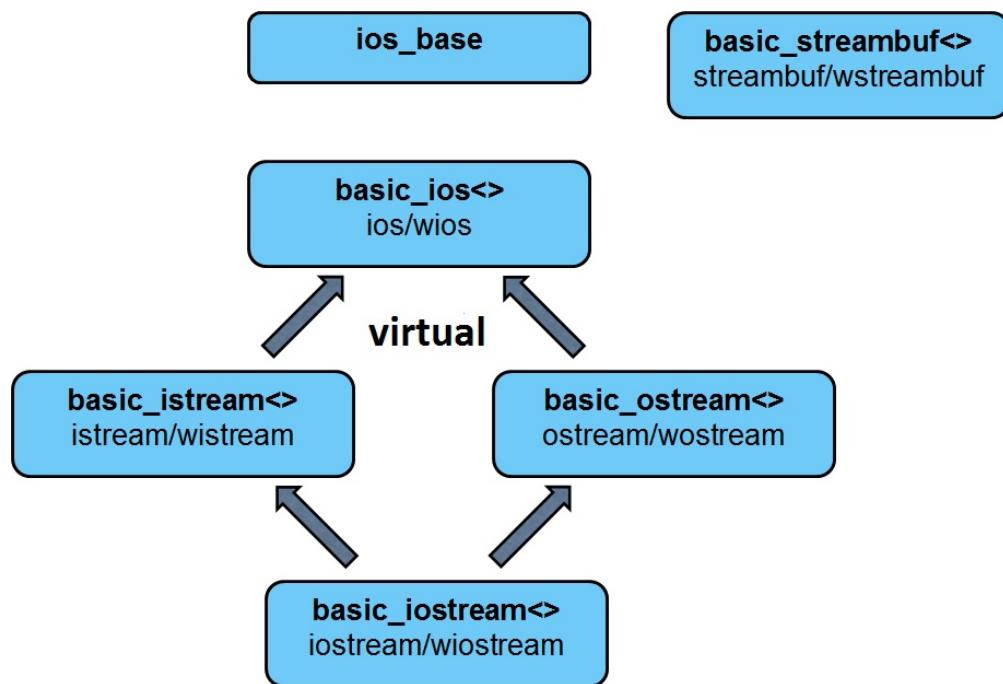
16. Input and Output Streams

The [input and output streams](#) enable you to communicate with the outside world. A stream is an infinite character stream on which you can push or pull data. Push is called writing, pull is called reading.

The input and output streams

- were used long before the first C++ standard (C++98) in 1998,
- are a for the extensibility designed framework,
- are implemented according to the object-oriented and generic paradigms.

Hierarchy



`basic_streambuf<>`

Reads and writes the data.

`ios_base`

Properties of all stream classes independent on the character type.

`basic_ios<>`

Properties of all stream classes dependent of the character type.

`basic_istream<>`

Base for the stream classes for the reading of the data.

`basic_ostream<>`

Base for the stream classes for the writing of the data.

`basic_iostream<>`

Base for the stream classes for the reading and writing of the data.

The class hierarchy has type synonyms for the character types `char` and `wchar_t`. Names not starting with `w` are type synonyms for `char`, names starting with `w` for `wchar_t`.

The base classes of the class `std::basic_iostream<>` are virtually derived from `std::basic_ios<>`, therefore `std::basic_iostream<>` has only one instance of `std::basic_ios`.

Input and Output Functions

The stream classes `std::istream` and `std::ostream` are often used for the reading and writing of data. Use of `std::istream` classes requires the `<iostream>` header; use of `std::ostream` classes requires the `<ostream>` header. You can have both with the header `<iostream>`. `std::istream` is a `typedef` for the class `basic_istream` and the character type `char`, `std::ostream` for the class `basic_ostream` respectively:

```
typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;
```

C++ has four predefined stream objects for the convenient dealing with the keyboard and the monitor.

The four predefined stream objects				
<u>Stream object</u>	<u>C pendant</u>	<u>Device</u>	<u>Buffered</u>	
<code>std::cin</code>	<code>stdin</code>	keyboard	yes	
<code>std::cout</code>	<code>stdout</code>	monitor	yes	
<code>std::cerr</code>	<code>stderr</code>	monitor	no	

std::clog	monitor	yes
-----------	---------	-----



The stream objects are also available for wchar_t

The four stream objects for wchar_t std::wcin, std::wcout, std::wcerr and std::wclog are by far not so heavily used as their char pendants. Therefore I treat them only marginally.

The stream objects are sufficient to write a program that reads from the command line and returns the sum.

The stream objects

```
// IOSTreams.cpp
...
#include <iostream>

int main(){
    std::cout << "Type in your numbers";
    std::cout << "(Quit with an arbitrary character): " << std::endl;
    // 2000 <Enter> 11 <a>
    int sum{0};
    int val;
    while (std::cin >> val) sum += val;
    std::cout << "Sum: " << sum;                                // Sum: 2011
}
```

The small program above uses the stream operators << and >> and the stream manipulator std::endl.

The insert operator << pushes characters onto the output stream std::cout; the extract operator >> pulls the characters from the input stream std::cin. You can build chains of insert or extract operators because both operators return a reference to themselves.

std::endl is a stream manipulator because it puts a '\n' character onto std::cout and flushes the output buffer.

Here are the most frequently used stream manipulators.

The most frequently used stream manipulators

Manipulator	Stream type	Description
-------------	-------------	-------------

<code>std::endl</code>	output	Inserts a new-line character and flushes the stream.
<code>std::flush</code>	output	Flushes the stream.
<code>std::ws</code>	input	Discards leading whitespace.

Input

You can read in C++ in two way from the input stream: Formatted with the extractor `>>` and unformatted with explicit methods.

Formatted Input

The extraction operator `>>`

- is predefined for all built-in types and strings,
- can be implemented for [user-defined data types](#),
- can be configured by [format specifiers](#).



`std::cin` ignores by default leading whitespace

```
#include <iostream>
...
int a, b;
std::cout << "Two natural numbers: " << std::endl;
std::cin >> a >> b; // < 2000 11>
std::cout << "a: " << a << " b: " << b;
```

Unformatted Input

There are many methods for the unformatted input from an input stream `is`.

Unformatted input from an input stream

Method	Description
<code>is.get(ch)</code>	Reads one character into <code>ch</code> .

<code>is.get(buf, num)</code>	Reads at most <code>num</code> characters into the buffer <code>buf</code> .
<code>is.getline(buf, num[, delim])</code>	Reads at most <code>num</code> characters into the buffer <code>buf</code> . Uses optionally the line-delimiter <code>delim</code> (default <code>\n</code>).
<code>is.gcount()</code>	Returns the number of characters that were last extracted from <code>is</code> by an unformatted operation.
<code>is.ignore(streamsize sz=1, int delim= end-of-file)</code>	Ignores <code>sz</code> characters until <code>delim</code> .
<code>is.peek()</code>	Gets one characters from <code>is</code> without consuming it.
<code>is.unget()</code>	Pushes the last read character back to <code>is</code> .
<code>is.putback(ch)</code>	Pushes the character <code>ch</code> onto the stream <code>is</code> .



std::string has a getline function

The [getline function](#) of `std::string` has a big advantage above the `getline` function of the `istream`. The `std::string` automatically takes care of its memory. On the contrary, you have to reserve the memory for the buffer `buf` in the `is.get(buf, num)` function.

```
// inputUnformatted.cpp
...
#include <iostream>
...
std::string line;
std::cout << "Write a line: " << std::endl;

std::getline(std::cin, line);           // <Only for testing purpose.>
std::cout << line << std::endl;       // Only for testing purpose.

std::cout << "Write numbers, separated by;" << std::endl;
while (std::getline(std::cin, line, ';') ) {
    std::cout << line << " ";
}
// <2000;11;a>
// 2000 11
```

Output

You can push characters with the insert operator `<<` onto the output stream.

The insert operator `<<`

- is predefined for all built-in types and strings,
- can be implemented for [user-defined data types](#),
- can be adjusted by [format specifiers](#).

Format Specifier

Format specifiers enable you to adjust the input and output data explicitly.



I use manipulators as format specifiers

The format specifiers are available as manipulators and flags. I only present manipulators in this book because their functionality is quite similar and manipulators are more comfortable to use.

Manipulators as format specifiers

```
// formatSpecifier.cpp
...
#include <iostream>
...
int num{2011};

std::cout.setf(std::ios::hex, std::ios::basefield);
std::cout << num << std::endl; // 7db
std::cout.setf(std::ios::dec, std::ios::basefield);
std::cout << num << std::endl; // 2011

std::cout << std::hex << num << std::endl; // 7db
std::cout << std::dec << num << std::endl; // 2011
```

The followings tables present the important format specifiers. The format specifiers are sticky except for the field width, which is reset after each application.

The manipulators without any arguments require the header `<iostream>`, and the manipulators with arguments require the header `<iomanip>`.

Manipulator	Stream type	Displaying of boolean values	Description
-------------	-------------	------------------------------	-------------

std::boolalpha	input and output	Displays the boolean as a word.
----------------	------------------	---------------------------------

std::noboolalpha	input and output	Displays the boolean as number (default).
------------------	------------------	---

Manipulator	Stream type	Description
-------------	-------------	-------------

std::setw(val)	input and output	Sets the field width to val.
----------------	------------------	------------------------------

std::setfill(c)	output stream	Sets the fill character to c (default: spaces).
-----------------	---------------	---

Alignment of the text

Manipulator	Stream type	Description
-------------	-------------	-------------

std::left	output	Aligns the output left.
-----------	--------	-------------------------

std::right	output	Aligns the output right.
------------	--------	--------------------------

std::internal	output	Aligns the signs of numbers left, the values right.
---------------	--------	---

Positive signs and upper or lower case

Manipulator	Stream type	Description
-------------	-------------	-------------

std::showpos	output	Displays positive signs.
--------------	--------	--------------------------

std::noshowpos	output	Doesn't display positive signs (default).
----------------	--------	---

std::uppercase	output	Uses upper case characters for numeric values (default).
----------------	--------	--

`std::lowercase` output Uses lower case characters for numeric values.

Manipulator	Stream type	Description
<code>std::oct</code>	input and output	Uses natural numbers in octal format.
<code>std::dec</code>	input and output	Uses natural numbers in decimal format (default).
<code>std::hex</code>	input and output	Uses natural numbers in hexadecimal format.
<code>std::showbase</code>	output	Displays the numeric base.
<code>std::noshowbase</code>	output	Doesn't display the numeric base (default).

There are special rules for floating point numbers:

- The number of significant digits (digits after the comma) is by default 6.
- If the number of significant digits is not big enough the numbers is displayed in scientific notation.
- Leading and trailing zeros are not be displayed.
- If possible the decimal point are not be displayed.

Manipulator	Stream type	Description
<code>std::setprecision(val)</code>	output	Adjusts the precision of the output to <code>val</code> .
<code>std::showpoint</code>	output	Displays the decimal point.
<code>std::noshowpoint</code>	output	Doesn't display the decimal point (default).

Displays the floating point number in

<code>std::fixed</code>	output	decimal format.
<code>std::scientific</code>	output	Displays the floating point number in scientific format.
<code>std::hexfloat</code>	output	Displays the floating-point number in hexadecimal format.
<code>std::defaultfloat</code>	output	Displays the floating-point number in default floating-point notation.

Format specifier

```
// formatSpecifierOutput.cpp
...
#include <iomanip>
#include <iostream>
...

std::cout.fill('#');
std::cout << -12345;
std::cout << std::setw(10) << -12345;           // #####-12345
std::cout << std::setw(10) << std::left << -12345;   // -12345#####
std::cout << std::setw(10) << std::right << -12345;  // #####-12345
std::cout << std::setw(10) << std::internal << -12345; // -#####12345

std::cout << std::oct << 2011; // 3733
std::cout << std::hex << 2011; // 7db

std::cout << std::showbase;
std::cout << std::dec << 2011; // 2011
std::cout << std::oct << 2011; // 03733
std::cout << std::hex << 2011; // 0x7db

std::cout << 123.456789;           // 123.457
std::cout << std::fixed;
std::cout << std::setprecision(3) << 123.456789; // 123.457
std::cout << std::setprecision(6) << 123.456789; // 123.456789
std::cout << std::setprecision(9) << 123.456789; // 123.456789000

std::cout << std::scientific;
std::cout << std::setprecision(3) << 123.456789; // 1.235e+02
std::cout << std::setprecision(6) << 123.456789; // 1.234568e+02
std::cout << std::setprecision(9) << 123.456789; // 1.234567890e+02

std::cout << std::hexfloat;
std::cout << std::setprecision(3) << 123.456789; // 0x1.edd3c07ee0b0bp+6
std::cout << std::setprecision(6) << 123.456789; // 0x1.edd3c07ee0b0bp+6
std::cout << std::setprecision(9) << 123.456789; // 0x1.edd3c07ee0b0bp+6

std::cout << std::defaultfloat;
std::cout << std::setprecision(3) << 123.456789; // 123
std::cout << std::setprecision(6) << 123.456789; // 123.457
std::cout << std::setprecision(9) << 123.456789; // 123.456789
```

Streams

A stream is an infinite data stream on which you can push or pull data. String streams and file streams enable strings and files to interact with the stream directly.

String Streams

String streams need the header <`sstream`>. They are not connected to an input or output stream and store their data in a string.

Whether you use a string stream for input or output or with the character type `char` or `wchar_t` there are various string stream classes:

`std::istringstream` and `std::wistringstream`

String stream for the input of data of type `char` and `wchar_t`.

`std::ostringstream` and `std::wostringstream`

String stream for the output of data of type `char` and `wchar_t`.

`std::stringstream` and `std::wstringstream`

String stream for the input or output of data of type `char` and `wchar_t`.

