Concurrency with Modern C++

What every professional C++ programmer should know about concurrency.

packaged_task
relaxed
atomic_noexcept
shared_lock
atomic_cancel
define_task_block
C++20
future
fence
co_yield
steady_clock
Parallel
C++11
then
async
make_ready_future
when_all
unique_lock
atomics
co_await
when_any
threads
barrier
mutex
latch
C++14
tasks
C++17
flex_barrier
synchronized
system_clock
atomic_commit
atomic_weak_ptr
atomic_shared_ptr
make_exeption_future
acquire-release
high_resolution_clock
sequential
execution::seq
execution::par
thread_local

Rainer
Grimm

ModernesCpp.com
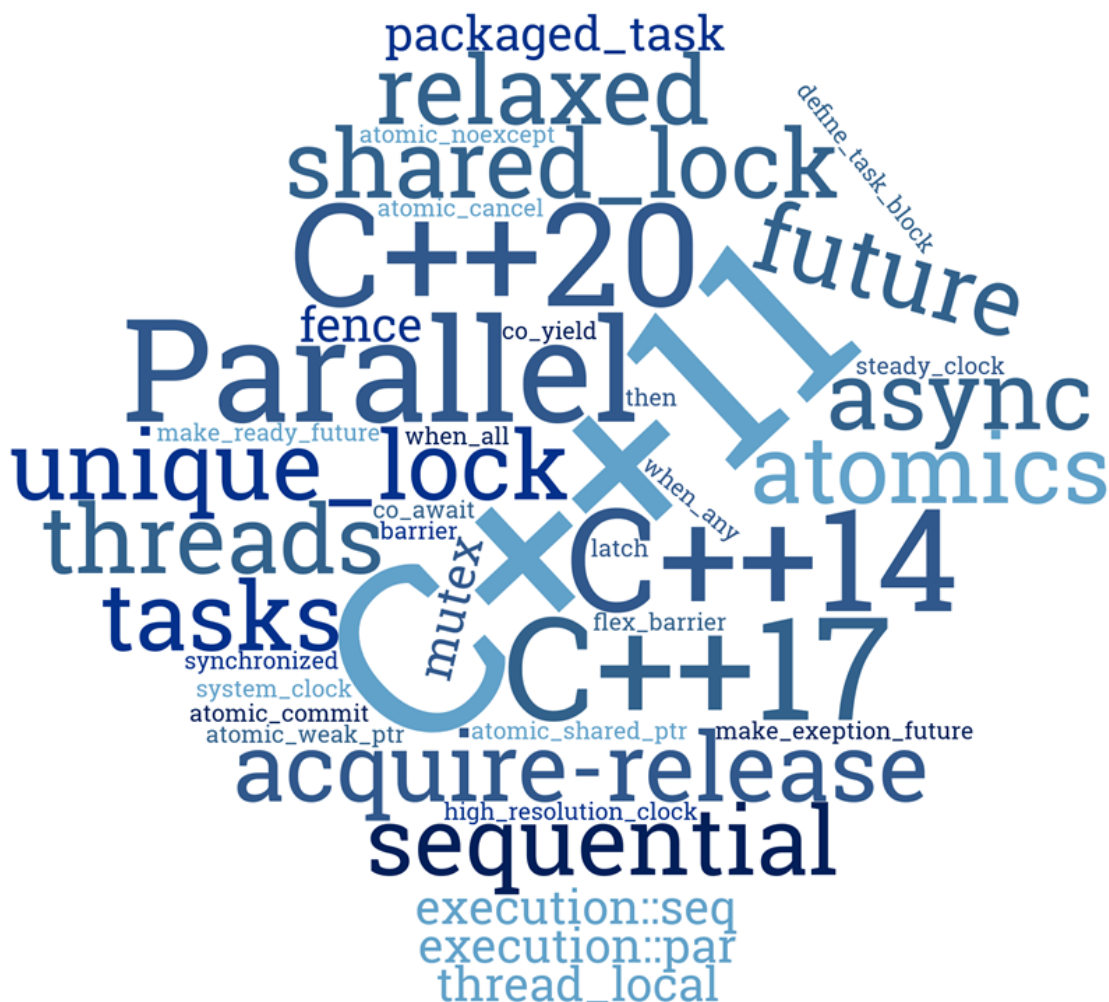
Concurrency with Modern C++

What every professional C++ programmer should know about concurrency.

Rainer Grimm

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

This version was published on 2022-07-30

Leanpub

Tweet This Book!

Please help Rainer Grimm by spreading the word about this book on Twitter!

The suggested hashtag for this book is #rainer_grimm.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#rainer_grimm

Contents

Patterns 448

Reader Testimonials

Bart Vandewoestyne

Senior Development Engineer Software at Esterline

"'Concurrency with Modern C++' is your practical guide to getting familiar with concurrent programming in Modern C++. Starting with the C++ Memory Model and using many ready-to-run code examples, the book covers a good deal of what you need to improve your C++ multithreading skills. Next to the enlightening case studies that will bring you up to speed, the overview of upcoming concurrency features might even wet your appetite for more!"

Ian Reeve

Senior Storage Software Engineer for Dell Inc.

"Rainer Grimm's Concurrency with Modern C++ is a well written book covering the theory and practice for working with concurrency per the existing C++ standards, as well as addressing the potential changes for the upcoming C++ 20 standard. He provides a conversational discussion of the applications and best practices for concurrency along with example code to reinforce the details of each topic. An informative and worthwhile read!"

Robert Badea
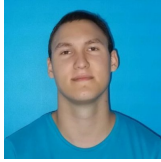
Technical Team Leader

"Concurrency with Modern C++ is the easiest way to become an expert in the multithreading environment. This book contains both simple and advanced topics, and it has everything a developer needs, in order to become an expert in this field: Lots of content, a big number of running code examples, along with great explanation, and a whole chapter for pitfalls. I enjoyed reading it, and I highly recommend it for everyone working with C++."

Introduction

Concurrency with Modern C++ is a journey through the present and upcoming concurrency features in C++.

- **C++11** and **C++14** have the basic building blocks for creating concurrent and parallel programs.
- With **C++17** we have the parallel algorithms from the Standard Template Library (STL). That means that most STL-based algorithms can be executed sequentially, parallel, or vectorized.
- The concurrency story in C++ goes on. With **C++20/23**, we can hope for extended futures, coroutines, transactions, and more.

This book explains the details of concurrency in modern C++ and gives you, also, many code examples. Consequently, you can combine theory with practice to get the most out of it.

Because this book is about concurrency, I'd like to present many pitfalls and show you how to overcome them.

Conventions

Only a few conventions.

Special Fonts

Italic: I use *Italic* to emphasize an expression.

Bold: I use **Bold** to emphasize even more.

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

Special Symbols

$\Rightarrow$ stands for a conclusion in the mathematical sense. For example, a $\Rightarrow$ b means if a then b.

Special Boxes

Boxes contain tips, warnings, and distilled information.

Tip Headline

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

Warning Headline

Warning boxes should help you to avoid pitfalls.

Distilled Information

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

Source Code

All source code examples are complete. That means, assuming you have a conforming compiler, you can compile and run them. The name of the source file is in the title of the listing. I use the `using namespace std` directive in the source files only if necessary.

Run the Programs

Compiling and running the examples is quite easy for the C++11 and C++14 examples in this book. Every modern C++ compiler should support them. For the GCC[1] and the clang[2] compiler, the C++ standard must be specified as well as the threading library. For example the g++ compiler from GCC creates an executable program called `thread` with the following command-line: `g++ -std=c++14 -pthread thread.cpp -o thread`.

- **-std=c++14**: use the language standard C++14
- **-pthread**: add support for multithreading with the pthread library
- **thread.cpp**: source file
- **-o thread**: executable program

The same command-line holds for the clang++ compiler. The Microsoft Visual Studio 17 C++ compiler supports C++14 as well.

If you have no modern C++ compiler at your disposal, there are many online compilers available. Arne Mertz' blog post C++ Online Compiler[3] gives an excellent overview.

With C++17 and C++20/23, the story becomes quite complicated. I installed the HPX (High Performance ParalleX)[4] framework, which is a general-purpose C++ runtime system for parallel and distributed applications of any scale. HPX has already implemented the Parallel STL of C++17 and many of the concurrency features of C++20 and C++23.

---

[1]https://gcc.gnu.org/
[2]https://clang.llvm.org/
[3]https://arne-mertz.de/2017/05/online-compilers/
[4]http://stellar.cct.lsu.edu/projects/hpx/

How should you read the book?

If you are not very familiar with concurrency in C++, start at the very beginning with A Quick Overview to get the big picture.

Once you get the big picture, you can proceed with the The Details. Skip the memory model in your first iteration of the book unless you are entirely sure that is what you are looking for. The chapter Case Studies should help you apply the theory. This is quite challenging as it requires a basic understanding of the memory model.

The chapters about The Near Future: C++20 and the The Future: C++23 are optional. I am very curious about the future. I hope you are too!

The last part, Further Information provides you with additional guidance towards a better understanding of my book and, finally, getting the most out of it.

Personal Notes

Acknowledgment

I started a request in my English blog to write this book in English: www.ModernesCpp.com[5] I received a much higher response than I expected. About 50 people wanted to proofread my book. Special thanks to all of you, including my daughter Juliette, who improved my layout, and my son Marius as the first proofreader.

Here are the names in alphabetic order: Nikos Athanasiou, Robert Badea, Joe Das, Jonas Devlieghere, Randy Hormann, László Kovacs, Lasse Natvig, Erik Newton, Ian Reeve, Bart Vandewoestyne, Vadim Vinnik, Dafydd Walters, Andrzej Warzynski, and Enrico Zschemisch.

About Me

I've worked as a software architect, team leader, and instructor for about 20 years. I enjoy writing articles on C++, Python, and Haskell and speaking at conferences in my spare time. In 2016 I decided to work for myself. I organize and lead seminars about modern C++ and Python.

My Special Circumstances

I began to write this book Concurrency With Modern C++ in Oberstdorf while getting a new hip joint. Formally, it was a total endoprosthesis of my left hip joint. I wrote the first half of this book during my stay in the clinic and the rehabilitation clinic. Honestly, writing a book helped me a lot during this challenging period.

---

[5]http://www.modernescpp.com/index.php/looking-for-proofreaders-for-my-new-book-concurrency-with-modern-c

A Quick Overview

1. Concurrency with Modern C++

With the publishing of the C++11 standard, C++ got a multithreading library and a memory model. This library has basic building blocks like atomic variables, threads, locks, and condition variables. That's the foundation on which future C++ standards such as C++20 and C++23 can establish higher abstractions. However, C++11 already knows tasks, which provide a higher abstraction than the cited basic building blocks.
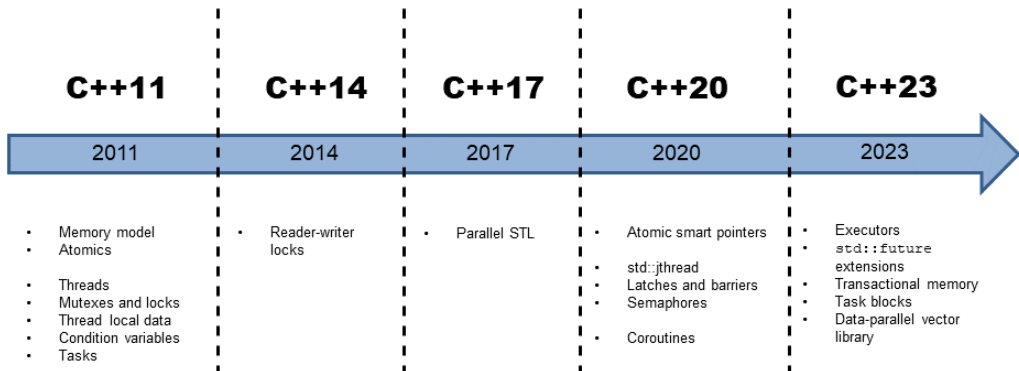
Concurrency in C++

Roughly speaking, you can divide the concurrency story of C++ into three evolution steps.

1.1 C++11 and C++14: The Foundation

C++11 introduced multithreading. The multithreading support consists of a *well-defined* memory model and a standardized threading interface. C++14 added reader-writer locks to the multithreading facilities of C++.

1.1.1 Memory Model
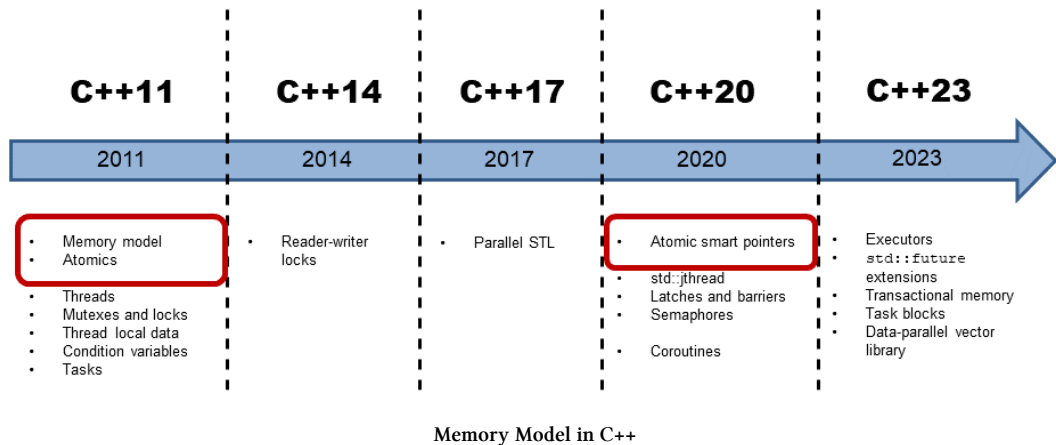
Memory Model in C++

The foundation of multithreading is a *well-defined* memory model. This memory model has to deal with the following aspects:

- Atomic operations: operations that are performed without interruption.
- Partial ordering of operations: a 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 is inspired by its predecessor: the Java memory model. Unlike the Java memory model, however, C++ allows us to break the constraints of sequential consistency, which is the default behavior of atomic operations.

Sequential consistency provides two guarantees.

1. The instructions of a program are executed in source code order.
2. There is a global order for all operations on all threads.

The memory model is based on atomic operations on atomic data types (short atomics).

1.1.1.1 Atomics

C++ has a set of simple atomic data types. These are booleans, characters, numbers, and pointers in many variants. You can define your atomic data type with the class template `std::atomic`. Atomics establishes synchronization and ordering constraints that can also hold for non-atomic types.

The standardized threading interface is the core of concurrency in C++.

1.1.2 Multithreading

C++11   C++14   C++17   C++20   C++23

| 2011 | 2014 | 2017 | 2020 | 2023 |

- Memory model
- Atomics
- Threads
- Mutexes and locks
- Thread local data
- Condition variables
- Tasks

- Reader-writer locks

- Parallel STL

- Atomic smart pointers
- std::jthread
- Latches and barriers
- Semaphores
- Coroutines

- Executors
- `std::future` extensions
- Transactional memory
- Task blocks
- Data-parallel vector library

Multithreading in C++

Multithreading in C++ consists of threads, synchronization primitives for shared data, thread-local data, and tasks.

1.1.2.1 Threads

C++ supports two kind of threads: the basic thread `std::thread` (C++11) and the improved thread `std::jthread` (C++20).

1.1.2.1.1 `std::thread`

A `std::thread` represents an independent unit of program execution. The executable unit, which is started immediately, receives its work package as a callable unit. A callable unit can be a named function, a function object, or a lambda function.

The creator of a thread is responsible for its lifecycle. The executable unit of the new thread ends with the end of the callable. Either the creator waits 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 calls `std::terminate` in its destructor, and the program terminates.