Typical operations on a string stream are:

- Write data in a string stream:

```
std::stringstream os;
os << "New String";
os.str("Another new String");
```

- Read data from a string stream:

```
std::string os;
std::string str;
os >> str;
str= os.str();
```

- Clear a string stream:

```
std::stringstream os;
os.str("");
```

String streams are often used for the type safe conversion between strings and numeric values:

```

// stringStreams.cpp
...
#include <sstream>
...

template <typename T>
T StringTo ( const std::string& source ){
    std::istringstream iss(source);
    T ret;
    iss >> ret;
    return ret;
}

template <typename T>
std::string ToString( const T& n){
    std::ostringstream tmp ;
    tmp << n;
    return tmp.str();
}

std::cout << "5= " << StringTo<int>("5");                                // 5
std::cout << "5 + 6= " << StringTo<int>("5") + 6;                          // 11
std::cout << ToString(StringTo<int>("5") + 6 );                           // "11"
std::cout << "5e10: " << std::fixed << StringTo<double>("5e10"); // 500000000000

```

File Streams

File streams enable you to work with files. They need the header `<fstream>`. The file streams automatically manage the lifetime of their file.

Whether you use a file stream for input or output or with the character type `char` or `wchar_t` there are various file stream classes:

`std::ifstream` and `std::wifstream`

File stream for the input of data of type `char` and `wchar_t`.

`std::ofstream` and `std::wofstream`

File stream for the output of data of type `char` and `wchar_t`.

`std::fstream` and `std::wfstream`

File stream for the input and output of data of type `char` and `wchar_t`.

`std::filebuf` and `std::wfilebuf`

Data buffer of type `char` and `wchar_t`.



Set the file position pointer

File streams used for reading and writing have to set the file position pointer after the contents change.

Flags enable you to set the opening mode of a file stream.

Flag	Description
<code>std::ios::in</code>	Opens the file stream for reading (default for <code>std::ifstream</code> and <code>std::wifstream</code>).
<code>std::ios::out</code>	Opens the file stream for writing (default for <code>std::ofstream</code> and <code>std::wofstream</code>).
<code>std::ios::app</code>	Appends the character to the end of the file stream.
<code>std::ios::ate</code>	Sets the initial position of the file position pointer on the end of the file stream.
<code>std::ios::trunc</code>	Deletes the original file.
<code>std::ios::binary</code>	Suppresses the interpretation of an escape sequence in the file stream.

It's quite easy to copy the file named `in` to the file named `out` with the file buffer `in.rdbuf()`. The error handling is missing in this short example.

```
#include <fstream>
...
std::ifstream in("inFile.txt");
std::ofstream out("outFile.txt");
out << in.rdbuf();
```

If you combine the C++ flags, you can compare the C++ and C modes to open a file.

Opening of a file with C++ and C

C++ mode	Description	C mode
<code>std::ios::in</code>	Reads the file.	"r"
<code>std::ios::out</code>	Writes the file.	"w"
<code>std::ios::out std::ios::app</code>	Appends to the file.	"a"
<code>std::ios::in std::ios::out</code>	Reads and writes the file.	"r+"
<code>std::ios::in std::ios::out std::ios::trunc</code>	Writes and reads the file.	"w+"

The file has to exist with the mode "r" and "r+". In contrary, the file is created with "a" and "w+". The file is overwritten with "w".

You can explicitly manage the lifetime of a file stream.

Managing the lifetime of a file stream

Flag	Description
<code>infile.open(name)</code>	Opens the file name for reading.
<code>infile.open(name, flags)</code>	Opens the file name with the flags flags for reading.
<code>infile.close()</code>	Closes the file name.
<code>infile.is_open()</code>	Checks if the file is open.

Random Access

Random access enables you to set the file position pointer arbitrarily.

When a file stream is constructed, the files position pointer points to the beginning of the file. You can adjust the position with the methods of the file

stream file.

Navigate in a file stream

Method	Description
<code>file.tellg()</code>	Returns the read position of <code>file</code> .
<code>file.tellp()</code>	Returns the write position of <code>file</code> .
<code>file.seekg(pos)</code>	Sets the read position of <code>file</code> to <code>pos</code> .
<code>file.seekp(pos)</code>	Sets the write position of <code>file</code> to <code>pos</code> .
<code>file.seekg(off, rpos)</code>	Sets the read position of <code>file</code> to the offset <code>off</code> relative to <code>rpos</code> .
<code>file.seekp(off, rpos)</code>	Sets the write position of <code>file</code> to the offset <code>off</code> relative to <code>rpos</code> .

`off` has to be a number. `rpos` can have three values:

`std::ios::beg`

Position at the beginning of the file.

`std::ios::cur`

Position at the current position.

`std::ios::end`

Position at the end of the file.



Respect the file boundaries

If you randomly access a file, the C++ runtime does not check the file boundaries. Reading or writing data outside the boundaries is *undefined behaviour*.

Random access

```
// randomAccess.cpp
...
#include <fstream>
...
```

```

void writeFile(const std::string name){
    std::ofstream outFile(name);
    if (!outFile){
        std::cerr << "Could not open file " << name << std::endl;
        exit(1);
    }
    for (unsigned int i= 0; i < 10 ; ++i){
        outFile << i << "      0123456789" << std::endl;
    }
}

std::string random{"random.txt"};
writeFile(random);
std::ifstream inFile(random);

if (!inFile){
    std::cerr << "Could not open file " << random << std::endl;
    exit(1);
}

std::string line;

std::cout << inFile.rdbuf();
// 0      0123456789
// 1      0123456789
...
// 9      0123456789

std::cout << inFile.tellg() << std::endl; // 200

inFile.seekg(0); // inFile.seekg(0, std::ios::beg);
getline(inFile, line); // 0      0123456789
std::cout << line;

inFile.seekg(20, std::ios::cur);
getline(inFile, line); // 2      0123456789
std::cout << line;

inFile.seekg(-20, std::ios::end);
getline(inFile, line); // 9      0123456789

```

State of the Stream

Flags represent the state of the stream stream. The methods for dealing with these flags need the header <iostream>.

Flag	State of a stream	Query of the flag	Description
std::ios::goodbit	stream.good()	No bit set.	
std::ios::eofbit	stream.eof()	end-of-file bit set.	

```
std::ios::failbit stream.fail() Error.
```

```
std::ios::badbit stream.bad() Undefined behaviour.
```

Here are examples for conditions causing the different states of a stream:

`std::ios::eofbit`

- Reading beyond the last valid character.

`std::ios::failbit`

- False formatted reading.
- Reading beyond the last valid character.
- Opening of a file went wrong.

`std::ios::badbit`

- Size of the stream buffer cannot be adjusted.
- Code conversion of the stream buffer went wrong.
- A part of the stream threw an exception.

`stream.fail()` returns whether `std::ios::failbit` or `std::ios::badbit` is set.

The state of a stream can be read and set.

`stream.clear()`

Initializes the flags and sets the stream in the goodbit state.

`stream.clear(sta)`

Initializes the flags and set the stream into sta state.

`stream.rdbuf()`

Returns the current state.

`stream.setstate(fla)`

Sets the additional flag fla.

Operations on a stream only work if the stream is in the goodbit state. If the stream is in the badbit state you cannot set it to goodbit state.

State of a stream

```
// streamState.cpp
```

```
...
```

```
#include <iostream>
```

```

...
std::cout << std::cin.fail() << std::endl;      // false

int myInt;

while (std::cin >> myInt){                      // <a>
    std::cout << myInt << std::endl;              //
    std::cout << std::cin.fail() << std::endl;      //
}

std::cin.clear();
std::cout << std::cin.fail() << std::endl;      // false

```

The input of the character a causes the stream `std::cin` to be in `std::ios::failbit` state. Therefore a and `std::cin.fail()` cannot be displayed. At first, you have to initialize the stream `std::cin`.

User-defined Data Types

If you overload the input and output operators, your data type behaves like a built-in data type.

```

friend std::istream& operator>> (std::istream& in, Fraction& frac);
friend std::ostream& operator<< (std::ostream& out, const Fraction& frac);

```

For overloading the input and output operators you have to keep a few rules in mind:

- To support the chaining of input and output operations you have to get and return the input and output streams by non-constant reference.
- To access the private members of the class, the input and output operators have to be *friends* of your data type.
- The input operator `>>` takes its data type as a non-constant reference.
- The output operator `<<` takes its data type as a constant reference.

Overloading input and output operator

```

// overloadingInOutput.cpp
class Fraction{
public:
    Fraction(int num= 0, int denom= 0):numerator(num), denominator(denom){}
    friend std::istream& operator>> (std::istream& in, Fraction& frac);
    friend std::ostream& operator<< (std::ostream& out, const Fraction& frac);
private:
    int numerator;
    int denominator;
};

std::istream& operator>> (std::istream& in, Fraction& frac){
    in >> frac.numerator;
    in >> frac.denominator;
    return in;
}

```

```
}

std::ostream& operator<< (std::ostream& out, const Fraction& frac){
    out << frac.numerator << "/" << frac.denominator;
    return out;
}

Fraction frac(3, 4);
std::cout << frac;                                // 3/4

std::cout << "Enter two numbers: ";
Fraction fracDef;

std::cin >> fracDef;                            // <1 2>
std::cout << fracDef;                           // 1/2
```

17. Formatting Library

The [formatting library](#) provides a safe and extensible alternative to the `printf` family and extends the [Input and Output streams](#). The library requires the `<format>` and the format string syntax follows the Python syntax.

Thanks to the formatting library, you can specify format specifications for

- the fill character and alignment of text.
- the sign, the width, and the precision of numbers.
- the data type.

Formatting Functions

The formatting library has three formatting functions.

`std::format`

Returns the formatted string

`std::format_to`

Writes the formatted string via an output iterator

`std::format_to_n`

Writes the formatted string via an output iterator but not more than `n` characters

```
std::format("{1} {0}!", "world", "Hello"); // Hello world!  
  
std::string buffer;  
std::format_to(std::back_inserter(buffer),  
    "Hello, C++{}!\n", "20"); // Hello, C++20!
```



I use `std::format` for the rest of this chapter

The formatting functions `std::format`, `std::format_to`, and `std::format_to_n` follow the same syntax. Most of the time, you use `std::format` and so do I for the rest of this chapter.

Syntax

`std::format` has the following 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

- Beginning character {
 - Argument-ID (optional)
 - Colon : followed by a format specification (optional)
- Closing character }

The argument-id allows you to specify the index of the arguments in **Args**. The id's start with 0. When you don't provide the argument-id, the arguments are used as given. Either all replacement fields have to use an argument-id or none.

`std::formatter` and its specializations define the **format specification** for the arguments.

- basic types and string types: [standard format specification](#) based on [Python's format specification](#).
- chrono types: [chrono format specification](#)
- other types: user-defined formatter specification

Format specification

You can specify the fill character and alignment of the text, the sign, the width, and the precision of numbers, and the data type.

Fill character and alignment of text

- Fill character: per default space
- Alignment:
 - <: left
 - >: right

- ^: centered

```
char c = 120;

std::format("{:7}", 42);      // "      42"
std::format("{:7}", 'x');     // "x      "
std::format("{:<7}", 'x');   // "x*****"
std::format("{:>7}", 'x');   // "*****x"
std::format("{:^7}", 'x');    // "***x***"
std::format("{:7d}", c);     // "      120"
std::format("{:7}", true);   // "true   "
```

Sign, width, and precision of numbers

- Sign

- +: number gets a sign symbol
- -: negative numbers get a sign (default)
- space: positive numbers get a space

```
double inf = std::numeric_limits<double>::infinity();
double nan = std::numeric_limits<double>::quiet_NaN();

std::format("{0:},{0:+},{0:-},{0: }", 1); // "1,+1,1, 1"
std::format("{0:},{0:+},{0:-},{0: }", -1); // "-1,-1,-1,-1"
std::format("{0:},{0:+},{0:-},{0: }", inf); // "inf,+inf,inf, inf"
std::format("{0:},{0:+},{0:-},{0: }", nan); // "nan,+nan,nan, nan"
```

- #:

- uses the alternative format
- integrals:
 - binary numbers: 0b
 - octal numbers: 0
 - hexadecimal numbers: 0x

- 0: fills with leading zeros

```
std::format("{:+06d}", 120); // "+00120"
std::format("{:#0f}", 120)); // "120.000000"
std::format("{:0>15f}", 120)); // "000000120.000000"
std::format("{:#06x}", 0xa); // "0x000a"
```

- Width: specifies the minimal width
- Precision: can be applied to floating pointer numbers and strings
 - Floating-point number: place after the decimal point
 - Strings: maximal number of characters

```
double d = 20.11;
std::format("{}", d)); // " 20.11 "
```

```
std::format("{:10}", d));           // " 20.11 "
std::format("{:10.3}", d));          // " 20.1 "
std::format("{:.3}", "Only a test."); // " Onl "
```

Data types

The values are copied to the output if not other specified. You can explicitly specify the presentation of a value:

- String presentation: s
- Integer presentation:
 - b, B: binary
 - d: decimal
 - o: octal
 - x, X: hexadecimal
- Character presentation:
 - b, B, d, o, x, X: integer
- Bool:
 - s: true or false
 - b, B, d, o, x, X: integer
- Floating point:
 - e, E: scientific
 - f, F: decimal

User-defined formatter

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

- **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.

Formatting a std::vector

The specialisation of the class std::formatter is as straightforward as possible. I specify a format specification, which is used for each element of the container.

```
// formatVector.cpp

#include <format>

...

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

    std::string formatString;

    auto constexpr parse(format_parse_context& ctx) {           // (3)
        formatString = "{:";

        std::string parseContext(std::begin(ctx), std::end(ctx));
        formatString += parseContext;
        return std::end(ctx) - 1;
    }

    template <typename FormatContext>
    auto format(const std::vector<T>& v, FormatContext& ctx) {
        auto out= ctx.out();
        std::format_to(out, "[");
        if (v.size() > 0) std::format_to(out, formatString, v[0]);
        for (int i= 1; i < v.size(); ++i) {
            std::format_to(out, ", " + formatString, v[i]);           // (1)
        }
        std::format_to(out, "]");                                     // (2)
        return std::format_to(out, "\n");
    }
};

...

std::vector<int> myInts{1, 2, 3, 4, 5};
std::cout << std::format("{:}", myInts);      // [1, 2 ,3 ,4, 5]          // (4)
std::cout << std::format("{:+}", myInts);     // [+1, +2, +3, +4, +5]
std::cout << std::format("{:03d}", myInts);   // [001, 002, 003, 004, 005]
std::cout << std::format("{:b}", myInts);     // [1, 10, 11, 100, 101] // (5)

std::vector<std::string> myStrings{"Only", "for", "testing"};
std::cout << std::format("{:}", myStrings);   // [Only, for, testing]
std::cout << std::format("{:.3}", myStrings); // [Onl, for, tes]
```

The specialization for std::vector has the member functions `parse` and `format`. `parse` essentially creates the `formatString` which is applied to each element of the `std::vector` (1 and 2). The `parse` context `ctx` (3) contains the characters between the colon (`:`) and the closing curly braces (`}`). On end, the function returns an iterator to the closing curly braces (`}`). 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`, the elements of a `std::vector` are nicely displayed.