A from its creator detached thread is typically called a daemon thread because it runs in the background.

A `std::thread` is a variadic template. This means that it can receive an arbitrary number of arguments; either the callable or the thread can get the arguments.

1.1.2.1.2 `std::jthread` (C++20)

`std::jthread` stands for joining thread. In addition to `std::thread` from C++11, std::jthread automatically joins in it destructor and can cooperatively be interrupted. Consequently, `std::jthread` extends the interface of `std::thread`.

1.1.2.2 Shared Data

You have to coordinate access to a shared variable if more than one thread uses it simultaneously and the variable is mutable (non-const). Reading and writing a shared variable concurrently is a data race and, therefore, undefined behavior. Coordinating access to a shared variable is achieved with mutexes and locks in C++.

1.1.2.2.1 Mutexes

A mutex (*mutual exclusion*) guarantees that only one thread can access a shared variable at any given time. A mutex locks and unlocks the critical section, to which the shared variable belongs. C++ has five different mutexes. They can lock recursively, tentatively, and with or without time constraints. Even mutexes can share a lock at the same time.

1.1.2.2.2 Locks

You should encapsulate a mutex in a lock to release the mutex automatically. A lock implements the RAII idiom by binding a mutex's lifetime to its own. C++ has a `std::lock_guard` / `std::scoped_lock` for the simple, and a `std::unique_lock` / `std::shared_lock` for the advanced use cases such as the explicit locking or unlocking of the mutex, respectively.

1.1.2.2.3 Thread-safe Initialization of Data

If shared data is read-only, it's sufficient to initialize it in a *thread-safe* way. C++ offers various ways to achieve this including using a constant expression, a static variable with block scope, or using the function `std::call_once` in combination with the flag `std::once_flag`.

1.1.2.3 Thread Local Data

Declaring a variable as thread-local ensures that each thread gets its own copy. Therefore, there is no shared variable. The lifetime of thread-local data is bound to the lifetime of its thread.

1.1.2.4 Condition Variables

Condition variables enable threads to be synchronized via messages. One thread acts as the sender while the other acts as the receiver of the message. The receiver blocks waiting for the message from the sender. Typical use cases for condition variables are producer-consumer workflows. A condition variable can be either the sender or the receiver of the message. Using condition variables correctly is quite challenging; therefore, tasks are often the easier solution.

1.1.2.5 Cooperative Interruption (C++20)

The additional functionality of the cooperative interruption of `std::jthread` is based on the `std::stop_source`, `std::stop_token`, and the `std::stop_callback` classes. `std::jthread` and `std::condition_variable_any` support cooperative interruption by design.

1.1.2.6 Semaphores (C++20)

Semaphores are a synchronization mechanism used to control concurrent access to a shared resource. A counting semaphore is a semaphore that has a counter that is bigger than zero. The counter is initialized in the constructor. Acquiring the semaphore decreases the counter, and releasing the semaphore increases the counter. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

1.1.2.7 Latches and Barriers (C++20)

Latches and barriers are coordination types that enable some threads to block until a counter becomes zero. The counter is initialized in the constructor. At first, don't confuse the new barriers with memory barriers, also known as fences. In C++20 we get latches and barriers in two variations: `std::latch`, and `std::barrier`. Concurrent invocations of the member functions of a `std::latch` or a `std::barrier` produce no data race.

1.1.2.8 Tasks

Tasks have a lot in common with threads. While you explicitly create a thread, a task is just a job you start. The C++ runtime automatically handles, such as in the simple case of `std::async`, the task's lifetime.

Tasks are like data channels between two communication endpoints. They enable thread-safe communication between threads. The promise as one endpoint puts data into the data channel, the future at the other endpoint picks the value up. The data can be a value, an exception, or simply a notification. In addition to `std::async`, C++ has the class templates `std::promise` and `std::future` that give you more control over the task.

1.1.2.9 Synchronized Outputstreams (C++20)

With C++20, C++ enables synchronized outputstreams. `std::basic_syncbuf` is a wrapper for a `std::basic_streambuf`[1]. It accumulates output in its buffer. The wrapper sets its content to the wrapped buffer when it is destructed. Consequently, the content appears as a contiguous sequence of characters, and no interleaving of characters can happen. Thanks to `std::basic_osyncstream`, you can directly write synchronously to std::cout.

1.2 C++17: Parallel Algorithms of the Standard Template Library
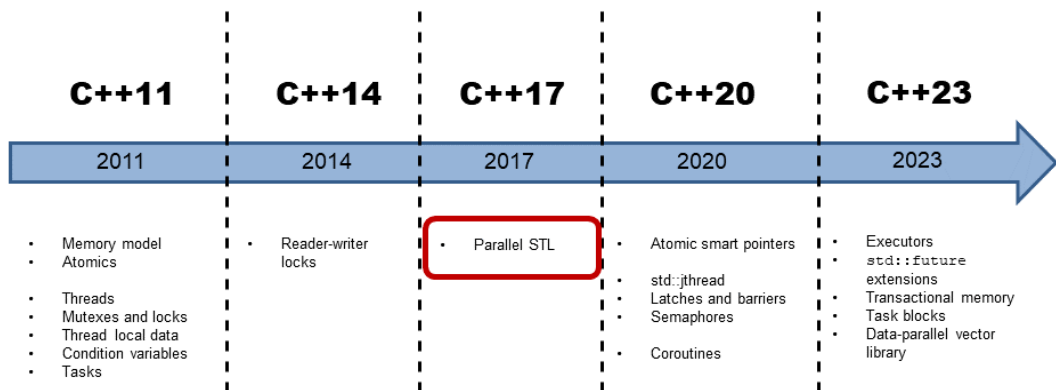
Parallel algorithms in C++17

With C++17, concurrency in C++ has drastically changed, in particular the parallel algorithms of the Standard Template Library (STL). C++11 and C++14 only provide the basic building blocks for concurrency. These tools are suitable for a library or framework developer but not for the application developer. Multithreading in C++11 and C++14 becomes an assembly language for concurrency in C++17!

1.2.1 Execution Policy

With C++17, most of the STL algorithms are available in a parallel implementation. This makes it possible for you to invoke an algorithm with a so-called execution policy. This policy specifies whether the algorithm runs sequentially (`std::execution::seq`), in parallel (`std::execution::par`), or in parallel with additional vectorisation (`std::execution::par_unseq`).

---

[1]https://en.cppreference.com/w/cpp/io/basic_streambuf

1.2.2 New Algorithms

In addition to the 69 algorithms available in overloaded versions for parallel, or parallel and vectorized execution, we get eight additional algorithms. These new ones are well suited for parallel reducing, scanning, or transforming ranges.

1.3 Coroutines
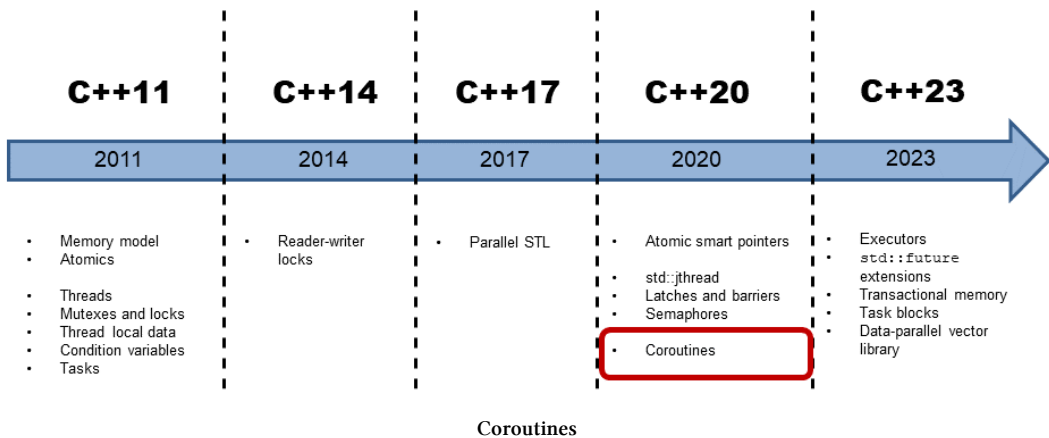
Coroutines

Coroutines are functions that can suspend and resume their execution while maintaining their state. Coroutines are often the preferred approach to implement cooperative multitasking in operating systems, event loops, infinite lists, or pipelines.

1.4 Case Studies

After presenting the theory of the memory model and the multithreading interface, I apply the theory in a few case studies.

1.4.1 Calculating the Sum of a Vector

Calculating the sum of a vector can be done in various ways. You can do it sequentially or concurrently with maximum and minimum sharing of data. The performance numbers differ drastically.

1.4.2 The Dining Philosophers Problem by Andre Adrian

The dining philosophers problem is a classic synchronization problem formulated by Edsger Dijkstra[2] in the article Hierarchical Ordering of Sequential Processes[3]: Five philosophers, numbered from 0 through 4 are living in a house where the table laid for them, each philosopher having his own place at the table. Their only problem - besides those of philosophy - is that the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks. There are two forks next to each plate, so that presents no difficulty: as a consequence, however, no two neighbours may be eating simultaneously.

1.4.3 Thread-Safe Initialization of a Singleton

Thread-safe initialization of a singleton is the classical use-case for thread-safe initialization of a shared variable. There are many ways to do it, with varying performance characteristics.

1.4.4 Ongoing Optimization with CppMem

I start with a small program and successively improve it by weakening the memory ordering. I verify each step of my process of ongoing optimization with CppMem. CppMem[4] is an interactive tool for exploring the behavior of small code snippets using the C++ memory model.

1.4.5 Fast Synchronization of Threads

There are many ways in C++20 to synchronize threads. You can use condition variables, `std::atomic_-flag`, `std::atomic<bool>`, or semaphores. I discuss the performance numbers of various ping-pong games.

1.5 Variations of Futures

Thanks to the new keyword `co_return`, I can implement in the case study variations of futures an eager future, a lazy future, or a future running in a separate thread. Heavily used comments make its workflow transparent.

1.6 Modification and Generalization of a Generator

`co_yield` enables it to create infinite data streams. In the case stucy modification and generalization of a generator, the infinite data streams become finite and generic.

---

[2]https://en.wikipedia.org/wiki/Edsger_W._Dijkstra
[3]https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD310.html
[4]http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/

1.7 Various Job Workflows

The case study of various job workflows presents a few coroutines that are automatically resumed if necessary. `co_await` makes this possible.

1.8 The Future: C++23

Concurrency in C++23
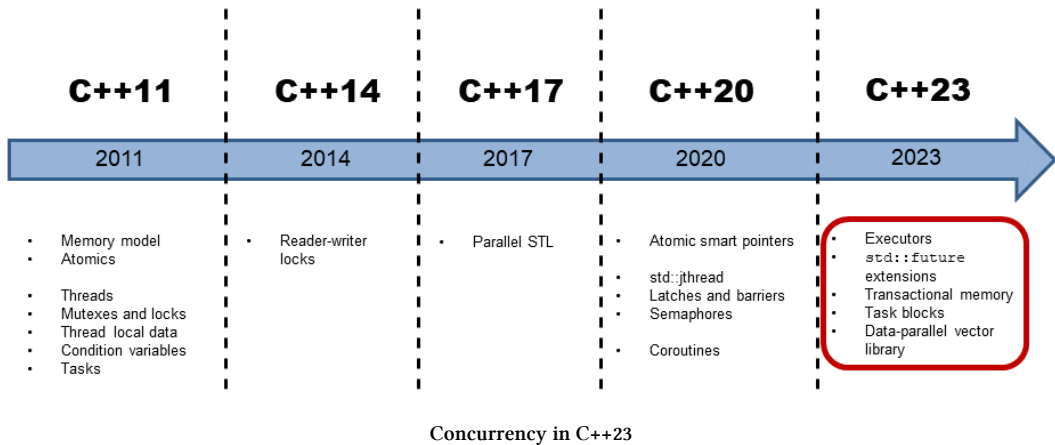
It isn't easy to make predictions, especially about the future (Niels Bohr[5]).

1.8.1 Executors

An executor consists of a set of rules about where, when, and how to run a callable unit. They are the basic building block to execute and specify if callables should run on an arbitrary thread, a thread pool, or even single threaded without concurrency. The extended futures, the extensions for networking N4734[6] depend on them but also the parallel algorithms of the STL, and the new concurrency features in C++20/23 such as latches and barriers, coroutines, transactional memory, and task blocks eventually use them.

1.8.2 Extended futures

Tasks called promises and futures, introduced in C++11, have a lot to offer, but they also have drawbacks: tasks are not composable into powerful workflows. That limitation does not hold for the extended futures in C++23. Therefore, an extended future becomes ready when its predecessor (`then`) becomes ready, `when_any` one of its predecessors becomes ready, or `when_all` of its predecessors becomes ready.

---

[5]https://en.wikipedia.org/wiki/Niels_Bohr
[6]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4734.pdf

1.8.3 Transactional Memory

Transactional memory is based on the ideas underlying transactions in database theory. A transaction is an action that provides the first three properties of ACID database transactions: Atomicity, Consistency, and Isolation. The durability that is characteristic for databases holds not for the proposed transactional memory in C++. The new standard has transactional memory in two flavors: synchronized blocks and atomic blocks. Both are executed in total order and behave as if a global lock protected them. In contrast to synchronized blocks, atomic blocks cannot execute transaction-unsafe code.

1.8.4 Task Blocks

Task Blocks implement the fork-join paradigm in C++. The following graph illustrates the key idea of a task block: you have a fork phase in which you launch tasks and a join phase in which you synchronize them.

Task Blocks
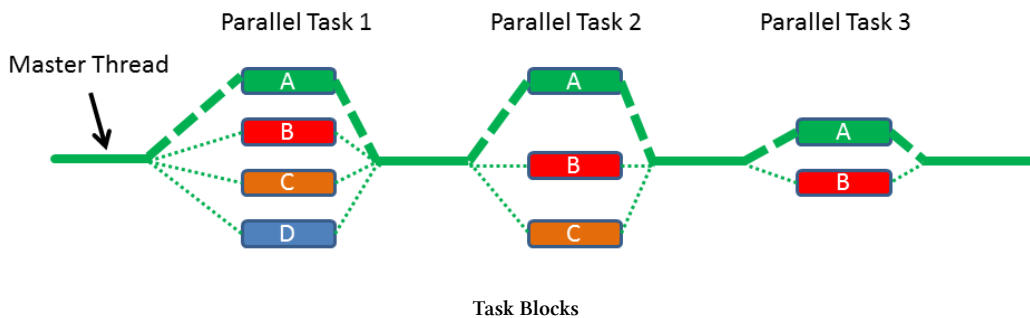
1.8.5 Data-Parallel Vector Library

The data-parallel vector library provides data-parallel (SIMD) programming via vector types. SIMD means that one operation is performed on many data in parallel.

1.9 Patterns and Best Practices

Patterns are documented best practices from the best. They "… express a relation between a certain context, a problem, and a solution." Christopher Alexander[7]. Thinking about the challenges of concurrent programming from a more conceptional point of view provides many benefits. In contrast to the more conceptional patterns to concurrency, the chapter best practices provide pragmatic tips to overcome the concurrency challenges.

---

[7]https://en.wikipedia.org/wiki/Christopher_Alexander

1.9.1 Synchronization

A necessary prerequisite for a data race is shared mutable state. Synchronization patterns boil down to two concerns: dealing with sharing and dealing with mutation.

1.9.2 Concurrent Architecture

The chapter to concurrent architecture presents five patterns. The two patterns Active Object and the Monitor Object synchronize and schedule member functions invocation. The third pattern Half-Sync/Half-Async has an architectural focus and decouples asynchronous and synchronous service processing in concurrent systems. The Reactor pattern and the Proactor pattern can be regarded as variations of the Half-Sync/Half-Async pattern. Both patterns enable event-driven applications to demultiplex and dispatch service requests. The reactor performs its job synchronously, but the proactor asynchronously.

1.9.3 Best Practices

Concurrent programming is inherently complicated, therefore having best practices in general, but also for multithreading, and the memory model makes a lot of sense.

1.10 Data Structures

A data structure that protects itself so that no data race can appear is called thread-safe. The chapter lock-based data structures presents the challenges to design those lock-based data structures.

1.11 Challenges

Writing concurrent programs is inherently complicated. This is particularly true if you only use C++11 and C++14 features. Therefore, I describe in detail the most challenging issues. I hope that if I dedicate a whole chapter to the challenges of concurrent programming, you become more aware of the pitfalls. I write about challenges such as race conditions, data races, and deadlocks.

1.12 Time Library

The time library is a key component of the concurrent facilities of C++. Often you let a thread sleep for a specific time duration or until a particular point in time. The time library consists of: time points, time durations, and clocks.

1.13 CppMem

CppMem is an interactive tool to get deeper inside into the memory model. It provides two precious services. First, you can verify your lock-free code, and second, you can analyze your lock-free code and get a more robust understanding of your code. I often use CppMem in this book. Because the configuration options and the insights of CppMem are quite challenging, the chapter gives you a basic understanding of CppMem.

1.14 Glossary

The glossary contains non-exhaustive explanations on essential terms.

The Details

2. Memory Model
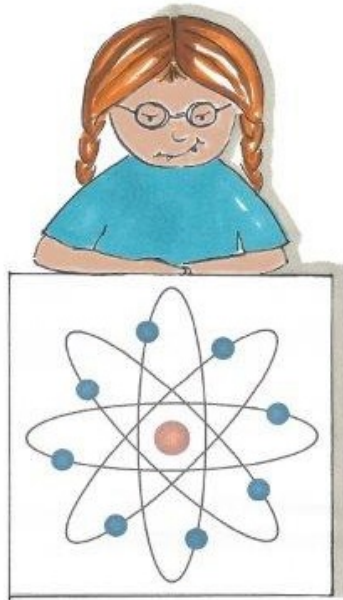
Cippi studies the atomics

The foundation of multithreading is a *well-defined* memory model. From the reader's perspective, it consists of two aspects. On the one hand, there is the enormous complexity of it, which often contradicts our intuition. On the other hand, it helps a lot to get a more in-depth insight into the multithreading challenges.

However, first of all, what is a memory model?
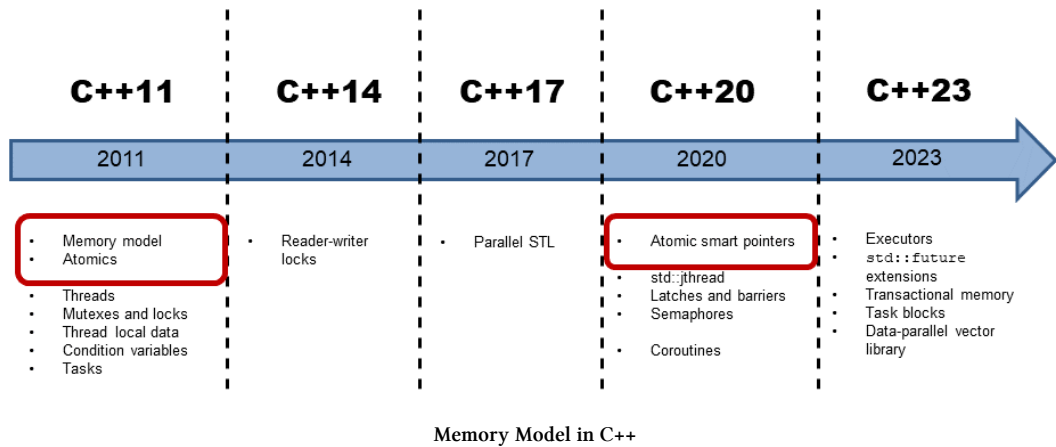
Memory Model in C++

2.1 Basics of the Memory Model

From the concurrency point of view, there are two main aspects of the memory model:

- What is a memory location?
- What happens if two threads access the same memory location?

Let me answer both questions.

2.1.1 What is a memory location?

A memory location is according to cppreference.com[1]

- an object of scalar type (arithmetic type, pointer type, enumeration type, or `std::nullptr_t`),
- or the largest contiguous sequence of bit fields of non-zero length.

Here is an example of a memory location:

---

[1]http://en.cppreference.com/w/cpp/language/memory_model

```cpp
struct S {
    char a;         // memory location #1
    int b : 5;      // memory location #2
    int c : 11,     // memory location #2 (continued)
          : 0,
        d : 8;      // memory location #3
    int e;          // memory location #4
    double f;       // memory location #5
    std::string g;  // several memory locations
};
```

First, the object `obj` consists of seven sub-objects, and the two bit fields `b`, and `c` share the same memory location.

Here are a few important observations:

- Each variable is an object.
- Scalar types occupy one memory location.
- Adjacent bit fields (`b` and `c`) have the same memory location.
- Variables occupy at least one memory location.

Now, to the crucial part of multithreading.

2.1.2 What happens if two threads access the same memory location?

If two threads access the same memory location - adjacent bit fields can share the same memory location - and at least one thread wants to modify it, your program has a data races unless

1. the memory location is modified by an atomic operation.
2. one access happens-before the other.

The second case is quite interesting because synchronization primitives such as mutexes establish happens-before relations. These happens-before relations are based on operations on atomics and apply in general also on operations on non-atomics. The memory-ordering defines the details which cause the happens-before relations and are, therefore, an essential fundamental part of the memory model.

This was my first formal approach to the memory model. Now, I want to give you a mental model for the memory model. The C++ memory model defines a contract.

2.2 The Contract

This contract is between the programmer and the system. The system consists of the compiler that generates machine code, the processor that executes the machine code and includes the different caches that store the program's state. Each of the participants wants to optimize their part. For example, the compiler uses registers or modifies loops; the processor performs out-of-order execution or branch prediction; the caches apply prefetching of instructions or buffering of values. The result is - in the right case - a *well-defined* executable that is fully optimized for the hardware platform. To be precise, there is not only a single contract but a fine-grained set of contracts. Or to say it differently: the weaker the rules are that the programmer has to follow, the more potential there is for the system to generate a highly optimized executable.

There is a rule of thumb. The stronger the contract, the fewer liberties for the system to generate an optimized executable. Sadly, the other way around does not work. When the programmer uses an extremely weak contract or memory model, there are many optimization choices. The consequences are that the program is only manageable by a handful of worldwide recognized experts, which probably neither you nor I belong.

Roughly speaking, there are three contract levels in C++11.

strong

Single threading
• One control flow

Multi-threading
• Tasks
• Threads
• Condition variables

Atomic
• Sequential consistency
• Acquire-release semantic
• Relaxed semantic

weak
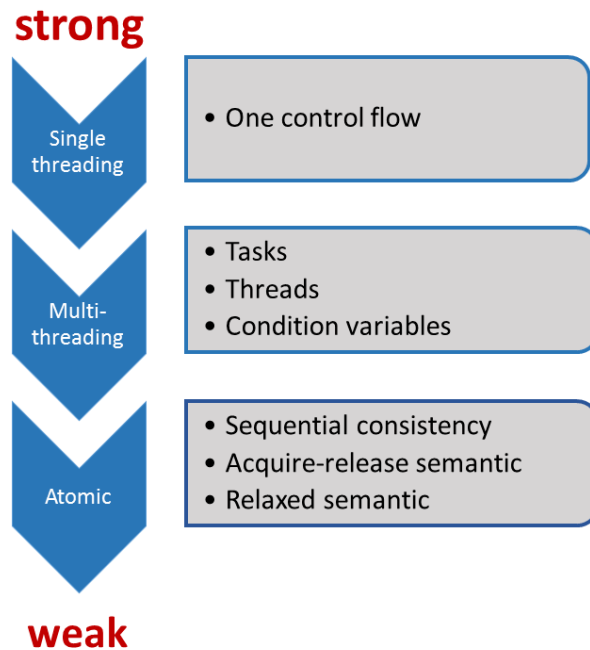
Three levels of the contract

Before C++11, there was only one contract. The C++ language specification did not include mul-

tithreading or atomics. The system only knew about one control flow, and therefore there were only restricted opportunities to optimize the executable. The system's key point was to guarantee for the programmer that the observed behavior of the program corresponded to the sequence of the instructions in the source code. Of course, this means that there was no memory model. Instead, there was the concept of a sequence point. Sequence points are points in the program at which the effects of all instructions preceding it must be observable. The start or the end of the execution of a function are sequence points. When you invoke a function with two arguments, the C++ standard makes no guarantee about which argument is evaluated first, so the behavior is unspecified. The reason is straightforward - the comma operator is not a sequence point, which does not change in C++.

With C++11 everything has changed. C++11 is the first standard aware of multiple threads. The reason for the *well-defined* behavior of threads is the C++ memory model that was heavily inspired by the Java memory model[2]. Still, the C++ memory model goes - as always - a few steps further. The programmer has to obey a few rules in dealing with shared variables to get a *well-defined* program. The program is undefined if there exists at least one data race. As I already mentioned, you have to be aware of data races if your threads share mutable data. Tasks are a lot easier to use than threads or condition variables.

With atomics, we enter the domain of the experts. This becomes more evident the more we weaken the C++ memory model. We often talk about lock-free programming when we use atomics. I spoke in this subsection about the weak and strong rules. Indeed, the sequential consistency is called the strong memory model, and the relaxed semantic is called the weak memory model.

2.2.1 The Foundation

The C++ memory model has to deal with the following points:

- Atomic operations: operations that can perform without interruption.
- Partial ordering of operations: sequences of operations that must not be reordered.
- Visible effects of operations: guarantees when operations on shared variables are visible to other threads.

The foundation of the contract are operations on atomics that have two characteristics: They are by definition atomic or indivisible, and they create synchronization and order constraints on the program execution. These synchronization and order constraints also hold for operations on non-atomics. On the one hand, an operation on an atomic is always atomic, but on the other hand, you can tailor the synchronizations and order constraints to your needs.

2.2.2 The Challenges

The more we weaken the memory model, the more we change our focus towards other things, such as:

---

[2]https://en.wikipedia.org/wiki/Java_memory_model

- The program has more optimization potential.
- The possible number of control flows of the program increases exponentially.
- We are in the domain for the experts.
- The program breaks our intuition.
- We apply micro-optimization.

To deal with multithreading, we should be an expert. In case we want to deal with atomics (sequential consistency), we should open the door to the next level of expertise. What happens when we talk about the acquire-release semantic or relaxed semantic? We advance one step higher to (or deeper into) the next expertise level.

Expert levels
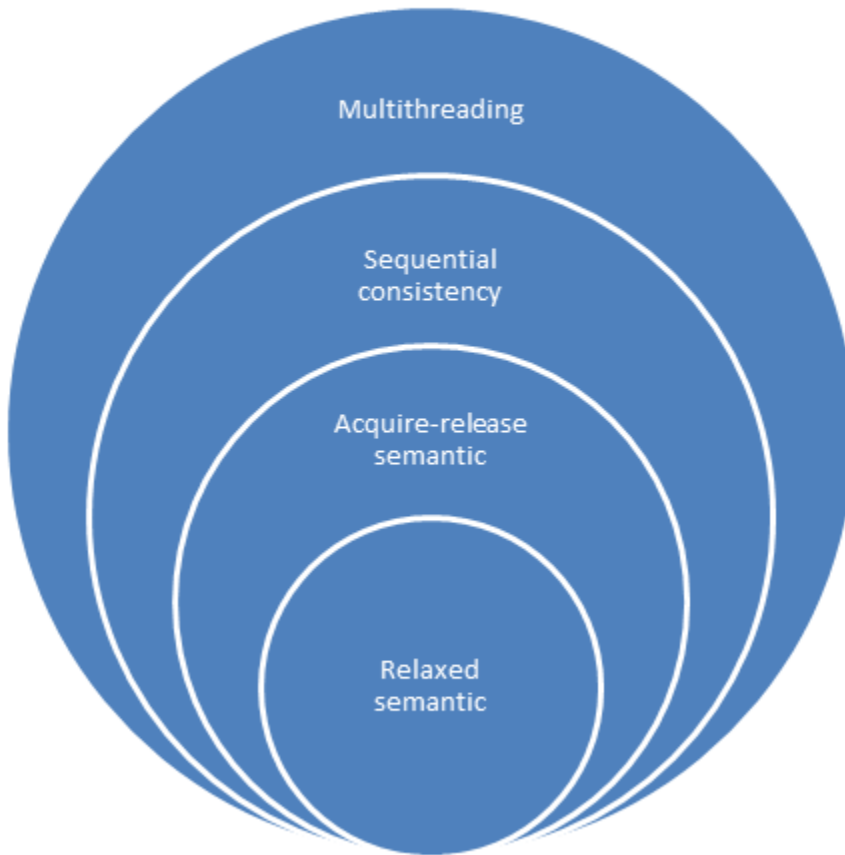
The expert levels

Now, we dive deeper into the C++ memory model and start with lock-free programming. On our journey, I write about atomics and their operations. Once we are done with the basics, the different levels of the memory model follow. The starting point is the straightforward sequential consistency, the acquire-release semantic follows, and the not so intuitive relaxed semantic is the endpoint of our journey.

Let's start with atomics.

2.3 Atomics

Atomics are the base of the C++ memory model. By default, the strong version of the memory model is applied to the atomics; therefore, it makes much sense to understand the features of the strong memory model.

2.3.1 Strong versus Weak Memory Model

As you may already know from the subsection on Contract: The Challenges, with the strong memory model I refer to sequential consistency, and with the weak memory model I refer to relaxed semantic.

2.3.1.1 Strong Memory Model

Java 5.0 got its current memory model in 2004, C++ in 2011. Before that, Java had an erroneous memory model, and C++ had no memory model. Those who think this is the endpoint of a long process are entirely wrong. The foundations of multithreaded programming are 40 to 50 years old. Leslie Lamport[3] defined the concept of sequential consistency in 1979.

Sequential consistency provides two guarantees:

- The instructions of a program are executed in the order written down.
- There is a global order of all operations on all threads.

Before I dive deeper into these two guarantees, I want to explicitly emphasize that these two guarantees only hold for atomics but influence non-atomics.

This graphic shows two threads. Each thread stores its variable x or y respectively, loads the other variable y or x, and stores them in the variable *res1* or *res2*.
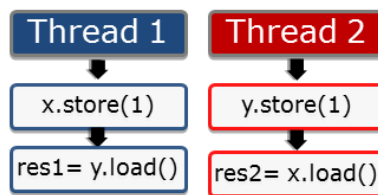
Two atomics

Because the variables are atomic, the operations are executed atomically. By default, sequential consistency applies. The question is: in which order can the statements be executed?

The first guarantee of sequential consistency is that the instructions are executed in the order defined in the source code. This is easy. No store operation can overtake a load operation.

---

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

The second guarantee of the sequential consistency is that all threads' instructions have to follow a global order. In the case listed above, it means that thread 2 sees the operations of thread 1 in the same order in which thread 1 executes them. This is the critical observation. Thread 2 sees all operations of thread 1 in the source code order of thread 1. The same holds from the perspective of thread 1. You can think about characteristic two as a global clock that all threads have to obey. The global clock is the global order. Each time the clock makes a tick, one atomic operation takes place, but you never know which one.