The elements of the `std::vector` are formatted in a few ways. The first line (4) displays the numbers. The following line writes a sign before each number, aligns them to 3 characters and uses the `0` as a fill character. Line (5) displays them in binary format. The remaining two lines output each string of the `std::vector`. The last line truncates each string to three characters.

18. Filesystem

The new [filesystem](#) is based on [boost::filesystem](#). Some of its components are optional. This means not all functionality of std::filesystem is available on each implementation of the filesystem library. For example, FAT-32 does not support symbolic links.

The library is based on the three concepts: file, file name and path.

- A *file* is an object that holds data such that you can write to it or read from it. A file has a name and a file type. A file type can be a directory, hard link, symbolic link or a regular file.
 - A *directory* is a container for holding other files. The current directory is represented by a dot ". "; the parent directory is represented by two dots "..".
 - A *hard link* associates a name with an existing file.
 - A *symbolic link* associates a name with a path that may exist.
 - A *regular file* is a directory entry which is neither a directory, a hard link, nor a symbolic link.
- A *file name* is a string that represents a file. It is implementation-defined which characters are allowed, how long the name could be or if the name is case sensitive.
- A *path* is a sequence of entries that identifies the location for a file. It has an optional root-name such a “C:” on Windows, followed by a root-directory such a “/” on Unix. Additional parts can be directories, hard links, symbolic links, or regular files. A path can be absolute, canonical, or relative.
 - An *absolute path* is a path that identifies a file.
 - A *canonical path* is a path that includes neither a symbolic link nor the relative paths ". ." (current directory) or "..."(parent directory).
 - A *relative path* specifies a path relative to a location in the file system. Paths such as ". ." (current directory), "..."(parent directory) or "home/rainer" are relative paths. On Unix, they do not start at the root-directory "/".

Here is an introductory example to the filesystem.

Overview to the filesystem library

```
// filesystem.cpp
...
#include <filesystem>
...

namespace fs = std::filesystem;

std::cout << "Current path: " << fs::current_path() << std::endl; // (1)

std::string dir= "sandbox/a/b";
fs::create_directories(dir); // (2)

std::ofstream("sandbox/file1.txt");
fs::path symPath= fs::current_path() /= "sandbox"; // (3)
symPath /= "syma";
fs::create_symlink("a", "symPath"); // (4)

std::cout << "fs::is_directory(dir): " << fs::is_directory(dir) << std::endl;
std::cout << "fs::exists(symPath): " << fs::exists(symPath) << std::endl;
std::cout << "fs::symlink(symPath): " << fs::is_symlink(symPath) << std::endl;

for(auto& p: fs::recursive_directory_iterator("sandbox")){ // (5)
    std::cout << p << std::endl;
}
fs::remove_all("sandbox");
```

`fs::current_path()` (1) returns the current path. You can create a directory hierarchy (2) with `std::filesystem::createDirectories`. The `/=` operator is overloaded for a path (3). Therefore, I can directly create a symbolic link (4) or check the properties of a file. The call `recursive_directory_iterator` (5) allows you to traverse directories recursively.

Output:

```
Current path: "/tmp/1469540273.75652"

fs::is_directory(dir): true
fs::exists(symPath): true
fs::symlink(symPath): true

"sandbox/syma"
"sandbox/file1.txt"
"sandbox/a"
"sandbox/a/b"
"sandbox/a/b/c"
```

Classes

There are many classes encapsulating a specific aspect of the filesystem.

Class	The various classes the filesystem	Description
path		Represents a path.
filesystem_error		Defines an exception object.
directory_entry		Represents a directory entry.
directory_iterator		Defines a directory iterator.
recursive_directory_iterator		Defines a recursive directory iterator.
file_status		Stores information about the file.
space_info		Represents filesystem information.
file_type		Indicates the type of a file.
perms		Represents file access permissions.
perm_options		Represents options for the function permissions .
copy_options		Represents options for the functions copy and copy_file .
directory_options		Represents options for the functions directory_iterator and recursive_directory_iterator .
file_time_type		Represents file time.

Manipulating the permissions of a file

The permissions for a file are represented by the class `std::filesystem::perms`. It is a [BitmaskType](#) and can, therefore, be manipulated by bitwise operations. The access permissions are based on [POSIX](#).

The program from [en.cppreference.com](#) shows how you can read and manipulate the owner, group, and other (world) bits of a file.

Permissions of a file

```
// perms.cpp
...
#include <filesystem>
...

namespace fs = std::filesystem;

void printPerms(fs::perms perm){
    std::cout << ((perm & fs::perms::owner_read) != fs::perms::none ? "r" : "-")
        << ((perm & fs::perms::owner_write) != fs::perms::none ? "w" : "-")
        << ((perm & fs::perms::owner_exec) != fs::perms::none ? "x" : "-")
        << ((perm & fs::perms::group_read) != fs::perms::none ? "r" : "-")
        << ((perm & fs::perms::group_write) != fs::perms::none ? "w" : "-")
        << ((perm & fs::perms::group_exec) != fs::perms::none ? "x" : "-")
        << ((perm & fs::perms::others_read) != fs::perms::none ? "r" : "-")
        << ((perm & fs::perms::others_write) != fs::perms::none ? "w" : "-")
        << ((perm & fs::perms::others_exec) != fs::perms::none ? "x" : "-")
        << std::endl;
}

...
std::ofstream("rainer.txt");

std::cout << "Initial file permissions for a file: ";
printPerms(fs::status("rainer.txt").permissions()); // (1)

fs::permissions("rainer.txt", fs::perms::add_perms |
    fs::perms::owner_all | fs::perms::group_all); // (2)
std::cout << "Adding all bits to owner and group: ";
printPerms(fs::status("rainer.txt").permissions());

fs::permissions("rainer.txt", fs::perms::remove_perms | // (3)
    fs::perms::owner_write | fs::perms::group_write | fs::perms::others_write);
std::cout << "Removing the write bits for all: ";
printPerms(fs::status("rainer.txt").permissions());
```

Thanks to the call `fs::status("rainer.txt").permissions()`, I get the permissions of the file `rainer.txt` and can display them in the function `printPerms` (1). By setting the type `std::filesystem::add_perms`, I can add permissions to the owner and the group of the file (2). Alternatively, I can set the constant `std::filesystem::remove_perms` for removing permissions (3).

```
Initial file permissions for a file: rw-r--r--
Adding all bits to owner and group: rwxrwxr--
Removing the write bits for all:      r-xr-xr--
```

Non-member functions

Many non-member functions exist for manipulating the filesystem.

The non-member functions for manipulating the filesystem

Non-member functions	Description
absolute	Composes an absolute path.
canonical and weakly_canonical	Composes a canonical path.
relative and proximate	Composes a relative path.
copy	Copies files or directories.
copy_file	Copies file contents.
copy_symlink	Copies a symbolic link.
create_directory and create_directories	Creates a new directory.
create_hard_link	Creates a hard link.
create_symlink and create_directory_symlink	Creates a symbolic link.
current_path	Returns the current working directory.
exists	Checks if path refers to an existing file.
equivalent	Checks if two paths refer to the same file.

<code>file_size</code>	Returns the size of the file.
<code>hard_link_count</code>	Returns the number of hard links to a file.
<code>last_write_time</code>	Gets and sets the time of the last file modification.
<code>permissions</code>	Modifies the file access permissions.
<code>read_symlink</code>	Gets the target of the symbolic link.
<code>remove</code>	Removes a file or an empty directory.
<code>remove_all</code>	Removes a file or a directory with all its content recursively.
<code>rename</code>	Moves or renames a file or directory.
<code>resize_file</code>	Changes the size of a file by truncation.
<code>space</code>	Returns the free space on the filesystem.
<code>status</code>	Determines the file attributes.
<code>symlink_status</code>	Determines the file attributes and checks the symlink target.
<code>temp_directory_path</code>	Returns a directory for temporary files.

Read and set the last write time of a file

Thanks to the global function `std::filesystem::last_write_time`, you can read and set the last write time of a file. Here is an example, based on the `last_write_time` example from en.cppreference.com.

Write time of a file

```

// fileTime.cpp
...
#include <filesystem>
...

namespace fs = std::filesystem;
using namespace std::chrono_literals;

...

fs::path path = fs::current_path() / "rainer.txt";
std::ofstream(path.c_str());
auto ftime = fs::last_write_time(path); // (1)

std::time_t cftime = std::chrono::system_clock::to_time_t(ftime); // (2)
std::cout << "Write time on server " // (3)
      << std::asctime(std::localtime(&cftime));
std::cout << "Write time on server " // (4)
      << std::asctime(std::gmtime(&cftime)) << std::endl;

fs::last_write_time(path, ftime + 2h); // (5)
ftime = fs::last_write_time(path); // (6)

cftime = std::chrono::system_clock::to_time_t(ftime);
std::cout << "Local time on client "
      << std::asctime(std::localtime(&cftime)) << std::endl;

```

Line (1) gives the write time of the newly created file. You can use `ftime` in (2) to initialise `std::chrono::system_clock`. `ftime` is of type `std::filesystem::file_time_type` which is in this case an alias for `std::chrono::system_clock`; therefore, you can initialise `std::localtime` in (3) and present the calendar time in a textual representation. If you use `std::gmtime` (4) instead of `std::localtime`, nothing will change. This puzzled me because the Coordinated Universal Time (UTC) differs 2 hours from the local time in Germany. That's due to the server for the online-compiler on en.cppreference.com. UTS and local time are set to the same time on the server.

Here is the output of the program. I moved the write time of the file 2 hours to the future (5) and read it back from the filesystem (6). This adjusts the time so it corresponds to the local time in Germany.

```

Write time on server Tue Oct 10 06:28:04 2017
Write time on server Tue Oct 10 06:28:04 2017

Local time on client Tue Oct 10 08:28:04 2017

```

Space Information on the Filesystem

The global function `std::filesystem::space` returns a `std::filesystem::space_info` object that has three members: `capacity`, `free`, and `available`.

- `capacity`: total size of the filesystem
- `free`: free space on the filesystem
- `available`: free space to a non-privileged process (equal or less than `free`)

All sizes are in bytes.

The output of the following program is from cppreference.com. All paths I tried were on the same filesystem; therefore, I always get the same answer.

```
Space information


---


// space.cpp
...
#include <filesystem>
...
namespace fs = std::filesystem;
...

fs::space_info root = fs::space("/");
fs::space_info usr = fs::space("/usr");

std::cout << ".      Capacity      Free      Available\n"
      << "/" << root.capacity << " "
      << root.free << " " << root.available << "\n"
      << "usr" << usr.capacity << " "
      << usr.free << " " << usr.available;
```

	Capacity	Free	Available
/	42140499968	18342744064	17054289920
usr	42140499968	18342744064	17054289920

File types

By using the following predicates, you can easily ask for the type of file.

file types	The file types of the filesystem
	Description

<code>is_block_file</code>	Checks if the path refers to a block file.
<code>is_character_file</code>	Checks if the path refers to a character file.
<code>is_directory</code>	Checks if the path refers to a directory.
<code>is_empty</code>	Checks if the path refers to an empty file or directory.
<code>is_fifo</code>	Checks if the path refers to a named pipe .
<code>is_other</code>	Checks if the path refers to another file.
<code>is_regular_file</code>	Checks if the path refers to a regular file.
<code>is_socket</code>	Checks if the path refers to an IPC socket.
<code>is_symlink</code>	Checks if the path refers to a symbolic link.
<code>status_known</code>	Checks if the file status is known.

Getting the type of a file

The predicates give you the information on the type of a file. More than one predicate may be right for one file. In the next example, a symbolic link referencing a regular file is both: a regular file and a symbolic link.

Type of a file

```
// fileType.cpp
...
#include <filesystem>
...

namespace fs = std::filesystem;

void printStatus(const fs::path& path_){
    std::cout << path_;
    if(!fs::exists(path_)) std::cout << " does not exist";
    else{
        if(fs::is_block_file(path_)) std::cout << " is a block file\n";
        if(fs::is_character_file(path_)) std::cout << " is a character device\n";
        if(fs::is_directory(path_)) std::cout << " is a directory\n";
        if(fs::is_fifo(path_)) std::cout << " is a named pipe\n";
        if(fs::is_regular_file(path_)) std::cout << " is a regular file\n";
        if(fs::is_socket(path_)) std::cout << " is a socket\n";
    }
}
```

```

        if(fs::is_symlink(path_)) std::cout << "           is a symlink\n";
    }
}

...

fs::create_directory("rainer");
printStatus("rainer");

std::ofstream("rainer/regularFile.txt");
printStatus("rainer/regularFile.txt");

fs::create_directory("rainer/directory");
printStatus("rainer/directory");

mkfifo("rainer/namedPipe", 0644);
printStatus("rainer/namedPipe");

struct sockaddr_un addr;
addr.sun_family = AF_UNIX;
std::strcpy(addr.sun_path, "rainer/socket");
int fd = socket(PF_UNIX, SOCK_STREAM, 0);
bind(fd, (struct sockaddr*)&addr, sizeof addr);
printStatus("rainer/socket");

fs::create_symlink("rainer/regularFile.txt", "symlink");
printStatus("symlink");

printStatus("dummy.txt");

fs::remove_all("rainer");

```

"rainer" is a directory
 "rainer/regularFile.txt" is a regular file
 "rainer/directory" is a directory
 "rainer/namedPipe" is a named pipe
 "rainer/socket" is a socket
 "symlink" is a regular file
 is a symlink
 "dummy.txt" does not exist

19. Multithreading

For the first time with C++11, C++ supports native multithreading. This support consists of two parts: A *well-defined* memory model and a standardised threading interface.

Memory Model

The foundation of multithreading is a *well-defined* memory model. This memory model has to deal with the following points:

- Atomic operations: Operations that can be performed without interruption.
- Partial ordering of operations: Sequence of operations that must not be reordered.
- Visible effects of operations: Guarantees when operations on shared variables are visible in other threads.