We are not yet done with our riddle. We still need to look at the different interleaving executions of the two threads. So the following six interleavings of the two threads are possible.
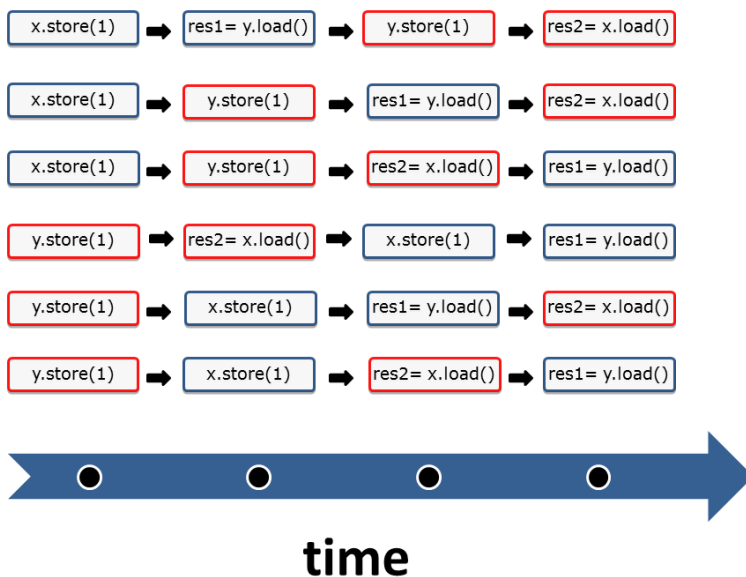
The six interleavings of the two threads

That was easy, right? That was sequential consistency, also known as the strong memory model.

2.3.1.2 Weak Memory Model

Let refer once more to the contract between the programmer and the system.

The programmer uses atomics in this particular example. He obeys his part of the contract. The system guarantees *well-defined* program behavior without data races. In addition to that, the system can execute the four operations in each combination. If the programmer uses the relaxed semantic, the pillars of the contract dramatically change. On the one hand, it is a lot more difficult for the programmer to understand possible interleavings of the two threads. On the other hand, the system has a lot more optimization possibilities.

With the relaxed semantic - also called weak memory model - there are many more combinations of the four operations possible. The *counter-intuitive* behavior is that thread 1 can see the operations of thread 2 in a different order, so there is no view of a global clock. From the perspective of thread 1, it is possible that the operation `res2= x.load()` overtakes `y.store(1)`. It is even possible that thread 1 or thread 2 do not perform their operations in the order defined in the source code. For example, thread 2 can first execute `res2= x.load()` and then `y.store(1)`.

Between the sequential consistency and the relaxed-semantic are a few more models. The most important one is the acquire-release semantic. With the acquire-release semantic, the programmer has to obey weaker rules than with sequential consistency. In contrast, the system has more optimization possibilities. The acquire-release semantic is the key to a deeper understanding of synchronization and partial ordering in multithreading programming. The threads are synchronized at specific synchronization points in the code. Without these synchronization points, there is no *well-defined* behavior of threads, tasks, or condition variables possible.

In the last section, I introduced sequential consistency as the default behavior of atomic operations. What does that mean? You can specify the memory order for each atomic operation. If no memory order is specified, sequential consistency is applied, meaning that the flag `std::memory_order_seq_cst` is implicitly applied to each operation on an atomic.

This piece of code

```
x.store(1);
res = x.load();
```

is equivalent to the following piece of code:

```
x.store(1, std::memory_order_seq_cst);
res = x.load(std::memory_order_seq_cst);
```

For simplicity, I use the first form in this book. Now it's time to take a deeper look into the atomics of the C++ memory model. We start with the elementary `std::atomic_flag`.

2.3.2 The Atomic Flag

`std::atomic_flag` is an atomic boolean. It has a clear and a set state. For simplicity reasons, I call the clear state `false` and the set state `true`. Its `clear` member functions enables you to set its value to `false`. With the `test_and_set` member functions, you can set the value back to `true` and return the previous value. There is no member functions to ask for the current value. This will change with C++20. With C++20, a `std::atomic_flag` has a `test` member functions and can be used for thread synchronization via the member functions `notify_one`, `notify_all`, and `wait`.

All operations of `std::atomic_flag atomicFlag`

Member functions	Description
`atomicFlag.clear()`	Clears the atomic flag.
`atomicFlag.test_and_set()`	Sets the atomic flag and returns the old value.
`atomicFlag.test()` (C++20)	Returns the value of the flag.
`atomicFlag.notify_one()` (C++20)	Notifies one thread waiting on the atomic flag.
`atomicFlag.notify_all` (C++20)	Notifies all threads waiting on the atomic flag.
`atomicFlag.wait(bo)` (C++20)	Blocks the thread until notified and the atomic value changes.

The call `atomicFlag.test()` returns the `atomicFlag` value without changing it. Further on, you can use `std::atomic_flag` for thread-synchronization: `atomicFlag.wait()`, `atomicFlag.notify_one()`, and `atomicFlag.notify_all()`. The member functions `notify_one` or `notify_all` notify one or all of the waiting atomic flags. `atomicFlag.wait(boo)` needs a boolean `boo`. The call `atomicFlag.wait(boo)` blocks until the next notification or spurious wakeup. It checks then if the value of `atomicFlag` is equal to `boo` and unblocks if not. The value `boo` serves as a predicate to protect against spurious wakeups.

Additionally to C++11, default-construction in C++20 set a `std::atomic_flag` in its `false` state.

> ⚠️ **Initialization of a `std::atomic_flag` in C++11**
>
> The `std::atomic_flag  flag` has to be initialized in C+++11 with the statement `std::atomic_flag  flag = ATOMIC_FLAG_INIT`. Other initialization contexts such as `std::atomic_flag flag(ATOMIC_FLAG_INIT)` are unspecified.

`std::atomic_flag` has two outstanding properties.

`std::atomic_flag` is

- the only lock-free atomic. A non-blocking algorithm is lock-free if there is guaranteed system-wide progress.
- the building block for higher-level thread abstractions.

The only lock-free atomic? The remaining more powerful atomics can provide their functionality by using a mutex internally according to the C++ standard. These remaining atomics have a member functions called `is_lock_free` to check if the atomic uses a mutex internally. On the popular microprocessor architectures, I always get the answer `true`. You should be aware of this and check it on your target system if you want to program lock-free.

> 🔑 `adress_free` Atomic operations that are lock-free should also be address-free. Address-free means that atomic operations from different processes on the same memory location are atomic.

std::is_always_lock_free

You can check for each instance of an atomic type or atomic_ref type (C++20) obj if its lock-free: obj.is_lock_free(). This check is performed at runtime. Thanks to the constexpr function atomic<type>::is_always_lock_free, you can check for each atomic type if it's lock-free on all supported hardware that the executable might run on. This check returns only true if it is true for all supported hardware. The check is performed at compile-time and is available since C++17.

The following expression should never fail:

```
if (std::atomic<T>::is_always_lock_free) assert(std::atomic<T>().is_lock_free());
```

The interface of std::atomic_flag is powerful enough to build a spinlock. With a spinlock, you can protect a critical section as you would with a mutex.

2.3.2.1 Spinlock

A spinlock is an elementary lock such as a mutex. In contrast to a mutex, it waits not until it gets its lock. It eagerly asks for the lock to get access to the critical section. The spinlock saves the expensive context switch in the wait state from the user space to the kernel space, but it utilizes the CPU and wastes CPU cycles. If threads are typically blocked for a short time, spinlocks are quite efficient. Often a lock uses a combination of a spinlock and a mutex. The lock first uses the spinlock for a limited time. If this does not succeed the thread is then put in the wait state.

Spinlocks should not be used on a single processor system. In the best case, a spinlock wastes resources and slows down the owner of the lock. In the worst case, you get a deadlock.

The example shows the implementation of a spinlock with the help of std::atomic_flag

A spinlock with std::atomic_flag

```
1   // spinLock.cpp
2
3   #include <atomic>
4   #include <thread>
5
6   class Spinlock{
7     std::atomic_flag flag = ATOMIC_FLAG_INIT;
8   public:
9
10    void lock(){
11      while( flag.test_and_set() );
12    }
```

```
13
14    void unlock(){
15       flag.clear();
16    }
17
18  };
19
20  Spinlock spin;
21
22  void workOnResource(){
23    spin.lock();
24    // shared resource
25    spin.unlock();
26  }
27
28
29  int main(){
30
31    std::thread t(workOnResource);
32    std::thread t2(workOnResource);
33
34    t.join();
35    t2.join();
36
37  }
```

Both threads t and t2 (lines 31 and 32) are competing for the critical section. For simplicity, the critical section in line 24 consists only of a comment. How does it work? The class Spinlock has - similar to a mutex - the member function lock and unlock. In addition to this, the std::atomic_flag is initialized with class member initialization to false (line 7).

If thread t is going to execute the function workOnResource, the following scenarios can happen.

1. Thread t gets the lock because the lock invocation was successful. The lock invocation is successful if the flag's initial value in line 11 is false. In this case, thread t sets it in an atomic operation to true. The value true is the value of the while loop that returns to thread t2 if it tries to get the lock. So thread t2 is caught in the rat race. Thread t2 cannot set the flag's value to false, so that t2 must wait until thread t1 executes the unlock member functions and sets the flag to false (lines 14 - 16).

2. Thread t doesn't get the lock. So we are in scenario 1 with swapped roles.

I want you to focus your attention on the member functions test_and_set of std::atomic_flag. The member function test_and_set consists of two operations: reading and writing. Both operations must be performed in one atomic operation. If not, we would have a read and a write on the shared resource (line 24). That is by definition a data race, and the program has undefined behavior.

It's exciting to compare the active waiting of a spinlock with the passive waiting of a mutex.

2.3.2.1.1 Spinlock versus Mutex

What happens to the CPU load if the function workOnResource locks the spinlock for 2 seconds (lines 23 - 25)?

Waiting with a spinlock

```cpp
// spinLockSleep.cpp

#include <atomic>
#include <thread>

class Spinlock{
  std::atomic_flag flag = ATOMIC_FLAG_INIT;
public:

  void lock(){
    while( flag.test_and_set() );
  }

  void unlock(){
    flag.clear();
  }

};

Spinlock spin;

void workOnResource(){
  spin.lock();
  std::this_thread::sleep_for(std::chrono::milliseconds(2000));
  spin.unlock();
}


int main(){

  std::thread t(workOnResource);
  std::thread t2(workOnResource);

  t.join();
  t2.join();

}
```

If the theory is correct, one of the four cores of my PC is fully utilized. This is precisely what happens.

Take a look at the screenshot.
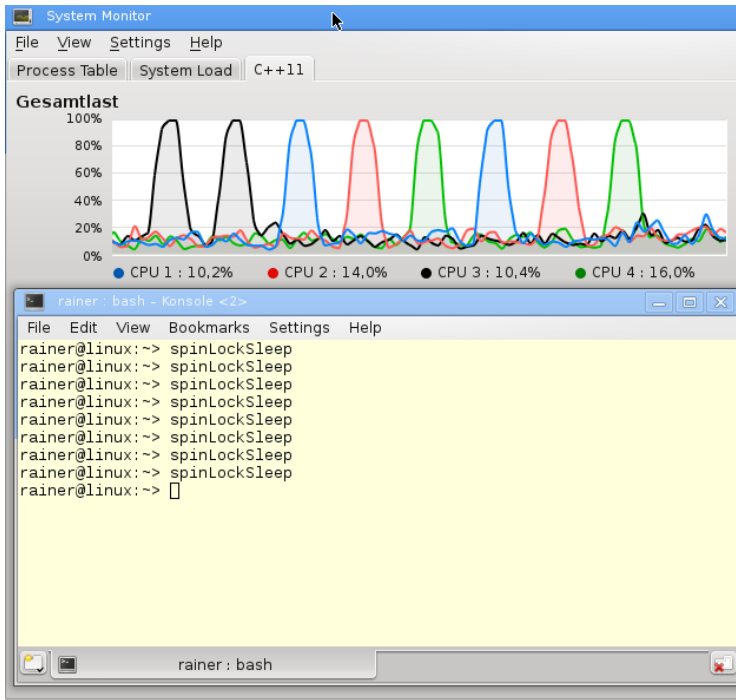
A spinlock that sleeps for two seconds

The screenshot shows nicely that the load of one core reaches 100% on my PC. Each time a different core performs busy waiting.

Now I use a mutex instead of a spinlock. Let's see what happens.

Waiting with a mutex

```cpp
// mutex.cpp

#include <mutex>
#include <thread>

std::mutex mut;

void workOnResource(){
  mut.lock();
  std::this_thread::sleep_for(std::chrono::milliseconds(5000));
  mut.unlock();
}

```

```
14    int main(){
15
16      std::thread t(workOnResource);
17      std::thread t2(workOnResource);
18
19      t.join();
20      t2.join();
21
22    }
```

Although I executed the program several times, I did not observe a significant load on any of the cores.
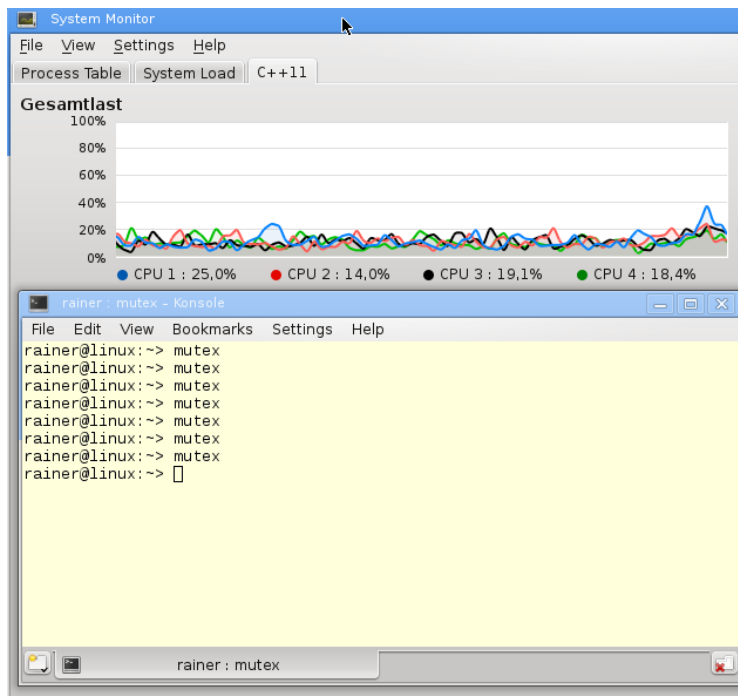
A mutex that sleeps for two seconds

Thanks to std::atomic_flag, thread synchronization is straightforward and fast.

2.3.2.2 Thread Synchronization

Thread synchronization with a `std::atomic_flag`

```cpp
// threadSynchronizationAtomicFlag.cpp

#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> myVec{};

std::atomic_flag atomicFlag{};

void prepareWork() {

    myVec.insert(myVec.end(), {0, 1, 0, 3});
    std::cout << "Sender: Data prepared."  << '\n';
    atomicFlag.test_and_set();
    atomicFlag.notify_one();

}

void completeWork() {

    std::cout << "Waiter: Waiting for data." << '\n';
    atomicFlag.wait(false);
    myVec[2] = 2;
    std::cout << "Waiter: Complete the work." << '\n';
    for (auto i: myVec) std::cout << i << " ";
    std::cout << '\n';

}

int main() {

    std::cout << '\n';

    std::thread t1(prepareWork);
    std::thread t2(completeWork);

    t1.join();
    t2.join();

    std::cout << '\n';

}
```

The thread preparing the work (line 16) sets the `atomicFlag` to `true` and sends the notification. The thread completing the work waits for the notification. It is only unblocked if `atomicFlag` is equal to `true`.

Here are a few runs of the program with the Microsoft Compiler.