The C++ memory model has a lot in common with its predecessor: the Java memory model. On the contrary, C++ permits the breaking of sequential consistency. The sequential consistency is the default behaviour of atomic operations.

The sequential consistency provides two guarantees.

1. The instructions of a program are executed in source code order.
2. There is a global order of all operations on all threads.

Atomic Data Types

C++ has a set of atomic data types for booleans, characters, integrals, floating points, pointers and smart pointers such as `std::shared_ptr` and `std::weak_ptr`. They require the header `<atomic>`. You can define your atomic data type with the class template `std::atomic`, but there are serious restrictions for your type `std::atomic<MyType>`:

For `MyType` there are the following restrictions:

- The copy assignment operator for `MyType`, for all base classes of `MyType` and all non-static members of `MyType`, must be trivial. Only a *compiler-generated* copy assignment operator is trivial.
- `MyType` must not have virtual methods or base classes.
- `MyType` must be bitwise copyable and comparable so that the C functions `memcpy` or `memcmp` can be applied.

Atomic data types have atomic operations. For example load and store:

Atomic data types

```
// atomic.cpp
...
#include <atomic>
...

std::atomic_int x, y;
int r1, r2;

void writeX(){
    x.store(1);
    r1= y.load();
}

void writeY(){
    y.store(1);
    r2= x.load();
}

x= 0;
y= 0;

std::thread a(writeX);
std::thread b(writeY);

a.join();
b.join();

std::cout << r1 << r2 << std::endl;
```

Possible outputs for `r1` and `r2` are `11`, `10` and `01`.

Additionally, the class template `std::atomic_ref` applies atomic operations to the referenced object. Concurrent writing and reading of atomic object is thread-safe. The lifetime of the referenced object must exceed the lifetime of the `atomic_ref`. Accessing a subobject of the referenced object with an `atomic_ref` is not well-defined.

An atomic counter with `std::atomic_ref`

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

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

Threads

To use the multithreading interface of C++ you need the header `<thread>`. C++ support with C++11 a `std::thread` and with C++20 a improved `std::jthread`.

`std::thread`

Creation

A thread `std::thread` represents an executable unit. This executable unit, which the thread immediately starts, gets its work package as a [callable unit](#). A callable unit can be a function, a function object or a lambda function:

Thread creation

```
// threadCreate.cpp
...
#include <thread>
...
using namespace std;

void helloFunction(){
    cout << "function" << endl;
}

class HelloFunctionObject {
public:
    void operator()() const {
        cout << "function object" << endl;
    }
};

thread t1(helloFunction);           // function

HelloFunctionObject helloFunctionObject;
thread t2(helloFunctionObject);     // function object

thread t3([]{ cout << "lambda function"; }); // lambda function
```

Lifetime

The creator of a thread has to take care of the lifetime of its created thread. The executable unit of the created thread ends with the end of the callable. Either the creator is waiting until the created thread `t` is done (`t.join()`) or the creator detaches itself from the created thread: `t.detach()`. A thread `t` is *joinable* if no operation `t.join()` or `t.detach()` was performed on it. A *joinable* thread throws in its destructor the exception `std::terminate`, and the program terminates.

Lifetime of a thread

```
// threadLifetime.cpp
...
#include <thread>
...

thread t1(helloFunction);           // function

HelloFunctionObject helloFunctionObject;
thread t2(helloFunctionObject);      // function object

thread t3([]{ cout << "lambda function"; }); // lambda function

t1.join();
t2.join();
t3.join();
```

A thread that is detached from its creator is typically called a daemon thread because it runs in the background.



Move threads with caution

Threads can be moved but not copied.

```
#include <thread>
...
std::thread t([]{ cout << "lambda function"; });
std::thread t2;
t2 = std::move(t);

std::thread t3([]{ cout << "lambda function"; });
t2 = std::move(t3);           // std::terminate
```

By performing `t2 = std::move(t)` thread `t2` gets the callable of thread `t`. Assuming thread `t2` already had a callable and is *joinable* the C++ runtime would call `std::terminate`. This happens exactly in `t2 = std::move(t3)` because `t2` neither executed `t2.join()` nor `t2.detach()` before.

Arguments

A `std::thread` is a variadic template. This means in particular that it can get an arbitrary number of arguments by copy or by reference. Either the callable or the thread can get the arguments. The thread delegates them to the callable:
`tPerCopy2` and `tPerReference2`.

```
#include <thread>
...
using namespace std;
```

```
void printStringCopy(string s){ cout << s; }
void printStringRef(const string& s){ cout << s; }

string s{"C++"};

thread tPerCopy([=]{ cout << s; });           // C++
thread tPerCopy2(printStringCopy, s);          // C++
tPerCopy.join();
tPerCopy2.join();

thread tPerReference([&]{ cout << s; });        // C++
thread tPerReference2(printStringRef, s);       // C++
tPerReference.join();
tPerReference2.join();
```

The first two threads get their argument s by copy, the second two by reference.



By default threads should get their arguments by copy

Arguments of a thread

```
// threadArguments.cpp
...
#include <thread>
...

using std::this_thread::sleep_for;
using std::this_thread::get_id;

struct Sleeper{
    Sleeper(int& i_):i{i_}{};
    void operator() (int k){
        for (unsigned int j= 0; j <= 5; ++j){
            sleep_for(std::chrono::milliseconds(100));
            i += k;
        }
        std::cout << get_id();           // undefined behaviour
    }
private:
    int& i;
};

int valSleeper= 1000;

std::thread t(Sleeper(valSleeper), 5);
t.detach();

std::cout << valSleeper;           // undefined behaviour
```

This program snippet has undefined behaviour. First the lifetime of `std::cout` is bound to the lifetime of the main thread and the created thread gets its variable `valSleeper` by reference. The issue is that the created thread lives longer than its creator, therefore `std::cout` and `valSleeper` lose their validity if the main thread is done. Second, `valSleeper` is a shared, mutable variable, which is concurrently used by the main-thread and the child-thread. Consequently, this is a [data race](#).

Operations

You can perform many operations on a thread `t`.

Operations with a `std::thread`

Method	Description
<code>t.join()</code>	Waits until thread <code>t</code> has finished its executable unit.

<code>t.detach()</code>	Executes the created thread <code>t</code> independent of the creator.
<code>t.joinable()</code>	Checks if thread <code>t</code> supports the calls <code>join</code> or <code>detach</code> .
<code>t.get_id()</code> and <code>std::this_thread::get_id()</code>	Returns the identity of the thread.
<code>std::thread::hardware_concurrency()</code>	Indicates the number of threads that can be run in parallel.
<code>std::this_thread::sleep_until(absTime)</code>	Puts the thread <code>t</code> to sleep until time <code>absTime</code> .
<code>std::this_thread::sleep_for(relTime)</code>	Puts the thread <code>t</code> to sleep for the duration <code>relTime</code> .
<code>std::this_thread::yield()</code>	Offers the system to run another thread.
<code>t.swap(t2)</code> and <code>std::swap(t1, t2)</code>	Swaps the threads.

You can only call `t.join()` or `t.detach()` once on a thread `t`. If you attempt to call these more than once you get the exception `std::system_error`. `std::thread::hardware_concurrency` returns the number of cores or 0 if the runtime cannot determine the number. The `sleep_until` and `sleep_for` operations need a [time point](#) or a [time duration](#) as an argument.

Threads cannot be copied but can be moved. A swap operation performs a move when possible.

Operations on a thread

```
// threadMethods.cpp
...
#include <thread>
...
```

```
using std::this_thread::get_id;

std::thread::hardware_concurrency();      // 4

std::thread t1([]{ get_id(); });
std::thread t2([]{ get_id(); });
t1.get_id();                            // 139783038650112
t2.get_id();                            // 139783030257408
t1.swap(t2);

t1.get_id();                            // 139783030257408
t2.get_id();                            // 139783038650112
get_id();                                // 140159896602432
```

std::jthread

std::jthread stands for joining thread. In addition to [std::thread](#) from C++11, std::jthread can automatically join the started thread and signal an interrupt.

Automatically Joining

The *non-intuitive* behaviour of [std::thread](#) is the following: when a std::thread is still joinable, std::terminate is called in its destructor.

To the contrary, a std::jthread thr automatically joins in its destructor such as in this case if still joinable.

Terminating a still joinable std::jthread

```
// jthreadJoinable.cpp
...
#include <thread>
...

std::jthread thr{}; std::cout << "std::jthread" << "\n"; } // std::jthread
std::cout << "thr.joinable(): " << thr.joinable() << "\n"; // thr.joinable(): true
```

Additionally to the [std::thread](#), a std::jthread is interruptible.

Stop Token

The additional functionality of the cooperatively joining thread is based on the std::stop_token, the std::stop_callback, and the std::stop_source. The following program should give you the general idea.

Interrupt an non-interruptable and interruptable std::jthread

```
// interruptJthread.cpp
```

```

...
#include <thread>
#include <stop_token>

using namespace::std::literals;

...

std::jthread nonInterruptable([]{ // (1)
    int counter{0};
    while (counter < 10){
        std::this_thread::sleep_for(0.2s);
        std::cerr << "nonInterruptable: " << counter << std::endl;
        ++counter;
    }
});

std::jthread interruptable([](std::stop_token stoken){ // (2)
    int counter{0};
    while (counter < 10){
        std::this_thread::sleep_for(0.2s);
        if (stoken.stop_requested()) return; // (3)
        std::cerr << "interruptable: " << counter << std::endl;
        ++counter;
    }
});

std::this_thread::sleep_for(1s);

std::cerr << "Main thread interrupts both jthreads" << std::endl;
nonInterruptable.request_stop(); // (4)
interruptable.request_stop();

```

I start in the main program the two threads `nonInterruptable` and `interruptable` ((1) and (2)). In contrast to the thread `nonInterruptable`, the thread `interruptable` gets a `std::stop_token` and uses it in (3) to check if it was interrupted: `stoken.stop_requested()`. In case of an interrupt the lambda function returns and, therefore, the thread ends. The call `interruptable.request_stop()` (4) triggers the end of the thread. This does not hold for the previous call `nonInterruptable.request_stop()`, which does not have an effect.

```

rainer@seminar:~> interruptJthread
interruptable: 0
nonInterruptable: 0
interruptable: 1
nonInterruptable: 1
interruptable: 2
nonInterruptable: 2
interruptable: 3
nonInterruptable: 3
Main thread interrupts both jthreads
nonInterruptable: 4
nonInterruptable: 5
nonInterruptable: 6
nonInterruptable: 7
nonInterruptable: 8
nonInterruptable: 9
rainer@seminar:~>

```

std::stop_token, std::stop_source, and std::stop_callback

A `stop_token`, a `std::stop_callback`, or a `std::stop_source` enables it to asynchronously request an execution to stop or ask if an execution gets a stop signal. The `std::stop_token` can be passed to an operation and afterwards 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 `std::stop_callback` share the ownership of an associated stop state. The calls `request_stop()`, `stop_requested()`, and `stop_possible()` are atomic operations.

The components `std::stop_source` and `std::stop_token` provide the following attributes for stop handling.

Constructors of `std::stop_source`

```

stop_source();
explicit stop_source(std::nostopstate_t) noexcept;

```

The default-constructor `std::stop_source` gets a stop-source. The constructor taking `std::nostopstate_t` constructs an empty `std::stop_source` without

associated stop-state.

Member functions of std::stop_source src

Member function	Description
src.get_token()	If <code>stop_possible()</code> , returns a <code>stop_token</code> for the associated stop-state. Otherwise, returns a default-constructed (empty) <code>stop_token</code> .
src.stop_possible()	<code>true</code> if <code>src</code> can be requested to stop.
src.stop_requested()	<code>true</code> if <code>stop_possible()</code> and <code>request_stop()</code> was called by one of the owners.
src.request_stop()	Calls a stop request if <code>stop_possible()</code> and <code>!stop_requested()</code> . Otherwise, the call has no effect.
src.stop_possible()	means that <code>src</code> has an associated stop-state. <code>src.stop_requested()</code> returns <code>true</code> when <code>src</code> has an associated stop-state and was requested to stop before. <code>src.request_stop()</code> is successful and returns <code>true</code> if <code>src</code> has an associated stop-state if was not request to stop before.

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 for its associated stop source `src`. The stop token `stoken` observes the stop source `src`.

Member functions of std::stop_token stoken

Member function	Description
stoken.stop_possible()	Returns <code>true</code> if <code>stoken</code> has an associated stop-state.
stoken.stop_requested()	<code>true</code> if <code>request_stop()</code> was called on the associated <code>std::stop_source src</code> , otherwise <code>false</code> .

A default-constructed stop token has no associated stop-state.
`stoken.stop_possible` also returns `true` if the stop request has already been

made. `stoken.stop_requested()` returns `true` when 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

```
std::jthread jthr([](std::stop_token stoken) {
    ...
    std::stop_token interruptDisabled;
    std::swap(stoken, interruptDisabled); // (1)
    ...
    std::swap(stoken, interruptDisabled); // (2)
    ...
})
```

`std::stop_token interruptDisabled` has no associated stop-state. This means, the thread `jthr` can in all lines excepts line (1) and (2) accept stop requests.

Shared Variables

If more than one thread is sharing a variable, you have to coordinate the access. That's the job for mutexes and locks in C++.

Data race

A data race

A data race is a state in which at least two threads access a shared data at the same time, and at least one of the threads is a writer. Therefore the program has undefined behaviour.

You can observe very well the interleaving of threads if a few threads write to `std::cout`. The output stream `std::cout` is, in this case, the shared variable.

Unsynchronised writing to `std::cout`

```
// withoutMutex.cpp
...
#include <thread>
...

using namespace std;

struct Worker{
    Worker(string n):name(n){}
    void operator() (){

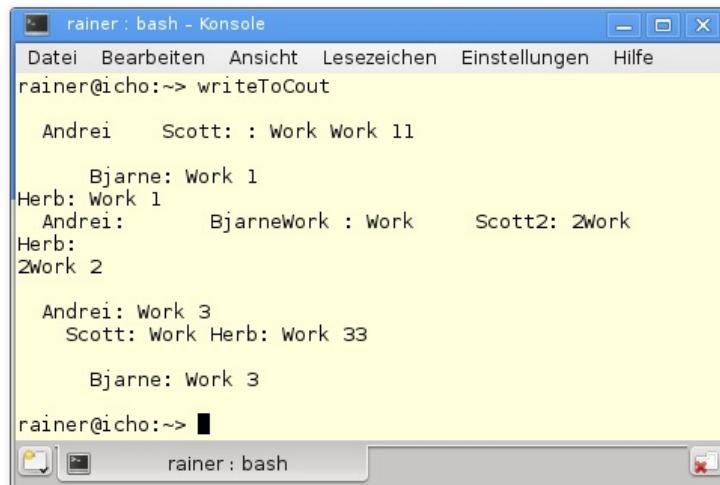

```

```

    for (int i= 1; i <= 3; ++i){
        this_thread::sleep_for(chrono::milliseconds(200));
        cout << name << ":" << "Work " << i << endl;
    }
}
private:
    string name;
};

thread herb= thread(Worker("Herb"));
thread andrei= thread(Worker("Andrei"));
thread scott= thread(Worker ("Scott"));
thread bjarne= thread(Worker("Bjarne"));

```



The screenshot shows a terminal window titled "rainer : bash - Konsole". The window has a menu bar with German labels: Datei, Bearbeiten, Ansicht, Lesezeichen, Einstellungen, Hilfe. The command "writeToCout" is run at the prompt "rainer@icho:~>". The output is as follows:

```

Andrei      Scott: : Work Work 11
          Bjarne: Work 1
Herb: Work 1
          Andrei:      BjarneWork : Work      Scott2: 2Work
Herb:
2Work 2

Andrei: Work 3
          Scott: Work Herb: Work 33
          Bjarne: Work 3

```

The output on `std::cout` is not coordinated.



The streams are *thread-safe*

The C++11 standard guarantees that the characters are written atomically. Therefore you don't need to protect them. You only have to protect the interleaving of the threads on the stream if the characters should be written or read in the right sequence. That guarantee holds for the input and output streams.

With C++20, we have synchronized outputstreams such as `std::osyncstream` and `std::wosyncstream`. The guarantee, that the writing to one outputstream is synchronized. The output is written to an internal buffer and flushed, when it goes out-of-scope. A synchronized outputstream can have a name such as `synced_out` or be without a name.

Synchronized outputstreams

```
{  
    std::osyncstream synced_out(std::cout);  
    synced_out << "Hello, "  
    synced_out << "World!"  
    synced_out << std::endl; // no effect  
    synced_out << "and more!\n";  
} // destroys the synced_output and emits the internal buffer  
  
std::osyncstream(std::cout) << "Hello, " << "World!" << "\n";
```

`std::cout` is in the example the shared variable, which should have exclusive access to the stream.

Mutexes

Mutex (*mutual exclusion*) guarantees that only one thread can access the critical region at one time. They need the header `<mutex>`. A mutex locks the critical section by the call `m.lock()` and unlocks it by `m.unlock()`.

Synchronisation with `std::mutex`

```
// mutex.cpp  
...  
#include <mutex>  
#include <thread>  
...  
  
using namespace std;  
  
std::mutex mutexCout;  
  
struct Worker{  
    Worker(string n):name(n){};  
    void operator() (){  
        for (int i= 1; i <= 3; ++i){  
            this_thread::sleep_for(chrono::milliseconds(200));  
            mutexCout.lock();  
            mutexCout.unlock();  
        }  
    }  
};
```

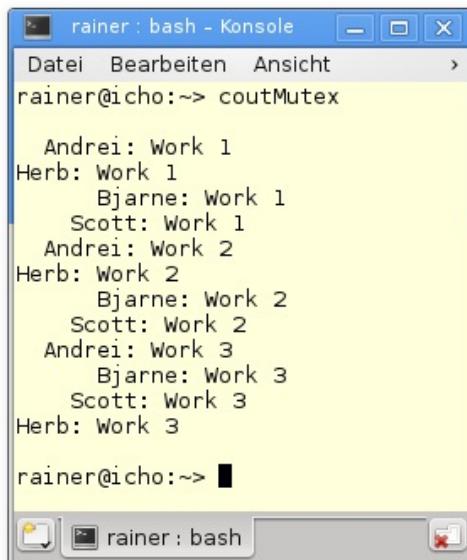
```

        cout << name << ":" << "Work " << i << endl;
        mutexCout.unlock();
    }
}
private:
    string name;
};

thread herb= thread(Worker("Herb"));
thread andrei= thread(Worker(" Andrei"));
thread scott= thread(Worker (" Scott"));
thread bjarne= thread(Worker(" Bjarne"));

```

Now each thread after each other writes coordinated to `std::cout` because it uses the same mutex `mutexCout`.



C++ has five different mutexes. They can lock recursively, tentative with and without time constraints.

Method	Mutex variations				
	<u>mutex</u>	<u>recursive_mutex</u>	<u>timed_mutex</u>	<u>recursive_timed_mutex</u>	
<code>m.lock</code>	yes	yes	yes	yes	yes
<code>m.unlock</code>	yes	yes	yes	yes	yes
<code>m.try_lock</code>	yes	yes	yes	yes	yes
<code>m.try_lock_for</code>			yes	yes	yes
<code>m.try_lock_until</code>			yes	yes	yes

The `std::shared_time_mutex` enables it to implement [reader-writer locks](#). The method `m.try_lock_for(relTime)` needs a relative [time duration](#); the method `m.try_lock_until(absTime)` a absolute [time point](#).

Deadlocks

Deadlocks

A deadlock is a state in which two or more threads are blocked because each thread waits for the release of a resource before it releases its resource.

You can get a deadlock very quickly if you forget to call `m.unlock()`. That happens for example in case of an exception in the function `getVar()`.

```
m.lock();
sharedVar= getVar();
m.unlock()
```



Don't call an unknown function while holding a lock

If the function `getVar` tries to get the same lock by calling `m.lock()` you will get a deadlock, because it will not be successful and the call will block forever.

Locking two mutexes in the wrong order is another typical reason for a deadlock.

A deadlock

```
// deadlock.cpp
...
#include <mutex>
...

struct CriticalData{
    std::mutex mut;
};

void deadLock(CriticalData& a, CriticalData& b){
    a.mut.lock();
    std::cout << "get the first mutex\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    b.mut.lock();
    std::cout << "get the second mutex\n";
    a.mut.unlock(), b.mut.unlock();
}

CriticalData c1;
CriticalData c2;
```

```
std::thread t1([&]{ deadLock(c1, c2); });
std::thread t2([&]{ deadLock(c2, c1); });
```

The short time window of one millisecond
(`std::this_thread::sleep_for(std::chrono::milliseconds(1))`) is enough to produce with high probability a deadlock because each thread is waiting forever on the other mutex. The result is a standstill.



Encapsulate a mutex in a lock

It's very easy to forget to unlock a mutex or lock mutexes in a different order. To overcome most of the problems with a mutex, encapsulate it in a lock.

Locks

You should encapsulate a mutex in a lock to release the mutex automatically. A lock is an implementation of the [RAII idiom](#) because the lock binds the lifetime of the mutex to its lifetime. C++11 has `std::lock_guard` for the simple and `std::unique_lock` for the advanced use case, respectively. Both need the header `<mutex>`. With C++14 C++ has a `std::shared_lock` which is in the combination with the mutex [`std::shared_time_mutex`](#) the base for reader-writer locks.

```
std::lock_guard
```

`std::lock_guard` supports only the simple use case. Therefore it can only bind its mutex in the constructor and release it in the destructor. So the synchronisation of the [worker example](#) is reduced to the call of the constructor.

Synchronisation with `std::lock_guard`

```

// lockGuard.cpp
...
std::mutex coutMutex;

struct Worker{
    Worker(std::string n):name(n){}
    void operator() (){
        for (int i= 1; i <= 3; ++i){
            std::this_thread::sleep_for(std::chrono::milliseconds(200));
            std::lock_guard<std::mutex> myLock(coutMutex);
            std::cout << name << ":" << "Work " << i << std::endl;
        }
    }
private:
    std::string name;
};

```

std::unique_lock

The usage of `std::unique_lock` is more expensive than the usage of `std::lock_guard`. In contrary a `std::unique_lock` can be created with and without mutex, can explicitly lock or release its mutex or can delay the lock of its mutex.

The following table shows the methods of a `std::unique_lock lk`.

Method	The interface of <code>std::unique_lock</code>	Description
<code>lk.lock()</code>		Locks the associated mutex.
<code>std::lock(lk1, lk2, ...)</code>		Locks atomically the arbitrary number of associated mutexes.
<code>lk.try_lock()</code> and <code>lk.try_lock_for(relTime)</code> and <code>lk.try_lock_until(absTime)</code>		Tries to lock the associated mutex.
<code>lk.release()</code>		Release the mutex. The mutex remains locked.
<code>lk.swap(lk2)</code> and <code>std::swap(lk, lk2)</code>		Swaps the locks.
<code>lk.mutex()</code>		Returns a pointer to the associated mutex.

`lk.owns_lock()` Checks if the lock has a mutex.

Deadlocks caused by [acquiring locks in different order](#) can easily be solved by `std::atomic`.

```
std::unique_lock
```

```
// deadLockResolved. cpp
...
#include <mutex>
...

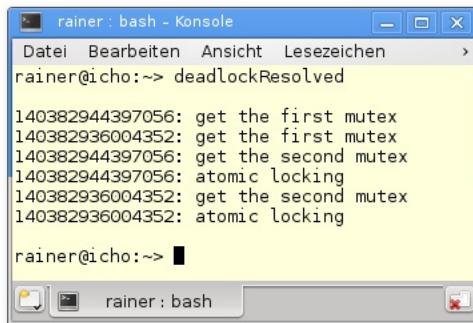
using namespace std;
struct CriticalData{
    mutex mut;
};

void deadLockResolved(CriticalData& a, CriticalData& b){
    unique_lock<mutex>guard1(a.mut, defer_lock);
    cout << this_thread::get_id() << ": get the first lock" << endl;
    this_thread::sleep_for(chrono::milliseconds(1));
    unique_lock<mutex>guard2(b.mut, defer_lock);
    cout << this_thread::get_id() << ": get the second lock" << endl;
    cout << this_thread::get_id() << ": atomic locking";
    lock(guard1, guard2);
}

CriticalData c1;
CriticalData c2;

thread t1([&]{ deadLockResolved(c1, c2); });
thread t2([&]{ deadLockResolved(c2, c1); });
```

Because of the argument `std::defer_lock` of the `std::unique_lock`, the locking of `a.mut` and `b.mut` is deferred. The locking takes place atomically in the call `std::lock(guard1, guard2)`.



The screenshot shows a terminal window titled "rainer : bash - Konsole". The command "deadlockResolved" was run. The output shows two threads attempting to acquire mutexes. Thread 1 (140382944397056) acquires the first mutex, sleeps for 1 millisecond, then acquires the second mutex, and performs an atomic locking operation. Thread 2 (140382936004352) also tries to acquire both mutexes, but since they are already held by thread 1, it waits until thread 1 releases them. This results in a deadlock.

```
rainer@icho:~> deadlockResolved
140382944397056: get the first mutex
140382936004352: get the first mutex
140382944397056: get the second mutex
140382944397056: atomic locking
140382936004352: get the second mutex
140382936004352: atomic locking
rainer@icho:~>
```

`std::shared_lock`

`std::shared_lock` has the same interface as `std::unique_lock`. Also, a `std::shared_lock` supports the case where multiple threads share the same locked mutex. For this special use case, you have to use a `std::shared_lock` in combination with a `std::shared_timed_mutex`. However, if multiple threads use the same `std::shared_time_mutex` in a `std::unique_lock` only one thread can possess it.

```
#include <mutex>
...
std::shared_timed_mutex sharedMutex;
std::unique_lock<std::shared_timed_mutex> writerLock(sharedMutex);
std::shared_lock<std::shared_time_mutex> readerLock(sharedMutex);
std::shared_lock<std::shared_time_mutex> readerLock2(sharedMutex);
```

The example presents the typical reader-writer lock scenario. The `writerLock` of type `std::unique_lock<std::shared_timed_mutex>` can only exclusively have the `sharedMutex`. Both of the reader locks `readerLock` and `readerLock2` of type `std::shared_lock<std::shared_time_mutex>` can share the same mutex `sharedMutex`.

Thread-safe Initialization

If you don't modify the data, it's sufficient to initialise them in a *thread-safe* way. C++ offers various ways to achieve this: using a constant expression, using static variables with block scope and using the function `std::call_once` in combination with the flag `std::once::flag`.

Constant Expressions

A constant expression is initialised at compile time. Therefore they are per se *thread-safe*. By using the keyword `constexpr` before a variable, the variable becomes a constant expression. Instances of *user-defined* type can also be a constant expression and therefore be initialised in a *thread-safe* way if the methods are declared as constant expressions.

```
struct MyDouble{
    constexpr MyDouble(double v):val(v){};
    constexpr double getValue(){ return val; }
private:
    double val
};

constexpr MyDouble myDouble(10.5);
std::cout << myDouble.getValue();           // 10.5
```

Static Variables Inside a Block

If you define a static variable in a block, the C++11 runtime guarantees that the variable is initialised in a *thread-safe* way.

```
void blockScope(){
    static int MySharedDataInt= 2011;
}
```

std::call_once and std::once_flag

std::call_once takes two arguments: the flag std::once_flag and a [callable](#). The C++ runtime guarantees with the help of the flag std::once_flag that the callable is executed exactly once.

Thread-safe initialisation

```
// callOnce.cpp
...
#include <mutex>
...

using namespace std;

once_flag onceFlag;
void do_once(){
    call_once(onceFlag, []{ cout << "Only once." << endl; });
}
thread t1(do_once);
thread t2(do_once);
```

Although both threads executed the function do_once only one of them is successful, and the lambda function []{cout << "Only once." << endl;} is executed exactly once.



You can further use the same std::once_flag to register different callables and only one of this callables is called.

Thread Local Data

By using the keyword `thread_local`, you have thread local data also known as thread local storage. Each thread has its copy of the data. *Thread-local* data behaves like static variables. They are created at their first usage, and their lifetime is bound to the lifetime of the thread.