Thread synchronization with `std::atomic_flag`

Even when the sender sends its notification before the waiter is in the wait state, the notification is not lost. `std::atomic_flag` cannot be victims of lost wakeups.

Let's go one step further from the basic building block `std::atomic_flag` to the more advanced atomics: the class template `std::atomic`.

2.3.3 `std::atomic`

They are various variations of the class template `std::atomic` available. `std::atomic<bool>` and `std::atomic<user-defined type>` use the primary template. Partial specialisations are available

for pointers `std::atomic<T*>`, and with C++20 for smart pointers `std::atomic<smart  T*>`, full specialisations for integral types `std::atomic<integral  type>`, and with C++20 for floating-point types `std::atomic<floating-point>`.

The atomic booleans, the atomic user-defined types, and the atomic smart pointer support the same interface. I call it the fundamental atomic interface. The atomic pointer extends the fundamental atomic interface. The same applies to the atomic arithmetic types: the atomic floating-point types extend the interface of the atomic pointers, and the atomic integral types extend the interface of the atomic floating-point types.

The downside of the various variations of `std::atomic` is that you do not have the guarantee that they are lock-free. In the next subsection, I present the various atomic types based on their interface. Roughly speaking, there are four subsections. Let me start with atomic booleans, atomic user-defined types, and atomic smart pointers (C++20).

2.3.3.1 Fundamental Atomic Interface

The threee partial specialization `std::atomic<bool>`, `std::atomic<user-defined type>`, and `std::atomic<smart T*>` support the fundamental atomic interface.

Member functions	Description
`is_lock_free`	Checks if the atomic object is lock-free.
`atomic_ref<T>::is_always_lock_free`	Checks at compile time if the atomic type is always lock-free.
`load`	Atomically returns the value of the atomic.
`operator T`	Atomically returns the value of the atomic. Equivalent to `atom.load()`.
`store`	Atomically replaces the value of the atomic with the non-atomic.
`exchange`	Atomically replaces the value with the new value. Returns the old value.
`compare_exchange_strong` `compare_exchange_weak`	Atomically compares and eventually exchanges the value. Details are here.
`notify_one` (C++20) `notify_all` (C++20)	Notifies one atomic wait operation. Notifies all atomic wait operations.
`wait` (C++20)	Blocks until it is notified. Compares itself with the `old` value to protect against spurious wakeups and lost wakeups. If the `old` value compares to unequal, returns.

Let me present more details about `std::atomic<bool>`.

2.3.3.1.1 `std::atomic<bool>`

`std::atomic<bool>` has a lot more to offer than `std::atomic_flag`. It can explicitly be set to `true` or `false`.

> **⚠ `atomic` is not `volatile`**
>
> What does the keyword `volatile` in C# and Java have in common with the keyword `volatile` in C++? Nothing! That is the difference between `volatile` and `std::atomic`.
>
> - **volatile**: is for special objects on which optimized read or write operations are not allowed.
> - **std::atomic**: defines atomic variables, which are meant for a thread-safe reading and writing.
>
> The confusion starts here. The keyword `volatile` in Java and C# has the meaning of `std::atomic` in C++. Alternatively, `volatile` has no multithreading semantics in C++.
>
> `volatile` is typically used in embedded programming to denote objects which can change independently of the regular program flow. One example is an object which represents an external device (memory-mapped I/O). Because these objects can change independently of the regular program flow and their value is directly written into main memory, no optimized storage in caches occurs.

`std::atomic<bool>` is already sufficient to synchronise two threads and I can, therefore, implement a kind of a condition variable with a `std::atomic<bool>`.

2.3.3.1.2 Simulating a Condition Variable

Let's first use a condition variable to synchronize two threads.

Usage of a condition variable

```cpp
// conditionVariable.cpp

#include <condition_variable>
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> mySharedWork;
std::mutex mutex_;
std::condition_variable condVar;

bool dataReady{false};

void waitingForWork(){
```

```
15      std::cout << "Waiting " << '\n';
16      std::unique_lock<std::mutex> lck(mutex_);
17      condVar.wait(lck, []{ return dataReady; });
18      mySharedWork[1] = 2;
19      std::cout << "Work done " << '\n';
20  }
21
22  void setDataReady(){
23      mySharedWork = {1, 0, 3};
24      {
25          std::lock_guard<std::mutex> lck(mutex_);
26          dataReady = true;
27      }
28      std::cout << "Data prepared" << '\n';
29      condVar.notify_one();
30  }
31
32  int main(){
33
34    std::cout << '\n';
35
36    std::thread t1(waitingForWork);
37    std::thread t2(setDataReady);
38
39    t1.join();
40    t2.join();
41
42     for (auto v: mySharedWork){
43         std::cout << v << " ";
44    }
45
46
47    std::cout << "\n\n";
48
49  }
```

Let me say a few words about the program. For a in-depth discussion of condition variables, read the chapter condition variables in this book.

Thread t1 waits in line 17 for the notification of thread t2. Both threads use the same condition variable condVar and synchronize on the same mutex mutex_. How does the workflow run?

- Thread t2

    – prepares the work package mySharedWork = {1, 0, 3}

  - set the non-atomic boolean `dataReady` to `true`
  - send its notification `condVar.notify_one`

- Thread `t1`

  - waits for the notification `condVar.wait(lck, []{ return dataReady; })` while holding the lock `lck`
  - continues its work `mySharedWork[1] = 2` after getting the notification

The boolean `dataReady` that thread `t2` sets to `true` and thread `t1` checks in the lambda-function `[]{ return dataReady; }` stands for a kind of memory for the stateless condition variable. Condition variables may be a victim to two phenomena:

1. spurious wakeup: the receiver of the message wakes up, although no notification happened,

2. lost wakeup: the sender sends its notification before the receiver is in the wait state.

And now the pendant with `std::atomic<bool>`.

Implementation of a condition variable with `std::atomic<bool>`

```cpp
// atomicCondition.cpp

#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> mySharedWork;
std::atomic<bool> dataReady(false);

void waitingForWork(){
    std::cout << "Waiting " << '\n';
    while (!dataReady.load()){
        std::this_thread::sleep_for(std::chrono::milliseconds(5));
    }
    mySharedWork[1] = 2;
    std::cout << "Work done " << '\n';
}

void setDataReady(){
    mySharedWork = {1, 0, 3};
    dataReady = true;
    std::cout << "Data prepared" << '\n';
}

int main(){
```

```
28
29     std::cout << '\n';
30
31     std::thread t1(waitingForWork);
32     std::thread t2(setDataReady);
33
34     t1.join();
35     t2.join();
36
37     for (auto v: mySharedWork){
38         std::cout << v << " ";
39     }
40
41
42     std::cout << "\n\n";
43
44 }
```

What guarantees that line 17 is executed after the line 14? Or more generally that the thread t1 executes mySharedWork[1] = 2 (line 17) after thread t2 had executed mySharedWork = {1, 0, 3} (line 22). Now it gets more formal.

- Line 22 *happens-before* line 23
- Line 14 *happens-before* line 17
- Line 23 *synchronizes-with* line 14
- Because *synchronizes-with* establishes a *happens-before* relation and *happens-before* is transitive, it follows: mySharedWork = {1, 0, 3} *happens-before* mySharedWork[1] = 2
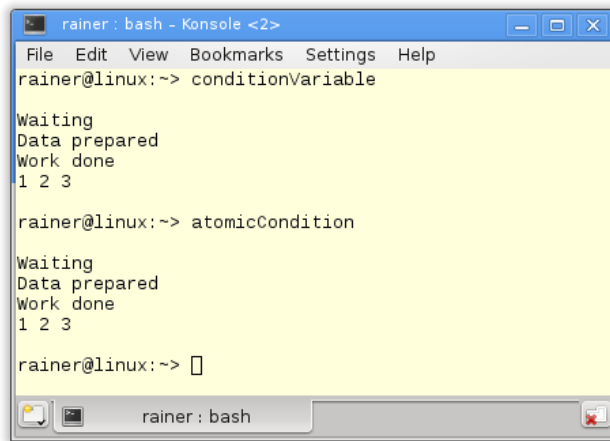
That was easy, wasn't it? For simplicity, I ignored that *synchronizes-with* establishes an *inter-thread happens before* and *inter-thread happens before* establishes a *happens-before* relation. In case you are curious, here are the details: memory_order[4].

I want to mention the critical point explicitly: access to the shared variable mySharedWork is synchronized using the condition variable condVar or the atomic dataReady. This holds, although mySharedWork itself is not protected by a lock or is atomic.

Both programs produce the same result for mySharedWork.

---

[4]http://en.cppreference.com/w/cpp/atomic/memory_order

Synchronizations of two threads with an atomic and a condition variable

🔑 Push versus Pull Principle

I cheated a little. There is one key difference between synchronizing threads with a condition variable and `std::atomic<bool>`. The condition variable notifies the waiting thread (`condVar.notify()`) that it should proceed with its work. The waiting thread with `std::atomic<bool>` checks if the sender is done with its work (`dataRead = true`).

The condition variable notifies the waiting thread (push principle) while the atomic boolean repeatedly asks for the value (pull principle).

`std::atomic<bool>` and the other full or partial specializations of `std::atomic` support the bread and butter of all atomic operations: `compare_exchange_strong` and `compare_exchange_weak`.

2.3.3.1.3 `compare_exchange_strong` and `compare_exchange_weak`

`compare_exchange_strong` has the syntax: `bool compare_exchange_strong(T& expected, T& desired)`. Because this operation compares and exchanges its values in one atomic operation, it is often called compare and swap (CAS). This kind of operation is available in many programming languages and is the foundation of non-blocking algorithms. Of course, the behavior may vary a little. `atomicValue.compare_exchange_strong(expected, desired)` has the following behavior.

- If the atomic comparison of `atomicValue` with `expected` returns `true`, `atomicValue` is set in the same atomic operation to `desired`.
- If the comparison returns `false`, `expected` is set to `atomicValue`.

The reason the operation `compare_exchange_strong` is called strong is apparent. There is also a member functions `compare_exchange_weak`. The weak version can fail spuriously. That means, although `*atomicValue == expected` holds, `atomicValue` was not set to `desired` and the function call returns

false, so you have to check the condition in a loop: `while (!atomicValue.compare_exchange_-weak(expected, desired))`. The weak form exists because some processors don't support an atomic compare-exchange instruction. When called in a loop, you should prefer the weak form. On some platforms, the weak form can be faster.

CAS operations are open for the so-called ABA problem. This means you read a value twice, and each time it returns the same value A; therefore, you conclude that nothing changed in between. However, you overlooked that the value may have changed to B in between readings.

The weak forms of the functions are allowed to fail spuriously, that is, act as if `*this != expected` even if they are equal. When a compare-and-exchange is in a loop, the weak version may have better performance on some platforms.

`std::atomic<bool>` makes it trivial in C++20 to synchronize two threads.

2.3.3.1.4 Thread Synchronization (C++20)

It is trivial to refactor the thread synchronization program using `std::atomic_flag` to `std::atomic<bool>`.

Thread synchronization with `std::atomic<bool>`

```cpp
1   // threadSynchronizationAtomicBool.cpp
2
3   #include <atomic>
4   #include <iostream>
5   #include <thread>
6   #include <vector>
7
8   std::vector<int> myVec{};
9
10  std::atomic<bool> atomicBool{false};
11
12  void prepareWork() {
13
14      myVec.insert(myVec.end(), {0, 1, 0, 3});
15      std::cout << "Sender: Data prepared."  << '\n';
16      atomicBool.store(true);
17      atomicBool.notify_one();
18
19  }
20
21  void completeWork() {
22
23      std::cout << "Waiter: Waiting for data." << '\n';
24      atomicBool.wait(false);
25      myVec[2] = 2;
26      std::cout << "Waiter: Complete the work." << '\n';
```

```
27        for (auto i: myVec) std::cout << i << " ";
28        std::cout << '\n';
29
30    }
31
32    int main() {
33
34        std::cout << '\n';
35
36        std::thread t1(prepareWork);
37        std::thread t2(completeWork);
38
39        t1.join();
40        t2.join();
41
42        std::cout << '\n';
43
44    }
```

The call `atomicBool.wait(false)` blocks if `atomicBool == false` holds. Consequently, the call `atomicBool.store(true)` (line 16) send `atomicBool` to `true` and send afterwards its notification.

According to `std::atomic_flag`, here are four runs with the Microsoft Compiler.

Thread synchronization with `std::atomic<bool>`

In addition to booleans, there is atomics for pointers, integrals, and user-defined types. The rules for user-defined types are unique.

All variations of `std::atomic` support the CAS operations.

2.3.3.1.5 User Defined Atomics `std::atomic<user-defined type>`

Thanks to the class template `std::atomic`, you can define your user-defined atomic type.

There are many strong restrictions on a user-defined type if you use it for an atomic type `std::atomic<user-defined type>`. The atomic type `std::atomic<user-defined type>` supports the same interface as `std::atomic<bool>`.

Here are the restrictions for a user-defined type to become an atomic type:

- The copy assignment operator for a user-defined type, for all its base classes and all non-static members, must be trivial. This means that you must not define the copy assignment operator, but you can request it from the compiler using `default`[5].
- a user-defined type must not have virtual member functions or virtual base classes

---

[5]http://en.cppreference.com/w/cpp/keyword/default

- a user-defined type must be bitwise comparable so that the C functions memcpy[6] or memcmp[7] can be applied

Most popular platforms can use atomic operations for std::atomic<user-defined type> if the size of the user-defined type is not bigger as the size of an int.

Check the type properties at compile time

The type properties on a user-defined type can be checked at compile time, by using the following functions: std::is_trivially_copy_constructible, std::is_polymorphic and std::is_trivial. All these functions are part of the very powerful type-traits library[8].

2.3.3.1.6 Atomic Smart Pointers std::atomic<smart T*> **(C++20)**

A std::shared_ptr consists of a control block and its resource. The control block is thread-safe, but access to the resource is not. This means modifying the reference counter is an atomic operation and you have the guarantee that the resource is deleted exactly once. These are the guarantees std::shared_ptr gives you. .

The Importance of being Thread-Safe

I want to take a short detour to emphasize how important it is that the std::shared_ptr has well-defined multithreading semantics. At first glance, use of a std::shared_ptr does not appear to be a sensible choice for multithreaded code. It is by definition shared and mutable and is the ideal candidate for non-synchronized read and write operations and hence for undefined behavior. On the other hand, there is the guideline in modern C++: **Don't use raw pointers**. This means, consequently, that you should use smart pointers in multithreaded programs.

The proposal N4162[9] for atomic smart pointers directly addresses the deficiencies of the current implementation. The deficiencies boil down to these three points: consistency, correctness, and performance.

- **Consistency**: the atomic operations for std::shared_ptr are the only atomic operations for a non-atomic data type.
- **Correctness**: the use of the global atomic operations is quite error-prone because the correct usage is based on discipline. It is easy to forget to use an atomic operation - such as using ptr = localPtr instead of std::atomic_store(&ptr, localPtr). The result is undefined behavior because of a data race. If we used an atomic smart pointer instead, the type system would not allow it.

---

[6]http://en.cppreference.com/w/cpp/string/byte/memcpy
[7]http://en.cppreference.com/w/cpp/string/byte/memcmp
[8]http://en.cppreference.com/w/cpp/header/type_traits
[9]http://wg21.link/n4162

- **Performance**: the atomic smart pointers have a big advantage compared to the free `atomic_*` functions. The atomic versions are designed for the special use case and can internally have a `std::atomic_flag` as a kind of cheap spinlock[10]. Designing the non-atomic versions of the pointer functions to be thread-safe would be overkill where they are used in a single-threaded scenario. They would have a performance penalty.

The correctness argument is probably the most important one. Why? The answer lies in the proposal. The proposal presents a thread-safe singly-linked list that supports insertion, deletion, and searching of elements. This singly-linked list is implemented in a lock-free way.

2.3.3.1.7 A thread-safe singly linked list

```cpp
template<typename T> class concurrent_stack {
    struct Node { T t; shared_ptr<Node> next; };
    atomic_shared_ptr<Node> head;
            // in C++11: remove "atomic_" and remember to use the special
            // functions every time you touch the variable
    concurrent_stack( concurrent_stack &) =delete;
    void operator=(concurrent_stack&) =delete;

public:
    concurrent_stack() =default;
    ~concurrent_stack() =default;
    class reference {
        shared_ptr<Node> p;
    public:
        reference(shared_ptr<Node> p_) : p{p_} { }
        T& operator* () { return p->t; }
        T* operator->() { return &p->t; }
    };

    auto find( T t ) const {
        auto p = head.load();  // in C++11: atomic_load(&head)
        while( p && p->t != t )
            p = p->next;
        return reference(move(p));
    }
    auto front() const {
        return reference(head); // in C++11: atomic_load(&head)
    }
    void push_front( T t ) {
      auto p = make_shared<Node>();
      p->t = t;
      p->next = head;         // in C++11: atomic_load(&head)
      while( !head.compare_exchange_weak(p->next, p) ){ }
      // in C++11: atomic_compare_exchange_weak(&head, &p->next, p);
    }
    void pop_front() {
        auto p = head.load();
        while( p && !head.compare_exchange_weak(p, p->next) ){ }
        // in C++11: atomic_compare_exchange_weak(&head, &p, p->next);
    }
};
```

A thread-safe singly linked list

---

[10]https://en.wikipedia.org/wiki/Spinlock

All changes that are required to compile the program with a C++11 compiler are marked in red. The implementation with atomic smart pointers is a lot easier and hence less error-prone. C++20's type system does not permit using a non-atomic operation on an atomic smart pointer.

The proposal N4162[11] proposed the new types `std::atomic_shared_ptr` and `std::atomic_weak_ptr` as atomic smart pointers. By merging them in the mainline ISO C++ standard, they became partial template specialization of `std::atomic`, namely `std::atomic<std::shared_ptr<T>>`, and `std::atomic<std::weak_ptr<T>>`.

Consequently, the atomic operations for `std::shared_ptr` are deprecated with C++20.

The following program shows five thread modifying a `std::atomic<std::shared_ptr<std::string>>` withoud synchronization.

```cpp
// atomicSharedPtr.cpp

#include <iostream>
#include <memory>
#include <atomic>
#include <string>
#include <thread>

int main() {

    std::cout << '\n';

    std::atomic<std::shared_ptr<std::string>> sharString(
        std::make_shared<std::string>("Zero"));

    std::thread t1([&sharString]{
        sharString.store(std::make_shared<std::string>(*sharString.load() + "One"));
    });
    std::thread t2([&sharString]{
        sharString.store(std::make_shared<std::string>(*sharString.load() + "Two"));
    });
    std::thread t3([&sharString]{
        sharString.store(std::make_shared<std::string>(*sharString.load() +"Three"));
    });
    std::thread t4([&sharString]{
        sharString.store(std::make_shared<std::string>(*sharString.load() +"Four"));
    });
    std::thread t5([&sharString]{
        sharString.store(std::make_shared<std::string>(*sharString.load() +"Five"));
    });
```

[11]http://wg21.link/n4162

```
31
32      t1.join();
33      t2.join();
34      t3.join();
35      t4.join();
36      t5.join();
37
38      std::cout << *sharString.load() << '\n';
39
40  }
```

The atomic `std::shared_ptr` `shaString` (line 13) is initialized with the string "Zero". Each of the five threads `t1` to `t5` (lines 16 - 28) adds a string to `sharString` that is displayed in line 38. Using a `std::shared_ptr` instead of `std::atomic<std::shared_ptr>` would be a data race.

Executing the program shows the interleaving of the threads.

Thread-safe modifying of a `std::string`

2.3.3.2 `std::atomic<floating-point type>` **(C++20)**

Additionally to the fundamental atomic interface, `std::atomic<floating-point type>` supports addition and substraction.

Additional operations to the fundamental atomic interface

Member functions	Description
`fetch_add`, `+=`	Atomically adds (subtracts) the value.
`fetch_sub`, `-=`	Returns the old value.

Full specializations for the types `float`, `double`, and `long double` are available.

2.3.3.3 `std::atomic<T*>`

`std::atomic<T*>` is a partial specialization of the class template `std::atomic`. It behaves like a plain pointer `T*`. Additionally to `std::atomic<floating-point type>`, `std::atomic<T*>` supports pre- and post-increment or pre- and post-decrement operations.

<div align="center">

Additional operations to the `std::atomic<floating-point type`

</div>

Member functions	Description
`++, --`	Increments or decrements (pre- and post-increment) the atomic.

Have a look at the short example.

```
int intArray[5];
std::atomic<int*> p(intArray);
p++;
assert(p.load() == &intArray[1]);
p+=1;
assert(p.load() == &intArray[2]);
--p;
assert(p.load() == &intArray[1]);
```

In C++11, there are atomic types for the integral data types.

2.3.3.4 `std::atomic<integral type>`

For each integral type there is a full specialization `std::atomic<integral type>` of `std::atomic`. `std::atomic<integral type>` supports all operations that `std::atomic<T*>` or `std::atomic<floating-point type>` supports. Additionally, `std::atomic<integral type` supports the bitwise logical operators AND, OR, and XOR.

<div align="center">

All operations on `atomic`

</div>

Member functions	Description
`fetch_or, |=`	Atomically performs bitwise (AND, OR, and XOR) operation with the value.
`fetch_and, &=`	Returns the old value.
`fetch_xor, ^=`	

There is a small difference between the composite bitwise-assignment operation and the fetch version. The composite bitwise-assignment operatorion returns the new value; the fetch variation returns the old value.

A more in-depth look provides more insight: there is no atomic multiplication, atomic division, nor an atomic shift operation available. This is not a significant limitation because these operations are seldom needed and can easily be implemented. Here is an example of an atomic `fetch_mult` function.

An atomic multiplication with `compare_exchange_strong`
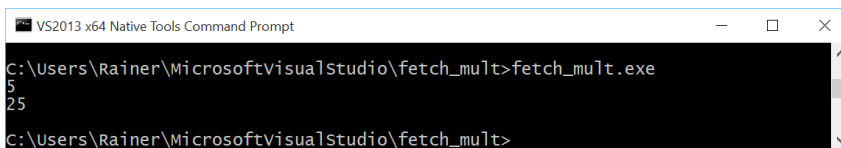
```cpp
// fetch_mult.cpp

#include <atomic>
#include <iostream>

template <typename T>
T fetch_mult(std::atomic<T>& shared, T mult){
  T oldValue = shared.load();
  while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
  return oldValue;
}

int main(){
  std::atomic<int> myInt{5};
  std::cout << myInt << '\n';
  fetch_mult(myInt,5);
  std::cout << myInt << '\n';
}
```

One point worth mentioning is that the multiplication in line 9 only happens if the relation `oldValue == shared` holds. I put the multiplication in a while loop to be sure that the multiplication always takes place because there are two instructions for the reading of `oldValue` in line 8 and its usage in line 9. Here is the result of the atomic multiplication.

An atomic multiplication

The `fetch_mult` algorithm is lock_free

The algorithm `fetch_mult` (line 6) multiplies `std::atomic shared` by `mult`. The key observation is that there is a small time-window between the reading of the old value `T oldValue = shared Load` (line 8) and the comparison with the new value in line 9. Therefore another thread can always step in and change `oldValue`. If you think about a bad interleaving of threads, you see no per-thread progress guarantee.

The consequence is that the algorithm is lock-free, but not wait-free.

Which specializations for integral types exist? Here are the details:

- character types: `char`, `char8_t` (C++20), `char16_t`, `char32_t`, and `wchar_t`

- standard signed integer types: `signed char`, `short`, `int`, `long`, and `long long`

- standard unsigned integer types: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`

- additional integer types, defined in the header `<cstdint>`[12]:

  - `int8_t`, `int16_t`, `int32_t`, and `int64_t` (signed integer with exactly 8, 16, 32, and 64 bits)
  - `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t` (unsigned integer with exactly 8, 16, 32, and 64 bits)
  - `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, and `int_fast64_t` (fastest signed integer with at least 8, 16, 32, and 64 bits)
  - `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, and `uint_fast64_t` (fastest unsigned integer with at least 8, 16, 32, and 64 bits)
  - `int_least8_t`, `int_least16_t`, `int_least32_t`, and `int_least64_t` (smallest signed integer with at least 8, 16, 32, and 64 bits)
  - `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, and `uint_least64_t` (smallest unsigned integer with at least 8, 16, 32, and 64 bits)
  - `intmax_t`, and `uintmax_t` (maximum signed and unsigned integer)
  - `intptr_t`, and `uintptr_t` (signed and unsigned integer for holding a pointer)

2.3.3.5 Type Aliases

For all `std::atomic<bool>` and all `std::atomic<integral type>` the C++ standard provide type aliases if the integral type is available.

Type aliases for `std::atomic<bool>` and `std::atomic<integral type>`

Type alias	Definition
`std::atomic_bool`	`std::atomic<bool>`
`std::atomic_char`	`std::atomic<char>`
`std::atomic_schar`	`std::atomic<signed char>`
`std::atomic_uchar`	`std::atomic<unsigned char>`
`std::atomic_short`	`std::atomic<short>`
`std::atomic_ushort`	`std::atomic<unsigned short>`
`std::atomic_int`	`std::atomic<int>`
`std::atomic_uint`	`std::atomic<unsigned int>`
`std::atomic_long`	`std::atomic<long>`
`std::atomic_ulong`	`std::atomic<unsigned long>`
`std::atomic_llong`	`std::atomic<long long>`
`std::atomic_ullong`	`std::atomic<unsigned long long>`
`std::atomic_char8_t` (C++20)	`std::atomic<char8_t>` (C++20)

---

[12]http://en.cppreference.com/w/cpp/header/cstdint

Type aliases for `std::atomic<bool>` and `std::atomic<integral type>`

Type alias	Definition
std::atomic_char16_t	std::atomic<char16_t>
std::atomic_char32_t	std::atomic<char32_t>
std::atomic_wchar_t	std::atomic<wchar_t>
std::atomic_int8_t	std::atomic<std::int8_t>
std::atomic_uint8_t	std::atomic<std::uint8_t>
std::atomic_int16_t	std::atomic<std::int16_t>
std::atomic_uint16_t	std::atomic<std::uint16_t>
std::atomic_int32_t	std::atomic<std::int32_t>
std::atomic_uint32_t	std::atomic<std::uint32_t>
std::atomic_int64_t	std::atomic<std::int64_t>
std::atomic_uint64_t	std::atomic<std::uint64_t>
std::atomic_int_least8_t	std::atomic<std::int_least8_t>
std::atomic_uint_least8_t	std::atomic<std::uint_least8_t>
std::atomic_int_least16_t	std::atomic<std::int_least16_t>
std::atomic_uint_least16_t	std::atomic<std::uint_least16_t>
std::atomic_int_least32_t	std::atomic<std::int_least32_t>
std::atomic_uint_least32_t	std::atomic<std::uint_least32_t>
std::atomic_int_least64_t	std::atomic<std::int_least64_t>
std::atomic_uint_least64_t	std::atomic<std::uint_least64_t>
std::atomic_int_fast8_t	std::atomic<std::int_fast8_t>
std::atomic_uint_fast8_t	std::atomic<std::uint_fast8_t>
std::atomic_int_fast16_t	std::atomic<std::int_fast16_t>
std::atomic_uint_fast16_t	std::atomic<std::uint_fast16_t>
std::atomic_int_fast32_t	std::atomic<std::int_fast32_t>
std::atomic_uint_fast32_t	std::atomic<std::uint_fast32_t>
std::atomic_int_fast64_t	std::atomic<std::int_fast64_t>
std::atomic_uint_fast64_t	std::atomic<std::uint_fast64_t>
std::atomic_intptr_t	std::atomic<std::intptr_t>
std::atomic_uintptr_t	std::atomic<std::uintptr_t>
std::atomic_size_t	std::atomic<std::size_t>
std::atomic_ptrdiff_t	std::atomic<std::ptrdiff_t>
std::atomic_intmax_t	std::atomic<std::intmax_t>
std::atomic_uintmax_t	std::atomic<std::uintmax_t>
std::atomic_signed_lock_free (C++20)	std::atomic<signed integral>
std::atomic_unsigned_lock_free (C++20)	std::atomic<unsigned integral>

The type aliases `atomic_signed_lock_free` and `atomic_unsigned_lock_free` are specializations of atomics whose template arguments are signed or unsigned integral types. Only implementations that support lock-free integral specializations provide these aliases. An implementation can choose the most efficient integral specialization.

2.3.4 All Atomic Operations

First, here is the list of all operations on atomics.

All operations on `atomic`

Member functions	Description
`test_and_set`	Atomically set the flag to `true` and returns the previous value.
`clear`	Atomically sets the flag to `false`.
`is_lock_free` `atomic_ref<T>::is_always_lock_free`	Checks if the atomic object is lock-free. Checks at compile time if the atomic type is always lock-free.
`load` `operator T`	Atomically returns the value of the atomic. Atomically returns the value of the atomic. Equivalent to `atom.load()`.
`store`	Atomically replaces the value of the atomic with a non-atomic.
`exchange`	Atomically replaces the value with the new value. Returns the old value.
`compare_exchange_strong` `compare_exchange_weak`	Atomically compares and eventually exchanges the value. Details are here.
`fetch_add, +=` `fetch_sub, -=`	Atomically adds (subtracts) the value. Returns the old value.
`fetch_or, |=` `fetch_and, &=` `fetch_xor, ^=`	Atomically performs bitwise (AND, OR, and XOR) operation with the value. Returns the old value.
`++, --`	Increments or decrements (pre- and post-increment) the atomic.
`notify_one` (C++20) `notify_all` (C++20)	Notifies one atomic wait operation. Notifies all atomic wait operations.
`wait` (C++20)	Blocks until it is notified. Compares itself with the `old` value to protect against spurious wakeups and lost wakeups. If the `old` value compares to unequal, returns.

The atomic types have no copy constructor or copy assignment operator, but they support an assignment from an implicit conversion to the underlying built-in type. The composite assignment operators return the new value; the fetch variations returns the old value. The composite assignment operators return values and not references to the assigned object.

Implicit conversion to the underlying type

```
std::atomic<long long> atomObj(2011);
atomObj = 2014;
long long nonAtomObj = atomObj;
```

Each member functions supports an additional memory-ordering argument. The default for the memory-ordering argument is `std::memory_order_seq_cst` but you can also use `std::memory_order_-relaxed`, `std::memory_order_consume`, `std::memory_order_acquire`, `std::memory_order_release`, or `std::memory_-order_acq_rel`. The `compare_exchange_strong` and `compare_exchange_weak` member functions can be parametrized with two memory-orderings. One for the success and one for the failure case. If you only explicitly provide one memory-ordering, it is used for the success and the failure case. Here are the details to memory-ordering.

Of course, not all operations are available on each atomic type. The table shows the list of the atomic operations depending on the atomic type.