Thread local data

```
// threadLocal.cpp
...
std::mutex coutMutex;
thread_local std::string s("hello from ");

void addThreadLocal(std::string const& s2){
    s+= s2;
    std::lock_guard<std::mutex> guard(coutMutex);
    std::cout << s << std::endl;
    std::cout << "&s: " << &s << std::endl;
    std::cout << std::endl;
}

std::thread t1(addThreadLocal, "t1");
std::thread t2(addThreadLocal, "t2");
std::thread t3(addThreadLocal, "t3");
std::thread t4(addThreadLocal, "t4");
```

Each thread has its copy of the `thread_local` string, therefore, each string `s` modifies its string independently, and each string has its unique address:



The screenshot shows a terminal window titled "rainer : bash - Konsole <2>". The command "threadLocal" was run, which resulted in four separate outputs, each consisting of a string and a memory address. The strings are "hello from t1", "hello from t4", "hello from t2", and "hello from t3". The memory addresses are "&s: 0x7f64a4b256f8", "&s: 0x7f64a33226f8", "&s: 0x7f64a43246f8", and "&s: 0x7f64a3b236f8" respectively. The terminal window has a standard Linux-style interface with a menu bar and a status bar at the bottom.

Condition Variables

Condition variables enable threads to be synchronised via messages. They need the header `<condition_variable>`. One thread acts as a sender, and the other as

a receiver of the message. The receiver waits for the notification of the sender. Typical use cases for condition variables are producer-consumer workflows.

A condition variable can be the sender but also the receiver of the message.

Method	The methods of the condition variable cv
Method	Description
cv.notify_one()	Notifies a waiting thread.
cv.notify_all()	Notifies all waiting threads.
cv.wait(lock, ...)	Waits for the notification while holding a std::unique_lock.
cv.wait_for(lock, relTime, ...)	Waits for a time duration for the notification while holding a std::unique_lock.
cv.wait_until(lock, absTime, ...)	Waits until a time for the notification while holding a std::unique_lock.

Sender and receiver need a lock. In case of the sender a std::lock_guard is sufficient because it only once calls lock and unlock. In the case of the receiver a std::unique_lock is necessary because it typically often locks and unlocks its mutex.

```
Condition variable
// conditionVariable.cpp
...
#include <condition_variable>
...

std::mutex mutex_;
std::condition_variable condVar;
bool dataReady= false;

void doTheWork(){
    std::cout << "Processing shared data." << std::endl;
}

void waitingForWork(){
    std::cout << "Worker: Waiting for work." << std::endl;
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
    doTheWork();
    std::cout << "Work done." << std::endl;
}
```

```

void setDataReady(){
    std::lock_guard<std::mutex> lck(mutex_);
    dataReady=true;
    std::cout << "Sender: Data is ready." << std::endl;
    condVar.notify_one();
}

std::thread t1(waitingForWork);
std::thread t2(setDataReady);

```



Using a condition variable may sound easy but there are two critical issues.



Protection against spurious wakeup

To protect itself against spurious wakeup, the `wait` call of the condition variable should use an additional [predicate](#). The predicate ensures that the notification is indeed from the sender. I use the lambda function `[]{ return dataReady; }` as the predicate. `dataReady` is set to `true` by the sender.



Protection against lost wakeup

To protect itself against lost wakeup, the `wait` call of the condition variable should use an additional [predicate](#). The predicate ensures that the notification of the sender is not lost. The notification is lost if the sender notifies the receiver before the receiver is waiting. Therefore the receiver waits forever. The receiver now checks at first its predicate: `[]{ return dataReady; }`.

Semaphores

Semaphores are a synchronisation mechanism used to control concurrent access to a shared resource. A counting semaphore is a special semaphore which has a counter that is bigger than zero. The counter is initialised 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.

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.

```
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 call can happen on different threads. The following table present the interface of a `std::counting_semaphore`.

Method	Methods of a semaphore <code>sem</code>	Description
<code>counting_semaphore::max()</code>		Return the maximum value of the counter.
<code>sem.release(upd = 1)</code>		Atomically increases counter by <code>upd</code> .
<code>sem.acquire()</code>		Performs <code>sem.try_acquire</code> and blocks until counter is greater than zero.
<code>sem.try_acquire()</code>		Atomically decrements the counter.
<code>sem.try_acquire_for(relTime)</code>		Performs <code>sem.try_acquire</code> for the time duration.
<code>sem.try_acquire_until(absTime)</code>		Performs <code>sem.try_acquire</code> until the time point.

The method `sem.try_lock_for(relTime)` requires a relative [time duration](#); the method `sem.try_lock_until(absTime)` requires an absolute [time point](#).

Semaphores are often the safer and faster alternative to [condition variables](#):

```
std::counting_semaphore
// threadSynchronisationSemaphore.cpp

#include <semaphore>

...
std::counting_semaphore<1> prepareSignal();           // (1)

void prepareWork() {

    myVec.insert(myVec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared." << '\n';
    prepareSignal.release();                           // (2)
}

void completeWork() {

    std::cout << "Waiter: Waiting for data." << '\n';
    prepareSignal.acquire();                          // (3)
    myVec[2] = 2;
    std::cout << "Waiter: Complete the work." << '\n';
    for (auto i: myVec) std::cout << i << " ";
    std::cout << '\n';
}

std::thread t1(prepareWork);
std::thread t2(completeWork);

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

The `std::counting_semaphore prepareSignal` (line 1) can have the values 0 and 1. In the concrete example, it's initialized with 0. This means, that the call `prepareSignal.release()` sets the value to 1 (line 2) and unblocks the call `prepareSignal.acquire()` (line 3).

```

C:\Users\seminar>threadSynchronisationSemaphore.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronisationSemaphore.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronisationSemaphore.exe
Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronisationSemaphore.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>

```

Coordination Types

Latches and barriers are coordination types which enable some threads to block until a counter becomes zero. With C++20 we have latches and barriers in two variations: `std::latch`, and `std::barrier`.

`std::latch`

Now, I have a closer look at the interface of a `std::latch`.

Member function	Member function of a <code>std::latch</code> lat
<code>lat.count_down(upd = 1)</code>	Atomically decrements the counter by <code>upd</code> without blocking the caller.
<code>lat.try_wait()</code>	Returns true if <code>counter == 0</code> .
<code>lat.wait()</code>	Returns immediately if <code>counter == 0</code> . If not

blocks until counter == 0.

```
lat.arrive_and_wait(upd = 1) Equivalent to count_down(upd); wait();
```

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() does never wait as its name suggests.

The following program 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.

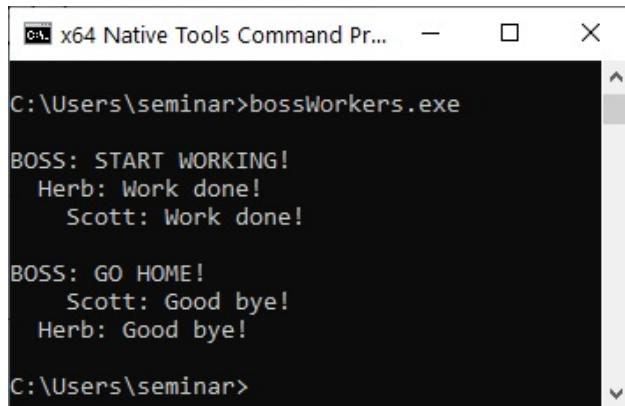
```
std::latch  
-----  
// bossWorkers.cpp  
#include <latch>  
  
...  
  
std::latch workDone(2);  
std::latch goHome(1); // (5)  
  
void synchronizedOut(const std::string s) {  
    std::lock_guard<std::mutex> lo(coutMutex);  
    std::cout << s;  
}  
  
class Worker {  
public:  
    Worker(std::string n): name(n) {};  
  
    void operator() (){  
        // notify the boss when work is done  
        synchronizedOut(name + ": " + "Work done!\n");  
        workDone.count_down(); // (3)  
  
        // waiting before going home  
        goHome.wait();  
        synchronizedOut(name + ": " + "Good bye!\n");  
    }  
private:  
    std::string name;  
};  
  
...  
  
std::cout << "BOSS: START WORKING! " << '\n';  
  
Worker herb(" Herb"); // (1)  
std::thread herbWork(herb);  
  
Worker scott(" Scott"); // (2)  
std::thread scottWork(scott);  
  
workDone.wait(); // (4)
```

```

std::cout << '\n';
goHome.count_down();
std::cout << "BOSS: GO HOME!" << '\n';
herbWork.join();
scottWork.join();

```

The idea of the workflow is straightforward. The two workers herb, and scott (lines 1 and 2) have to fullfil their job. When they finished their job (line 3), they count down the `std::latch workDone`. The boss (main-thread) is blocked in line (4) 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 5). The call `goHome.wait(0)` blocks until the counter becomes 0.



```

C:\Users\seminar>bossWorkers.exe
BOSS: START WORKING!
Herb: Work done!
Scott: Work done!

BOSS: GO HOME!
Scott: Good bye!
Herb: Good bye!

C:\Users\seminar>

```

A `std::barrier` is similar to a `std::latch`.

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 set the counter for the next step (iteration). Immediately after the counter becomes zero, the so-called completion step starts. In this completion step, a [callable](#) is invoked. The `std::barrier` gets its callable in its constructor.

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 function	Member functions of a <code>std::barrier</code> bar
Description	
<code>bar.arrive(upd)</code>	Atomically decrements counter by <code>upd</code> .
<code>bar.wait()</code>	Blocks at the synchronization point until the completion step is done.
<code>bar.arrive_and_wait()</code>	Equivalent to <code>wait(arrive())</code>
<code>bar.arrive_and_drop()</code>	Decrements the counter for the current and the subsequent phase by one.
<code>std::barrier::max</code>	Maximum value supported by the implementation.

The `bar.arrive_and_drop()` call means essentially, that the counter is decremented by one for the next phase.

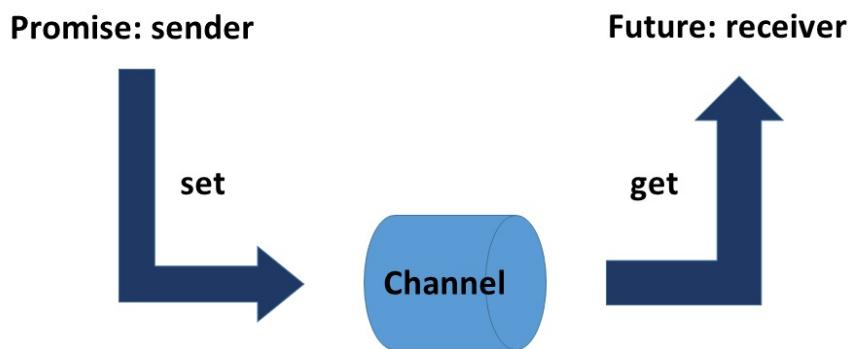
Tasks

In addition to threads, C++ has tasks to perform work asynchronously. Tasks need the header `<future>`. A task is parameterised with a work package and consists of the two associated components, a promise and a future. Both are connected via a data channel. The promise executes the work packages and puts the result in the data channel; the associated future picks up the result. Both communication endpoints can run in separate threads. What's special is that the future can pick up the result at a later time. Therefore the calculation of the result by the promise is independent of the query of the result by the associated future.



Regard tasks as data channels

Tasks behave like data channels. The promise puts its result in the data channel. The future waits for it and picks it up.



Threads versus Tasks

Threads are very different from tasks.

For the communication between the creator thread and the created thread, you have to use a shared variable. The task communicates via its data channel, which is implicitly protected. Therefore a task must not use a protection mechanism like a [mutex](#).

The creator thread is waiting for its child with the `join` call. The `future fut` is using the `fut.get()` call which is blocking if no result is there.

If an exception happens in the created thread, the created thread terminates and therefore the creator and the whole process. On the contrary, the promise can send exceptions to the future, which has to handle the exception.

A promise can serve one or many futures. It can send a value, an exception or only a notification. You can use a task as a safe replacement for a condition variable.

```
#include <future>
#include <thread>
```

```

...
int res;
std::thread t([&]{ res= 2000+11;});
t.join();
std::cout << res << std::endl;           // 2011

auto fut= std::async([]{ return 2000+11; });
std::cout << fut.get() << std::endl;       // 2011

```

The child thread `t` and the asynchronous function call `std::async` calculates the sum of 2000 and 11. The creator thread gets the result from its child thread `t` via the shared variable `res`. The call `std::async` creates the data channel between the sender (promise) and the receiver (future). The future asks the data channel with `fut.get()` for the result of the calculation. The `fut.get` call is blocking.

`std::async`

`std::async` behaves like an asynchronous function call. This function call takes a [callable](#) and its arguments. `std::async` is a variadic template and can, therefore, take an arbitrary number of arguments. The call of `std::async` returns a future object `fut`. That's your handle for getting the result via `fut.get()`. Optionally you can specify a start policy for `std::async`. You can explicitly determine with the start policy if the asynchronous call should be executed in the same thread (`std::launch::deferred`) or in another thread (`std::launch::async`).

What's special about the call `auto fut= std::async(std::launch::deferred, ...)` is that the promise will not immediately be executed. The call `fut.get()` lazy starts the promise.

Lazy and eager with `std::async`

```

// asyncLazyEager.cpp
...
#include <future>
...
using std::chrono::duration;
using std::chrono::system_clock;
using std::launch;

auto begin= system_clock::now();

auto asyncLazy= std::async(launch::deferred, []{ return system_clock::now(); });
auto asyncEager= std::async(launch::async, []{ return system_clock::now(); });
std::this_thread::sleep_for(std::chrono::seconds(1));

auto lazyStart= asyncLazy.get() - begin;
auto eagerStart= asyncEager.get() - begin;

auto lazyDuration= duration<double>(lazyStart).count();
auto eagerDuration= duration<double>(eagerStart).count();

```

```
std::cout << lazyDuration << " sec"; // 1.00018 sec.  
std::cout << eagerDuration << " sec"; // 0.00015489 sec.
```

The output of the program shows that the promise associated with the future `asyncLazy` is executed one second later than the promise associated with the future `asyncEager`. One second is exactly the time duration the creator is sleeping before the future `asyncLazy` asks for its result.



std::async should be your first choice

The C++ runtime decides if `std::async` is executed in a separated thread. The decision of the C++ runtime may be dependent on the number of your cores, the utilisation of your system or the size of your work package.

std::packaged_task

`std::packaged_task` enables you to build a simple wrapper for a callable, which can later be executed on a separate thread.