All atomic operations depending on the atomic type

Member functions	atomic_flag	atomic<bool> atomic<user> atomic<smart T*>	atomic<floating>	atomic<T*>	atomic<integral>
test_and_set	yes				
clear	yes				
is_lock_free		yes	yes	yes	yes
atomic<T>::is_-always_lock_free		yes	yes	yes	yes
load		yes	yes	yes	yes
operator T		yes	yes	yes	yes
store		yes	yes	yes	yes
exchange		yes	yes	yes	yes
compare_exchange_-strong		yes	yes	yes	yes
compare_exchange_-weak		yes	yes	yes	yes
fetch_add, +=			yes	yes	yes
fetch_sub, -=			yes	yes	yes

All atomic operations depending on the atomic type

Member functions	atomic_flag	atomic<bool> atomic<user> atomic<smart T*>	atomic<floating>	atomic<T*>	atomic<integral>
`fetch_or, |=`					yes
`fetch_and, &=`					yes
`fetch_xor, ^=`					yes
`++, --`				yes	yes
`notify_one` (C++20)	yes	yes	yes	yes	yes
`notify_all` (C++20)	yes	yes	yes	yes	yes
`wait` (C++20)	yes	yes	yes	yes	yes

2.3.5 Free Atomic Functions

The functionality of the flag `std::atomic_flag` and the class template `std::atomic` can also be used with free functions. Because these functions use pointers instead of references they are compatible with C. The atomic free functions are available for the `std::atomic_flag` and the types such as the types you can use with the class template `std::atomic`.

The free functions for a `std::atomic_flag` are called: `std::atomic_clear()`, `std::atomic_clear_-explicit`, `std::atomic_flag_test_and_set()`, and `std::atomic_flag_test_set_explicit()`. The first argument of all four variations is a pointer to a `std::atomic_flag`. Additionally, the two `_exclicit` variations expect a memory-ordering.

For each `std::atomic` type there is a corresponding free function available. They free functions follow a straightforward naming convention: add just the prefix `atomic_` in front of it. For example, a member functions call `at.store()` on a `std::atomic` becomes `std::atomic_store()`, and `std::atomic_store_-explicit()`. The first overload expects, in this case, a pointer and the second overload, additionally, a memory-ordering.

For completeness, here are all overloads: atomic[13].

With one exception, free functions are only available on atomic types. The prominent exception to this rule is `std::shared_ptr`.

2.3.5.1 `std::shared_ptr` (deprecated in C++20)

`std::shared_ptr` is the only non-atomic data type on which you can apply atomic operations. First, let me write about the motivation for this exception.

The C++ committee saw the necessity that smart pointers should provide a minimum atomicity guarantee in multithreading programs. What is the meaning of the minimal atomicity guarantee

---

[13]http://en.cppreference.com/w/cpp/atomic

for `std::shared_ptr`? The control block of the `std::shared_ptr` is thread-safe. This means that the increase and decrease operations of the reference-counter are atomic. You also have the guarantee that the resource is destroyed exactly once.

The assertions that a `std::shared_ptr` provides, are described by Boab[14].

1. A shared_ptr instance can be "read" (accessed using only constant operations) simultaneously by multiple threads.

2. Different `shared_ptr` instances can be "written to" (accessed using mutable operations such as operator= or reset) simultaneously by multiple threads (even when these instances are copies and share the same reference count underneath).

To make the two statements clear, let me show a simple example. When you copy a `std::shared_ptr` in a thread, all is fine.

Thread-safe copying of a `std::shared_ptr`

```
1   std::shared_ptr<int> ptr = std::make_shared<int>(2011);
2
3   for (auto i = 0; i < 10; i++){
4       std::thread([ptr]{
5           std::shared_ptr<int> localPtr(ptr);
6           ptr = std::make_shared<int>(2014);
7       }).detach();
8   }
```

Let's first look at line 5. By using copy construction for the `std::shared_ptr localPtr`, only the control block is used. This is thread-safe. Line 6 is a little bit more interesting. The `localPtr` is set to a new `std::shared_ptr`. This is not a problem from the multithreading point of view: the lambda-function (line 4) binds `ptr` by copy. Therefore, the modification of `localPtr` takes place on a copy.

The story changes dramatically if I get the `std::shared_ptr` by reference.

A data race on a `std::shared_ptr`

```
1   std::shared_ptr<int> ptr = std::make_shared<int>(2011);
2
3   for (auto i = 0; i < 10; i++){
4       std::thread([&ptr]{
5           ptr = std::make_shared<int>(2014);
6       }).detach();
7   }
```

---

[14]http://www.boost.org/doc/libs/1_57_0/libs/smart_ptr/shared_ptr.htm#ThreadSafety

The lambda-expression binds the `std::shared_ptr` `ptr` in line 4 by reference. This means, the assignment (line 5) may become a concurrent reading and writing of the underlying resource; therefore, the program has undefined behavior.

Admittedly that last example was not very easy to achieve. `std::shared_ptr` requires special attention in a multithreading environment. `std::shared_ptr` is the only non-atomic data type in C+ for which atomic operations exist.

2.3.5.1.1 Atomic Operations on `std::shared_ptr`

There are specializations for the atomic operations `load`, `store`, `compare_and_exchange` for a `std::shared_-ptr`. By using the explicit variant, you can even specify the memory-ordering. Here are the free atomic operations for `std::shared_ptr`.

Atomic operations for `std::shared_ptr`

```
std::atomic_is_lock_free(std::shared_ptr)
std::atomic_load(std::shared_ptr)
std::atomic_load_explicit(std::shared_ptr)
std::atomic_store(std::shared_ptr)
std::atomic_store_explicit(std::shared_ptr)
std::atomic_exchange(std::shared_ptr)
std::atomic_exchange_explicit(std::shared_ptr)
std::atomic_compare_exchange_weak(std::shared_ptr)
std::atomic_compare_exchange_strong(std::shared_ptr)
std::atomic_compare_exchange_weak_explicit(std::shared_ptr)
std::atomic_compare_exchange_strong_explicit(std::shared_ptr)
```

For the details, have a look at cppreference.com[15]. It is quite easy to modify a shared pointer bound by reference in a thread-safe way.

A data race for a `std::shared_ptr` resolved

```
1  std::shared_ptr<int> ptr = std::make_shared<int>(2011);
2
3  for (auto i = 0;i < 10; i++){
4      std::thread([&ptr]{
5          auto localPtr = std::make_shared<int>(2014);
6          std::atomic_store(&ptr, localPtr);
7      }).detach();
8  }
```

The update of the `std::shared_ptr` `ptr` in the expression `std::atomic_store(&ptr, localPtr)` is thread-safe. All is well? NO! Finally, we need atomic smart pointers.

---

[15]http://en.cppreference.com/w/cpp/memory/shared_ptr

Atomic Smart Pointers with C++20

That is not the end of the story for atomic smart pointers. With C++20 we have two new smart pointers: `std::atomic<std::shared_ptr>` and `std::atomic<std::weak_ptr>`. I present them in the chapter about C++20: atomic smart pointers. They should be your first choice if possible.

Let's continue with C++20.

2.3.6 `std::atomic_ref` (C++20)

The class template `std::atomic_ref` applies atomic operations to the referenced object. Therefore, concurrent writing and reading of atomic object is no data race. The lifetime of the referenced object must exceed the lifetime of the `atomic_ref`. When any `atomic_ref` is accessing an object, all other accesses to the object must use an `atomic_ref`. In addition, no subobject of the `atomic_ref`-accessed object may be accessed by another `atomic_ref`.

2.3.6.1 Motivation

Stop. You may think that using a reference inside an atomic would do the job. Unfortunately not.

In the following program, I have a class `ExpensiveToCopy`, which includes a `counter`. A few threads concurrently increment the `counter`. Consequently, `counter` has to be protected.

Using an atomic reference

```cpp
// atomicReference.cpp

#include <atomic>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

struct ExpensiveToCopy {
    int counter{};
};

int getRandom(int begin, int end) {

    std::random_device seed;          // initial seed
    std::mt19937 engine(seed());      // generator
    std::uniform_int_distribution<> uniformDist(begin, end);

    return uniformDist(engine);
```

```
20    }
21
22    void count(ExpensiveToCopy& exp) {
23
24        std::vector<std::thread> v;
25        std::atomic<int> counter{exp.counter};
26
27        for (int n = 0; n < 10; ++n) {
28            v.emplace_back([&counter] {
29                auto randomNumber = getRandom(100, 200);
30                for (int i = 0; i < randomNumber; ++i) { ++counter; }
31            });
32        }
33
34        for (auto& t : v) t.join();
35
36    }
37
38    int main() {
39
40        std::cout << '\n';
41
42        ExpensiveToCopy exp;
43        count(exp);
44        std::cout << "exp.counter: " << exp.counter << '\n';
45
46        std::cout << '\n';
47
48    }
```

Variable exp (line 42) is the expensive-to-copy object. For performance reasons, the function count (line 22) takes exp by reference. Function count initializes the std::atomic<int> with exp.counter (line 25). The following lines create ten threads (line 27), each performing the lambda expression, which takes counter by reference. The lambda expression gets a random number between 100 and 200 (line 29) and increments the counter exactly as often. The function getRandom (line 13) starts with an initial seed and creates via the random number generator Mersenne Twister[16] a uniform distributed number between 100 and 200.

In the end, the exp.counter (line 44) should have an approximate value of 1500 because ten threads increment on average 150 times. Executing the program on the Wandbox online compiler[17] gives me a surprising result.

---

[16]https://en.wikipedia.org/wiki/Mersenne_Twister
[17]https://wandbox.org/

Surprise with an atomic reference

The counter is 0. What is happening? The issue is in line 25. The initialization in the expression `std::atomic<int> counter{exp.counter}` creates a copy. The following small program exemplifies the issue.

Copying the reference
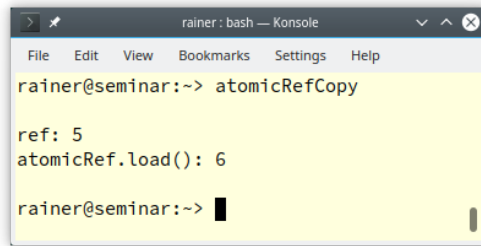
```cpp
// atomicRefCopy.cpp

#include <atomic>
#include <iostream>

int main() {

    std::cout << '\n';

    int val{5};
    int& ref = val;
    std::atomic<int> atomicRef(ref);
    ++atomicRef;
    std::cout << "ref: " << ref << '\n';
    std::cout << "atomicRef.load(): " << atomicRef.load() << '\n';

    std::cout << '\n';

}
```

The increment operation in line 13 does not address the reference `ref` (line 11). The value of `ref` is not changed.

Copying the reference

Replacing the `std::atomic<int> counter{exp.counter}` with `std::atomic_ref<int> counter{exp.counter}` solves the issue:

Using a `std::atomic_ref`

```cpp
// atomicReference.cpp

#include <atomic>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

struct ExpensiveToCopy {
    int counter{};
};

int getRandom(int begin, int end) {

    std::random_device seed;          // initial randomness
    std::mt19937 engine(seed());      // generator
    std::uniform_int_distribution<> uniformDist(begin, end);

    return uniformDist(engine);
}

void count(ExpensiveToCopy& exp) {

    std::vector<std::thread> v;
    std::atomic_ref<int> counter{exp.counter};

    for (int n = 0; n < 10; ++n) {
        v.emplace_back([&counter] {
            auto randomNumber = getRandom(100, 200);
            for (int i = 0; i < randomNumber; ++i) { ++counter; }
```

```
        });
    }

    for (auto& t : v) t.join();

}

int main() {

    std::cout << '\n';

    ExpensiveToCopy exp;
    count(exp);
    std::cout << "exp.counter: " << exp.counter << '\n';

    std::cout << '\n';

}
```
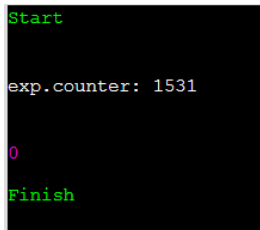
Now, the value of `counter` is as expected:

The expected result with `std::atomic_ref`

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

2.3.6.2 Specializations of `std::atomic_ref`

You can specialize `std::atomic_ref` for user-defined type, use partial specializations for pointer types, or full specializations for arithmetic types such as integral or floating-point types.

2.3.6.2.1 Primary Template

The primary template `std::atomic_ref` can be instantiated with a TriviallyCopyable type T.

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

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

2.3.6.2.2 Partial Specializations for Pointer Types

The standard provides partial specializations for pointer types: `std::atomic_ref<T*>`. Additionally to the following `std::atomic_ref<floating-point type>`, `std::atomic_ref<T*>` supports pre- and post-increment or pre- and post-decrement operations.

2.3.6.2.3 Specialization for Arithmetic Types

The standard provides specialization for the integral and floating-point types: `std::atomic_ref<arithmetic type>`.

- character types: `char`, `char8_t` (C++20), `char16_t`, `char32_t`, and `wchar_t`
- standard signed integer types: `signed char`, `short`, `int`, `long`, and `long long`
- standard unsigned integer types: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`
- additional integer types, defined in the header `<cstdint>` [18]:
  - `int8_t`, `int16_t`, `int32_t`, and `int64_t` (signed integer with exactly 8, 16, 32, and 64 bits)
  - `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t` (unsigned integer with exactly 8, 16, 32, and 64 bits)
  - `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, and `int_fast64_t` (fastest signed integer with at least 8, 16, 32, and 64 bits)
  - `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, and `uint_fast64_t` (fastest unsigned integer with at least 8, 16, 32, and 64 bits)
  - `int_least8_t`, `int_least16_t`, `int_least32_t`, and `int_least64_t` (smallest signed integer with at least 8, 16, 32, and 64 bits)
  - `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, and `uint_least64_t` (smallest unsigned integer with at least 8, 16, 32, and 64 bits)
  - `intmax_t`, and `uintmax_t` (maximum signed and unsigned integer)
  - `intptr_t`, and `uintptr_t` (signed and unsigned integer for holding a pointer)
- standard floating-point types: `float`, `double`, and `long double`

2.3.6.3 All Atomic Operations

First, here is the list of all operations on `atomic_ref`.

---

[18]http://en.cppreference.com/w/cpp/header/cstdint

All operations on `atomic_ref`

Member functions	Description
`is_lock_free` `atomic_ref<T>::is_always_lock_free`	Checks if the `atomic_ref` object is lock-free. Checks at compile time if the atomic type is always lock-free.
`load` `operator T`	Atomically returns the value of the referenced object. Atomically returns the value of the referenced object. Equivalent to `atomic.load()`.
`store`	Atomically replaces the value of the referenced object with a non-atomic.
`exchange`	Atomically replaces the value of the referenced object with the new value.

`compare_exchange_strong`
 	
`compare_exchange_weak`
	Atomically compares and eventually exchanges the value of the referenced object. Details are here.

`fetch_add, +=` `fetch_sub, -=`	Atomically adds(subtracts) the value to(from) the referenced object.

`fetch_or, |=`
 	
`fetch_and, &=`
`fetch_xor, ^=`
	Atomically performs bitwise OR, AND, and XOR) operation on the referenced object.

`++, --`	Increments or decrements (pre- and post-increment) the referenced object.
`notify_one` `notify_all`	Notifies one atomic wait operation. Notifies all atomic wait operations.
`wait`	Blocks until it is notified. Compares itself with the `old` value to protect against spurious wakeups and lost wakeups. If the `old` value compares to unequal, returns.

The composite assignment operators return the new value; the fetch variations return the old value.

Each member functions supports an additional memory-ordering argument. The default for the memory-ordering argument is `std::memory_order_seq_cst` but you can also use `std::memory_order_-relaxed`, `std::memory_order_consume`, `std::memory_order_acquire`, `std::memory_order_release`, or `std::memory_-order_acq_rel`. The `compare_exchange_strong` and `compare_exchange_weak` member functions can be parametrized with two memory-orderings. One for the success and one for the failure case. If you only explicitly provide one memory-ordering, it is used for the success and the failure case. Here are the details of memory-ordering.