Therefore four steps are necessary.

I. Wrap your work:

```
std::packaged_task<int(int, int)> sumTask([](int a, int b){ return a+b; });
```

II. Create a future:

```
std::future<int> sumResult= sumTask.get_future();
```

III. Perform the calculation:

```
sumTask(2000, 11);
```

IV. Query the result:

```
sumResult.get();
```

You can move either the `std::package_task` or the `std::future` in a separate thread.

std::packaged_task

```

// packaged_task.cpp
...
#include <future>
...

using namespace std;

struct SumUp{
    int operator()(int beg, int end){
        for (int i= beg; i < end; ++i ) sum += i;
        return sum;
    }
private:
    int beg;
    int end;
    int sum{0};
};

SumUp sumUp1, sumUp2;

packaged_task<int(int, int)> sumTask1(sumUp1);
packaged_task<int(int, int)> sumTask2(sumUp2);

future<int> sum1= sumTask1.get_future();
future<int> sum2= sumTask2.get_future();

deque< packaged_task<int(int, int)>> allTasks;
allTasks.push_back(move(sumTask1));
allTasks.push_back(move(sumTask2));

int begin{1};
int increment{5000};
int end= begin + increment;

while (not allTasks.empty()){
    packaged_task<int(int, int)> myTask= move(allTasks.front());
    allTasks.pop_front();
    thread sumThread(move(myTask), begin, end);
    begin= end;
    end += increment;
    sumThread.detach();
}
auto sum= sum1.get() + sum2.get();
cout << sum;                                // 50005000

```

The promises (std::packaged_task) are moved into the std::deque allTasks. The program iterates in the while loop through all promises. Each promise runs in its thread and performs its addition in the background (sumThread.detach()). The result is the sum of all numbers from 1 to 100000.

std::promise and std::future

The pair std::promise and std::future give the full control over the task.

The methods of the promise prom

Method	Description
<code>prom.swap(prom2)</code> and <code>std::swap(prom, prom2)</code>	Swaps the promises.
<code>prom.get_future()</code>	Returns the future.
<code>prom.set_value(val)</code>	Sets the value.
<code>prom.set_exception(ex)</code>	Sets the exception.
<code>prom.set_value_at_thread_exit(val)</code>	Stores the value and makes it ready if the promise exits.
<code>prom.set_exception_at_thread_exit(ex)</code>	Stores the exception and makes it ready if the promise exits.

If the promise sets the value or the exception more then once a `std::future_error` exception is thrown.

Method	Description
<code>The methods of the future fut</code>	
<code>fut.share()</code>	Returns a <code>std::shared_future</code> .
<code>fut.get()</code>	Returns the result which can be a value or an exception.
<code>fut.valid()</code>	Checks if the result is available. Returns after the call <code>fut.get()</code> false.
<code>fut.wait()</code>	Waits for the result.
<code>fut.wait_for(relTime)</code>	Waits for a time duration for the result.
<code>fut.wait_until(absTime)</code>	Waits until a time for the result.

If a future `fut` asks more than once for the result, a `std::future_error` exception is thrown. The future creates a shared future by the call `fut.share()`. Shared future are associated with their promise and can independently ask for the result. A shared future has the same interface as a future.

Here is the usage of promise and future.

Promise and future

```
// promiseFuture.cpp
...
#include <future>
...

void product(std::promise<int>&& intPromise, int a, int b){
    intPromise.set_value(a*b);
}

int a= 20;
int b= 10;

std::promise<int> prodPromise;
std::future<int> prodResult= prodPromise.get_future();

std::thread prodThread(product, std::move(prodPromise), a, b);
std::cout << "20*10= " << prodResult.get();           // 20*10= 200
```

The promise `prodPromise` is moved into a separate thread and performs its calculation. The future gets the result by `prodResult.get()`.



Use promise and future for the synchronisation of threads

A future `fut` can be synchronised with its associated promise by the call `fut.wait()`. Contrary to [condition variables](#), you need no locks and mutexes, and spurious and lost wakeups are not possible.

Tasks for synchronisation

```
// promiseFutureSynchronise.cpp
...
#include <future>
...

void doTheWork(){
    std::cout << "Processing shared data." << std::endl;
}

void waitingForWork(std::future<void>&& fut){
    std::cout << "Worker: Waiting for work." <<
    std::endl;
    fut.wait();
    doTheWork();
    std::cout << "Work done." << std::endl;
}

void setDataReady(std::promise<void>&& prom){
    std::cout << "Sender: Data is ready." <<
    std::endl;
    prom.set_value();
}

std::promise<void> sendReady;
auto fut= sendReady.get_future();

std::thread t1(waitingForWork, std::move(fut));
std::thread t2(setDataReady, std::move(sendReady));
```

The call of the promise `prom.set_value()` wakes up the future which then can perform its work.



20. Coroutines

We don't get with C++20 concrete coroutines; we get with C++20 a framework for implementing coroutines. To use concrete coroutines, you have to wait for C++23 or use the [cppcoro](#) library from Lewis Baker. Consequently, this chapter gives you only a rough idea of coroutines.

Coroutines are functions that can suspend and resume their execution while keeping their state. To achieve this, a coroutine consists of three parts: the promise object, the coroutine handle, and the coroutine frame.

- The promise object is manipulated from inside the coroutine, and it delivers its result via the promise object.
- The coroutine handle is a non-owning handle to resume or to destroy the coroutine frame from outside.
- The coroutine frame is an internal, typically heap-allocated state. It consists of the already mentioned promise object, the copied parameters of the coroutine, the representation of the suspensions point, local variables which lifetime ends before the current suspension point, and local variables which lifetime exceeds the current suspension point.

Two requirements are necessary to optimise out the allocation of the coroutine:

A function that uses the keywords `co_return` instead of `return`, `co_yield`, or `co_await` becomes a coroutine implicitly.

The new keywords extend the execution of C++ functions with two new concepts.

`co_yield` expression

Allows it to write a generator function. The generator function returns a new value each time. A generator function is a kind of data stream from which you can pick values. The data stream can be infinite.

A generator

```
Generator<int> getNext(int start = 0, int step = 1) noexcept {
    auto value = start;
    for (int i = 0;; ++i){
```

```
        co_yield value;
        value += step;
    }
}
```

co_await expression

Suspends and resumes 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. The expression has to be a so-called awaitable expression and support the following three functions `await_ready`, `await_suspend`, and `await_resume`.

A coroutine

```
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);
}
```

The awaitable expression `expr` in the `co_await expr` has to implement the functions `await_ready`, `await_suspend`, and `await_resume`.

Awaitables

The C++20 standard already defines two awaitables as basic-building blocks: `std::suspend_always`, and `std::suspend_never`.

The predefined awaitables

```
struct suspend_always {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};

struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

An Infinite Data Stream with `co_yield`

The following program 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>

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() {
            return std::suspend_always{};
        }
        auto final_suspend() {
            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);
        }
    } current_value;
};
```

```

};

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

int main() {
    std::cout << std::endl;

    std::cout << "getNext():";
    auto gen = getNext();
    for (int i = 0; i <= 10; ++i) {
        gen.next();
        std::cout << " " << gen.getValue(); // (7)
    }

    std::cout << std::endl;
}

```

The `main` program creates a coroutines. The coroutine `gen` (7) returns the values from 0 to 10.



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, suspends always.
5. asks for the next value and returns if the generator is consumed
6. triggered by the `co_yield` call. The next value is afterwards available.
7. gets the next value

In additional iterations, only the steps 5 and 6 are performed.

Index

Entries in capital letters stand for sections and subsections.

A

abs	assign (vector)
absolute path	Assign and swap
absolute	Associative Containers
accumulate	async
acons	at (string)
acos	atan2
acquire	atan
add const	atanh
add cv	ate (ios)
add lvalue reference	Atomic Data Types
add pointer	auto_ptr
add rvalue reference	await_ready
add volatile	await_resume
adjacent_difference	await_suspend
adjacent_find	Awaitables
Algorithms (Standard Template Library)	B
all of	back (queue)
all views	back (span)
An Infinite Data Stream with co_yield	back (string)
any	back_inserter (insert iterator)
any cast	bad
any of	bad_any_cast
app (ios)	bad_optional_access
append (string)	bad_variante_access (variant)
Arguments (thread)	badbit (ios)
Arrays	barrier
arrive	basic_ios<>
arrive_and_drop	basic_iostream<>
	basic_istream<>

<u>arrive_and_wait (barrier)</u>	<u>basic_istream_view</u>
<u>arrive_and_wait (latch)</u>	<u>basic_ostream<></u>
<u>asin</u>	<u>basic_streambuf</u>
<u>asinh</u>	<u>before_begin (forward_list)</u>
<u>assign (string)</u>	<u>beg (ios)</u>

[begin \(container\)](#) [begin \(unordered associative container\)](#) [bidirectional iterator](#) [binary \(ios\)](#) [binary_search](#) [binary_semaphore](#) [bind](#) [bind_front](#) [bit_and](#) [bit_or](#) [bit_xor](#) [boolalpha](#)

C

[c_str \(string\)](#) [calendar](#) [call_once](#) [callable unit](#) [Callable Units](#) [callable](#) [canonical_path](#) [canonical](#) [capacity \(vector\)](#) [cbegin \(container\)](#) [cbegin \(unordered associative container\)](#) [ceil](#) [cend \(container\)](#) [cend \(unordered associative container\)](#) [cerr](#) [cin](#) [Classes \(filesystem\)](#) [clear \(string\)](#) [clear \(vector\)](#) [cmp_not_equal](#) [co_await](#) [co_yield](#) [common_type](#) [common_views](#) [Compare \(container\)](#) [complexity](#) [Condition Variables](#) [conditional](#) [Constant Expressions](#) [contains \(associative container\)](#) [Conversion Between C++ and C Strings](#) [Coordination Types](#) [copy \(algorithms\)](#) [copy \(filesystem\)](#) [copy \(string\)](#) [copy constructor \(container\)](#) [copy_backward](#) [copy_file](#) [copy_if](#) [copy_n](#) [copy_options](#) [copy_symlink](#) [Coroutines](#) [cos](#) [cosh](#) [count \(algorithms\)](#) [count \(sorted associative container\)](#) [count_down](#) [count_if](#) [counting semaphores](#) [cout](#)

clear	crbegin (container)
clock	crbegin (ordered associative container)
clock_cast	Create and Delete (string)
clog	Create and Delete
close	Create and Initialise (string_view)
cmp_equal	create_directories
cmp_greater	create_directory
cmp_greater_equal	create_hard_link
cmp_less	create_symlink
cmp_less_equal	cref

[cregex_iterator](#)
[cregex_token_iterator](#)
[crend \(container\)](#)
[crend \(ordered associative container\)](#)
[cur \(ios\)](#)
[current_path](#)
[current_zone](#)
[Cyclic References](#)
D
[data \(span\)](#)
[data \(string\)](#)
[Data Race](#)
[day \(calendar\)](#)
[deadlock](#)
[dec](#)
[default constructor \(container\)](#)
[default_random_engine](#)
[defaultfloat](#)
[Deques](#)
[detach](#)
[Direct on the Containers \(ranges\)](#)
[directory](#)
[directory_entry](#)
[directory_iterator](#)
[directory_options](#)
[div](#)
[divides](#)
[drop_views](#)
[drop_while_views](#)
E
[e_v](#)
[egamma_v](#)
[elements_views](#)
[empty \(span\)](#)
[enable_if](#)
[enable_shared_from_this](#)
[end \(container\)](#)
[end \(ios\)](#)
[end \(unordered associative container\)](#)
[endl](#)
[ends_with \(string\)](#)
[eof](#)
[eofbit \(ios\)](#)
[equal](#)
[equal_range \(algorithms\)](#)
[equal_range \(sorted associative container\)](#)
[equal_to](#)
[equivalent](#)
[erase \(associative container\)](#)
[erase \(std\)](#)
[erase \(string\)](#)
[erase \(vector\)](#)
[erase-remove idiom](#)
[erase_after \(forward_list\)](#)
[erase_if \(std\)](#)
[exclusive_scan](#)
[exists](#)
[exp](#)
[expired \(weak_ptr\)](#)
F
[fabs](#)
[fail](#)
[failbit \(ios\)](#)
[FIFO](#)
[File Streams](#)
[File Types](#)

emplace (any)	file
emplace (associative container)	file_size
emplace (optional)	file_status
emplace (variant)	file_time_type
emplace (vector)	file_type
emplace_after (forward_list)	filebuf
emplace_back (vector)	filesystem_error
emplace_front (forward_list)	fill
empty (container)	fill_n

[filter_views](#) [generate](#)
[find \(algorithms\)](#) [generate_n](#)
[find \(sorted associative container\)](#) [get \(input and output streams\)](#)
[find \(string\)](#) [get \(shared_ptr\)](#)
[find_first_not_of \(string\)](#) [get \(std\)](#)
[find_first_of \(algorithm\)](#) [get \(unique_ptr\)](#)
[find_first_of \(string\)](#) [get \(variant\)](#)
[find_if](#) [get_deleter \(shared_ptr\)](#)
[find_if_not](#) [get_deleter \(unique_ptr\)](#)
[find_last_not_of \(string\)](#) [get_future](#)
[find_last_of \(string\)](#) [get_id \(this_thread\)](#)
[findend](#) [get_id \(thread\)](#)
[first \(span\)](#) [get_if \(variant\)](#)
[floor](#) [get_token \(stop_source\)](#)
[flush](#) [getline \(input and output streams\)](#)
[fmod](#) [getline \(string\)](#)
[for_each](#) [good](#)
[format \(formatting library\)](#) [goodbit \(ios\)](#)
[format \(regex\)](#) [greater](#)
[Format Specifier](#) [greater_equal](#)
[format_to \(formatting library\)](#) [H](#)
[format_to_n \(formatting library\)](#) [hard link](#)
[Formatted Input \(streams\)](#) [hard_link_count](#)
[formatter](#) [hardwareConcurrency \(thread\)](#)
[Formatting Library](#) [hasUniqueObjectRepresenation](#)
[forward iterator](#) [hasValue \(any\)](#)
[Forward lists](#) [hasValue \(optional\)](#)
[forward](#) [hasVirtualDestructor](#)
[forwarding reference](#) [hex](#)
[frexp](#) [hexfloat](#)
[front \(queue\)](#) [Hierarchy \(input and output streams\)](#)
[front \(span\)](#) [high_resolution_clock](#)
[front \(string\)](#) [holdsAlternative \(variant\)](#)
[front_inserter \(insert iterator\)](#) [hours \(time_of_day\)](#)
[fstream](#) [I](#)

Function composition (ranges)	ifstream
function	ignore (input stream)
Functions object	ignore (std)
Functions	in (ios)
future	includes
G	inclusive_scan
gcount	index (variant)

inner_product	is_fifo
inplace_merge	is_final
Input(streams)	is_floating_point
Input and Output(string)	is_function
Input and Output Functions(streams)	is_fundamental
Input and Output Streams	is_heap
input iterator	is_heap_until
insert(associative container)	is_integral
insert(string)	is_lvalue_reference
insert(vector)	is_member_function_pointer
Insert Iterators	is_member_object_pointer
insert_after(forward_list)	is_member_pointer
inserter(insert iterator)	is_move_assignable
Interface Of All Containers	is_move_constructible
internal	is_nothrow_assignable
inv_pi_v	is_nothrow_constructible
inv_sqrt3_v	is_nothrow_copy_assignable
ios_base	is_nothrow_copy_constructible
iota	is_nothrow_default_constructible
is_abstract	is_nothrow_destructible
is_aggregate	is_nothrow_move_assignable
is_am(chrono)	is_nothrow_move_constructible
is_arithmetic	is_nothrow_swappable
is_array	is_nothrow_swappable_with
is_assignable	is_null_pointer
is_base_of	is_object
is_block_file	is_open
is_character_file	is_other
is_class	is_partitioned
is_compound	is_pm(chrono)
is_const	is_pod
is_constant_evaluated	is_pointer
is_constructible	is_polymorphic
is_convertible	is_reference
is_copy_assignable	is_regular_file

[is_copy_constructible](#)
[is_default_constructible](#)
[is_destructible](#)
[is_directory](#)
[is_empty \(filesystem\)](#)
[is_empty \(type traits\)](#)
[is_enum](#)

[is_rvalue_reference](#)
[is_same](#)
[is_scalar](#)
[is_signed](#)
[is_socket](#)
[is_sorted](#)
[is_sorted_until](#)

[is_standard_layout](#) [Lazy evaluation \(ranges\)](#)
[is_swappable](#) [ldexp](#)
[is_swappable_with](#) [ldiv](#)
[is_symlink](#) [leap_second](#)
[is_trivial](#) [left](#)
[is_trivially_assignable](#) [lerp](#)
[is_trivially_constructible](#) [less](#)
[is_trivially_copy_assignable](#) [less_equal](#)
[is_trivially_copy_constructible](#) [lexicographical_compare](#)
[is_trivially_copyable](#) [Lifetime \(thread\)](#)
[is_trivially_default_constructible](#) [LIFO](#)
[is_trivially_destructible](#) [Lists](#)
[is_trivially_move_assignable](#) [llabs](#)
[is_trivially_move_constructible](#) [lldiv](#)
[is_union](#) [ln10_v](#)
[is_unsigned](#) [ln2_v](#)
[is_void](#) [local_info \(time zone\)](#)
[is_volatile](#) [locate_zone](#)
[istream](#) [lock \(unique_lock\)](#)
[istream_view](#) [lock \(weak_ptr\)](#)
[istringstream](#) [lock_guard](#)
[Iterators categories](#) [log10](#)
[Iterators](#) [log10e_v](#)
J [log2e_v](#)
[join](#) [log](#)
[join_views](#) [logical_and](#)
[joinable](#) [logical_not](#)
[jthread](#) [logical_or](#)
K [lower_bound \(binary_search\)](#)
[key \(ordered associative container\)](#) [lower_bound \(sorted associative container\)](#)
[key \(unordered associative container\)](#) [lowercase](#)
[Keys and Value](#) **M**
[keys_views](#) [make12 \(time_of_day\)](#)

L	make24(time_of_day)
labs	make_any
Lambda Functions	make_heap
last(span)	make_optional
last_spec(calendar)	make_pair
last_write_time	make_shared
latch	make_signed
launch::async	make_tuple
launch::deferred	make_unique

[make_unsigned](#)
[match \(regex\)](#)
[match_results \(regex\)](#)
[Mathematical Constants](#)
[max \(counting_semaphore\)](#)
[max \(std\)](#)
[max_element](#)
[memory model](#)
[merge \(forward_list\)](#)
[merge \(list\)](#)
[merge](#)
[Mersenne Twister](#)
[midpoint](#)
[min \(std\)](#)
[min_element](#)
[minmax \(algorithm\)](#)
[minmax \(std\)](#)
[minus](#)
[minutes \(time_of_day\)](#)
[mismatch](#)
[modf](#)
[Modifying Operatios](#)
[modulus](#)
[month \(calendar\)](#)
[month_day \(calendar\)](#)
[month_day_last \(calendar\)](#)
[month_weekday \(calendar\)](#)
[month_weekday_last \(calendar\)](#)
[move constructor \(container\)](#)
[move](#)
[multimap](#)
[multiplies](#)
[multiset](#)
[Multithreading](#)
[mutex](#)
[none_of](#)
[noshowbase](#)
[noshowpos](#)
[not_equal_to](#)
[notify_all](#)
[notify_one](#)
[nth_element](#)
[Numeric Conversion](#)
[Numeric functions from C](#)
[Numeric](#)
O
[oct](#)
[ofstream](#)
[once_flag](#)
[open](#)
[operations \(thread\)](#)
[optional](#)
[Ordered Associative Containers](#)
[ostream](#)
[ostringstream](#)
[osyncstream](#)
[out \(ios\)](#)
[Output \(streams\)](#)
[output iterator](#)
[owns_lock \(unique_lock\)](#)
P
[packaged_task](#)
[pair](#)
[par \(execution\)](#)
[par_unseq \(execution\)](#)
[parse \(chrono\)](#)
[partial_sort](#)
[partial_sort_copy](#)
[partial_sum](#)
[partition](#)

Mutexes	partition_copy
N	partition_point
negate	path (classes)
next_permutation	path (filesystem)
noboolalpha	path name
Non-member Functions (filesystem)	peek
Non-modifying operations (string_view)	Performance associative container

[perm_options](#)
[permissions](#)
[perms](#)
[phi_v](#)
[pi_v](#)
[plus](#)
[pop \(priority_queue\)](#)
[pop \(queue\)](#)
[pop \(stack\)](#)
[pop_back \(string\)](#)
[pop_back \(vector\)](#)
[pop_heap](#)
[pow](#)
[predicate \(callable units\)](#)
[predicate \(utilities\)](#)
[prev_permutation](#)
[promise](#)
[proximate](#)
[push \(priority_queue\)](#)
[push \(queue\)](#)
[push \(stack\)](#)
[push_back \(string\)](#)
[push_back \(vector\)](#)
[push_heap](#)
[putback](#)
R
[RAII](#)
[rand](#)
[Random Access \(filestreams\)](#)
[random access iterator](#)
[Random Number Distribution](#)
[Random Number Generator](#)
[Random Number](#)
[random_device](#)
[random_shuffle](#)
[read_symlink](#)
[recursive_directory_iterator](#)
[recursive_mutex](#)
[recursive_timed_mutex](#)
[reduce](#)
[ref](#)
[ref_views](#)
[Reference Wrappers](#)
[regex_iterator](#)
[regex_token_iterator](#)
[Regular Expression Objects](#)
[relative path](#)
[relative](#)
[release \(counting_semaphore\)](#)
[release \(shared_ptr\)](#)
[release \(unique_lock\)](#)
[release \(unique_ptr\)](#)
[remove \(algorithm\)](#)
[remove \(filesystem\)](#)
[remove \(list\)](#)
[remove_all](#)
[remove_const](#)
[remove_copy](#)
[remove_copy_if](#)
[remove_cv](#)
[remove_if \(algorithm\)](#)
[remove_if \(list\)](#)
[remove_pointer](#)
[remove_prefix \(string_view\)](#)
[remove_reference](#)
[remove_suffix \(string_view\)](#)
[remove_volatile](#)
[rename](#)
[rend \(container\)](#)
[rend \(ordered associative container\)](#)

Range (ranges)	Repeated Search (regex)
range constructor (container)	replace (algorithm)
Ranges	replace (regex)
ratio	replace (string)
rbegin (container)	replace_copy
rbegin (ordered associative container)	replace_copy_if
rdstate	replace_if

request_stop (stop_source)	setw
reserve (vector)	share
reset (any)	Shared Variable (thread)
reset (optional)	shared_from_this
reset (shared_ptr)	shared_lock
reset (weak_ptr)	shared_ptr
resize (vector)	shared_timed_mutex
resize_file	showbase
reverse	showpos
reverse_copy	shrink_to_fit (vector)
reverse_views	shuffle
rfind (string)	sin
right	sinh
root-directory	Size (container)
root-name	size (span)
rotate	size (vector)
rotate_copy	Size versus Capacity
S	size_bytes (span)
search (algorithm)	sleep_for (this_thread)
search (regex)	sleep_until (this_thread)
Search	Smart Pointers
search_n	sort
seconds (time_of_day)	sort_heap
seed	Space Information on the Filesystem
seekg	space
seekp	space_info
Semaphores	span
seq (execution)	splice (list)
sequence constructor (container)	splice_after (forward_list)
Sequence Containers	split_views
sequential consistency	sqrt2_v
set	sqrt3_v
set_difference	sqrt
set_exception	srand
set_exception_at_thread_exit	sregex_iterator

set_intersection	sregex_token_iterator
set_symmetric_difference	stable
set_union	stable_partition
set_value	stable_sort
set_value_at_thread_exit	Stack
setfill	starts_with (string)
setstate	State of a Stream

Static Variables	swap (string_view)
status	swap (unique_ptr)
status_known	swap (weak_ptr)
stderr	swap_ranges
stdin	symbolic link
stdout	symlink_status
steady_clock	sys_info (time zone)
stod (string)	system_clock
stof (string)	T
stoi (string)	take_views
stol (string)	take_while_views
stold (string)	tan
stoll (string)	tanh
Stop Token	Tasks
stop_callback	tellg
stop_possible (stop_source)	tellp
stop_possible (stop_token)	temp_directory_path
stop_requested (stop_source)	Thread Local Data
stop_requested (stop_token)	thread safe initialization
stop_source	thread
stop_token	Threads versus tasks
stoul (string)	Threads
stoull (string)	tie (std)
Stream Iterators	time duration
Streams	time library
strict weak ordering	time of day
string literal (C)	time point
string literal (C++)	Time zone
String streams	time_of_day (chrono)
String	time_zone
string	timed_mutex
string_view	to_duration (time_of_day)
stringstream	to_string (string)
sub_match (match_results)	to_wstring (string)
subseconds (time_of_day)	top (priority_queue)

[subspan \(span\)](#)
[suspend_always](#)
[suspend_never](#)
[swap \(algrorithm\)](#)
[swap \(shared_ptr\)](#)
[swap \(std\)](#)
[swap \(string\)](#)

[top \(stack\)](#)
[transform_exclusive_scan](#)
[transform_inclusive_scan](#)
[transform_reduce](#)
[transform_views](#)
[Transfrom Ranges](#)
[trunc \(ios\)](#)

[try_acquire](#) [value \(unordered associative container\)](#)
[try_acquire_for](#) [value_by_exception \(variant\)](#)
[try_acquire_until](#) [value_or \(optional\)](#)
[try_lock \(unique_lock\)](#) [values_view](#)
[try_lock_for \(unique_lock\)](#) [variant](#)
[try_lock_until \(unique_lock\)](#) [Vectors](#)
[try_wait](#) [View \(ranges\)](#)
[Tuples](#) [Views on Contiguous Sequences](#)
[type \(any\)](#) [views::all](#)
[Type Traits](#) [views::common](#)
[tzdb](#) [views::drop](#)
U [views::drop_while](#)
[u16string](#) [views::elements](#)
[u16string_view](#) [views::filter](#)
[u32string](#) [views::iota](#)
[u32string_view](#) [views::join](#)
[Unformatted Input \(streams\)](#) [views::keys](#)
[unget](#) [views::reverse](#)
[unique \(forward_list\)](#) [views::split](#)
[unique \(list\)](#) [views::take](#)
[unique \(shared_ptr\)](#) [views::take_while](#)
[unique \(std\)](#) [views::transform](#)
[unique_copy](#) [views::values](#)
[unique_lock](#) [visit](#)
[unique_ptr](#) **W**
[universal reference](#) [wait \(barrier\)](#)
[Unordered Associative Containers](#) [wait \(condition_variable\)](#)
[unordered_map](#) [wait \(future\)](#)
[unordered_multimap](#) [wait \(latch\)](#)
[unordered_multiset](#) [wait_for \(condition_variable\)](#)
[unordered_set](#) [wait_for \(future\)](#)
[upper_bound \(sorted associative container\)](#) [wait_until \(condition_variable\)](#)
[upper_bound](#) [wait_until \(future\)](#)

<u>uppercase</u>	<u>wcerr</u>
<u>use_count (shared_ptr)</u>	<u>wcin</u>
<u>use_count (weak_ptr)</u>	<u>wclog</u>
<u>User-defined Data Types</u>	<u>wcout</u>
<u>Utilities</u>	<u>wcregex_iterator</u>
V	<u>wcregex_token_iterator</u>
<u>valid</u>	<u>weak_ptr</u>
<u>value (optional)</u>	<u>weakly_canonical</u>
<u>value (ordered associative container)</u>	<u>weekday (calendar)</u>

[weekday_indexed \(calendar\)](#)
[weekday_last \(calendar\)](#)
[wfilebuf](#)
[wfstream](#)
[wifstream](#)
[wistringstream](#)
[wofstream](#)
[wostream](#)
[wosyncstream](#)
[ws](#)
[wsregex_iterator](#)
[wsregex_token_iterator](#)
[wstring](#)
[wstring_view](#)
[wstringstream](#)
Y
[year \(calendar\)](#)
[year_month \(calendar\)](#)
[year_month_day \(calendar\)](#)
[year_month_day_last \(calendar\)](#)
[year_month_day_weekday \(calendar\)](#)
[year_month_weekday_last \(calendar\)](#)
[yield \(this_thread\)](#)
Z
[zoned_time](#)