Of course, not all operations are available on all types referenced by `std::atomic_ref`. The table shows

the list of all atomic operations depending on the type referenced by `std::atomic_ref`.

All atomic operations, depending on the type referenced by `std::atomic_ref`

Function	atomic_ref<T>	atomic_ref<floating>	atomic_ref<T*>	atomic_ref<integral>
`is_lock_free`	yes	yes	yes	yes
`atomic_ref<T>::is_lock_-` `free`	yes	yes	yes	yes
`load`	yes	yes	yes	yes
`operator T`	yes	yes	yes	yes
`store`	yes	yes	yes	yes
`exchange`	yes	yes	yes	yes
`compare_exchange_strong`	yes	yes	yes	yes
`compare_exchange_weak`	yes	yes	yes	yes
`fetch_add, +=`		yes	yes	yes
`fetch_sub, -=`		yes	yes	yes
`fetch_or, |=`				yes
`fetch_and, &=`				yes
`fetch_xor, ^=`				yes
`++, --`			yes	yes
`notify_one`	yes	yes	yes	yes
`notify_all`	yes	yes	yes	yes
`wait`	yes	yes	yes	yes

Atomics and their atomic operations are the basic building blocks for the memory model. They establish synchronization and ordering constraints that hold for both atomics and non-atomics. Let's have a more in-depth look into the synchronization and ordering constraints.

2.4 The Synchronization and Ordering Constraints

You cannot configure the atomicity of an atomic data type, but you can accurately adjust the synchronization and ordering constraints of atomic operations. Changing the synchronization and ordering constraints is a possibility that is unique to C++ and is not possible in C#'s or Java's memory model.

There are six different variants of the memory model in C++. The critical question is what their characteristics are?

2.4.1 The Six Variants of Memory Orderings in C++

We already know C++ has six variants of memory ordering. The default for atomic operations is `std::memory_order_seq_cst`. This expression stands for sequential consistency. Besides, you can explicitly specify one of the other five. So what does C++ have to offer?

The memory orderings

```cpp
enum memory_order{
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
}
```

To classify these six memory ordering, it helps to answer two questions:

1. Which kind of atomic operations should use which memory model?
2. Which synchronization and ordering constraints are defined by the six variants?

My plan is quite simple: I answer both questions.

2.4.1.1 Kind of Atomic Operation

There are three different kinds of operations:

- Read operations: `memory_order_acquire` and `memory_order_consume`
- Write operations: `memory_order_release`
- Read-modify-write operations: `memory_order_acq_rel` and `memory_order_seq_cst`

`memory_order_relaxed` defines no synchronization and ordering constraints. It does not fit in this taxonomy.

The following table orders the atomic operations based on their reading and writing characteristics.

Characteristics of Atomic Operations

Operation	read	write	read-modify-write
test_and_set			yes
clear		yes	
is_lock_free	yes		
load	yes		
operator T	yes		
store		yes	
exchange			yes
compare_exchange_strong			yes
compare_exchange_weak			yes
fetch_add, +=			yes
fetch_sub, -=			
fetch_or, |=			yes
fetch_and, &=			
fetch_xor, ^=			
++, --			yes
notify_one		yes	
notify_all		yes	
wait	yes		

Read-modify-write operations have an additional guarantee: they always provide the newest value. This means a sequence of `atomVar.fetch_sub(1)` operations on different threads counts down one after the other without any gaps or duplicates.

If you use an atomic operation `atomVar.load()` with a memory model that is designed for a write or read-modify-write operation, the write part has no effect. The result is that operation `atomVar.load(std::memory_order_acq_rel)` is equivalent to operation `atomVar.load(std::memory_order_acquire)`. Accordingly, operation `atomVar.load(std::memory_order_release)` is equivalent to `atomVar.load(std::memory_order_relaxed)`.

2.4.1.2 Different Synchronization and Ordering Constraints

There are, roughly speaking, three different types of synchronization and ordering constraints in C++:

- Sequential consistency: `memory_order_seq_cst`
- Acquire-release: `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, and `memory_order_acq_rel`
- Relaxed: `memory_order_relaxed`

While the sequential consistency establishes a global order between threads, the acquire-release semantic establishes an ordering between reading and writing operations on the same atomic variable with different threads. The relaxed semantic only guarantees the modification order of some atomic `m`. Modification order means that all modifications on a particular atomic `m` occur in some particular total order. Consequently, a particular thread reads of an atomic object never see "older" values than those the thread has already observed.

The different memory models and their effects on atomic and non-atomic operations make the C++ memory model an exciting and challenging topic. Let us discuss the synchronization and ordering constraints of the sequential consistency, the acquire-release semantic, and the relaxed semantic.

2.4.2 Sequential Consistency

Let us dive deeper into sequential consistency. The key to sequential consistency is that all operations on all threads obey a universal clock. This global clock makes it quite intuitive to think about it.

The intuitiveness of the sequential consistency comes with a price. The downside is that the system has to synchronize threads.

The following program synchronizes the producer and the consumer thread with the help of sequential consistency.

Producer-Consumer synchronization with sequential consistency

```cpp
1  // producerConsumer.cpp
2
3  #include <atomic>
4  #include <iostream>
5  #include <string>
6  #include <thread>
7
8  std::string work;
9  std::atomic<bool> ready(false);
10
11 void consumer(){
12   while(!ready.load()){}
13   std::cout<< work << '\n';
14 }
15
16 void producer(){
17   work= "done";
```

```
18     ready=true;
19   }
20
21   int main(){
22     std::thread prod(producer);
23     std::thread con(consumer);
24     prod.join();
25     con.join();
26   }
```

The output of the program is not very exciting.
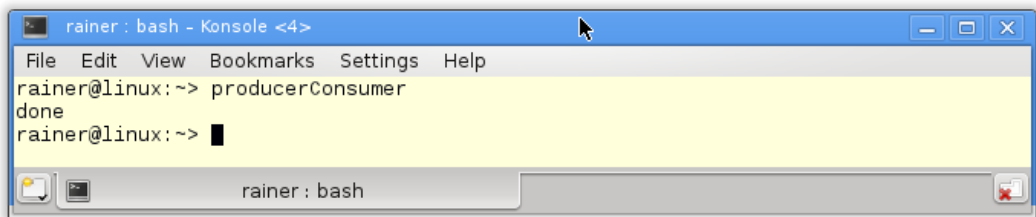
Producer-Consumer synchronization with sequential consistency

Because of sequential consistency, the program execution is deterministic. Its output is always "done".

The graphic depicts the sequence of operations. The consumer thread waits in the while-loop until the atomic variable ready is set to true. When this happens, the consumer threads continue their work.

Execution order of the sequential consistency
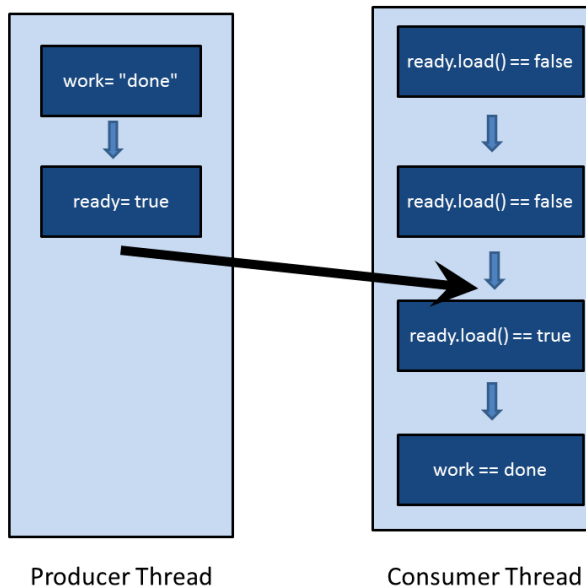
It is relatively easy to understand that the program always returns "done". We only have to use the two characteristics of sequential consistency. On the one hand, both threads execute their instructions in source code order; on the other hand, each thread sees the other thread's operations in the same order. Both threads follow the same universal clock. This synchronization does also hold - with the help of the `while(!ready.load()){}` loop - for the synchronization of the producer and the consumer thread.

I can explain the reasoning a lot more formally by using the terminology of the memory ordering. Here is the formal version:

1. `work= "done"` is *sequenced-before* `ready = true`
   ⇒ work= `"done"` *happens-before* `ready = true`

2. `while(!ready.load()){}` is *sequenced-before* `std::cout << work << '\n'`
   ⇒ `while(!ready.load()){}` *happens-before* `std::cout<< work << '\n'`

3. `ready= true` *synchronizes-with* `while(!ready.load()){}`
   ⇒ `ready= true` *inter-thread happens-before* `while (!ready.load()){}`
   ⇒ `ready= true` *happens-before* `while (!ready.load()){}`

The conclusion: because the *happens-before* relation is transitive, it follows `work = "done"` *happens-before* `ready= true` *happens-before* `while(!ready.load()){}` *happens-before* `std::cout<< work << '\n'`

In sequential consistency, a thread sees another thread's operations and, therefore of all other threads in the same order. The critical characteristic of sequential consistency does not hold if we use the

acquire-release semantic for atomic operations. The acquire-release semantic is an area where C#
and Java do not follow. That's also an area where our intuition begins to wane.

2.4.3 Acquire-Release Semantic

There is no global synchronization between threads in the acquire-release semantic; there is only
synchronization between atomic operations **on the same atomic variable**. A write operation on one
thread synchronizes with a read operation on another thread on the same atomic variable.

The acquire-release semantic is based on one fundamental idea: a release operation synchronizes
with an acquire operation on the same atomic and establishes an ordering constraint. The ordering
constraint means all read and write operations cannot be moved after a release operation. All read
and write operations cannot be moved before an acquire operation.

What is an acquire or a release operation? The reading of an atomic variable with `load` or `test_and_set`
is an acquire operation. There is more: releasing a lock or mutex *synchronizes-with* the acquiring of
a lock or a mutex. The construction of a thread *synchronizes-with* the invocation of the callable. The
completion of the thread *synchronizes-with* the `join`-call. The completion of the callable of the task
synchronizes-with the call to `wait` or `get` in the future. Acquire and release operations come in pairs.

It helps a lot to keep that picture in mind.
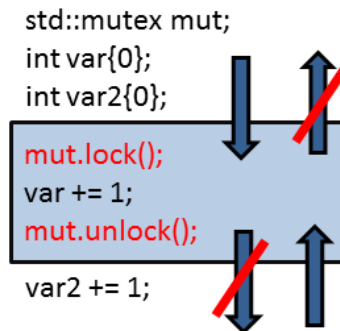
The critical region

🔑 The memory model for a deeper understanding of multithreading

This is the main reason you should keep the memory model in mind. Acquire-release
semantic helps you get a better understanding of the high-level synchronization primitives
such as a mutex. The same reasoning holds for the starting of a thread and the join-call
on a thread. Both are acquire-release operations. The story goes on with the `wait` and
`notify_one` call on a condition variable; `wait` is the acquire, and `notify_one` the release
operation. What's about `notify_all`? That is a release operation as well.

Now, let us look once more at the spinlock in the subsection std::atomic_flag. We can write it more efficiently because the synchronization is done with the atomic_flag `flag`. Therefore the acquire-release semantic applies.

A Spinlock with acquire-release semantic

```
1   // spinlockAcquireRelease.cpp
2
3   #include <atomic>
4   #include <thread>
5
6   class Spinlock{
7     std::atomic_flag flag;
8   public:
9     Spinlock(): flag(ATOMIC_FLAG_INIT) {}
10
11    void lock(){
12      while(flag.test_and_set(std::memory_order_acquire));
13    }
14
15    void unlock(){
16      flag.clear(std::memory_order_release);
17    }
18  };
19
20  Spinlock spin;
21
22  void workOnResource(){
23    spin.lock();
24    // shared resource
25    spin.unlock();
26  }
27
28
29  int main(){
30
31    std::thread t(workOnResource);
32    std::thread t2(workOnResource);
33
34    t.join();
35    t2.join();
36
37  }
```

The `flag.clear` call in line 16 is a release, the `flag.test_and_set` call in line 12 an acquire operation, and the acquire synchronizes with the release operation. The heavyweight synchronization

of two threads with sequential consistency (`std::memory_order_seq_cst`) is replaced by the more lightweight and performant acquire-release semantic (`std::memory_order_acquire` and `std::memory_order_release`). The behavior is not affected.

Although the `flag.test_and_set(std::memory_order_acquire)` call is a read-modify-write operation, the acquire semantic is sufficient. In summary, `flag` is an atomic and guarantees, therefore, modification order. This means all modifications to `flag` occur in some particular total order.

The acquire-release semantic is transitive. That means if you have an acquire-release semantic between two threads (a, b) and an acquire-release semantic between (b, c), you get an acquire-release semantic between (a, c).

2.4.3.1 Transitivity

A release operation synchronizes with an acquire operation on the same atomic variable and establishes ordering constraint. These are the components to synchronize threads in a performant way if they act on the same atomic. How can that work if two threads share no atomic variable? We do not want any sequential consistency because that is too expensive, but we want the light-weight acquire-release semantic.

The answer to this question is straightforward. Applying the transitivity of the acquire-release semantic, we can synchronize independent threads.

In the following example, thread `t2` with its work package `deliveryBoy` is the connection between two independent threads `t1` and `t3`.

Transitivity of the acquire-release semantics

```cpp
1   // transitivity.cpp
2
3   #include <atomic>
4   #include <iostream>
5   #include <thread>
6   #include <vector>
7
8   std::vector<int> mySharedWork;
9   std::atomic<bool> dataProduced(false);
10  std::atomic<bool> dataConsumed(false);
11
12  void dataProducer(){
13      mySharedWork = {1,0,3};
14      dataProduced.store(true, std::memory_order_release);
15  }
16
17  void deliveryBoy(){
18      while(!dataProduced.load(std::memory_order_acquire));
19      dataConsumed.store(true, std::memory_order_release);
```

```
20   }
21
22   void dataConsumer(){
23       while(!dataConsumed.load(std::memory_order_acquire));
24       mySharedWork[1] = 2;
25   }
26
27   int main(){
28
29     std::cout << '\n';
30
31     std::thread t1(dataConsumer);
32     std::thread t2(deliveryBoy);
33     std::thread t3(dataProducer);
34
35     t1.join();
36     t2.join();
37     t3.join();
38
39     for (auto v: mySharedWork){
40         std::cout << v << " ";
41     }
42
43     std::cout << "\n\n";
44
45   }
```

The output of the program is deterministic. `mySharedWork` has the values 1,2, and 3.
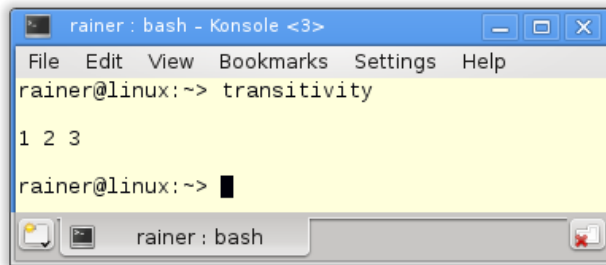
Output of the program transitivity.cpp

There are two critical observations:

1. Thread `t2` waits in line 18, until thread `t3` sets `dataProduced` to `true` (line 14).

2. Thread `t1` waits in line 23, until thread `t2` sets `dataConsumed` to `true` (line 19).

Let me explain the rest with a graphic.

```cpp
void dataProducer(){
    mySharedWork={1,0,3};
    dataProduced.store(true, std::memory_order_release);
}

void deliveryBoy(){
    while( !dataProduced.load(std::memory_order_acquire) );
    dataConsumed.store(true,std::memory_order_release);
}

void dataConsumer(){
    while( !dataConsumed.load(std::memory_order_acquire) );
    mySharedWork[1]= 2;
}
```

sequenced-before
synchronizes-with
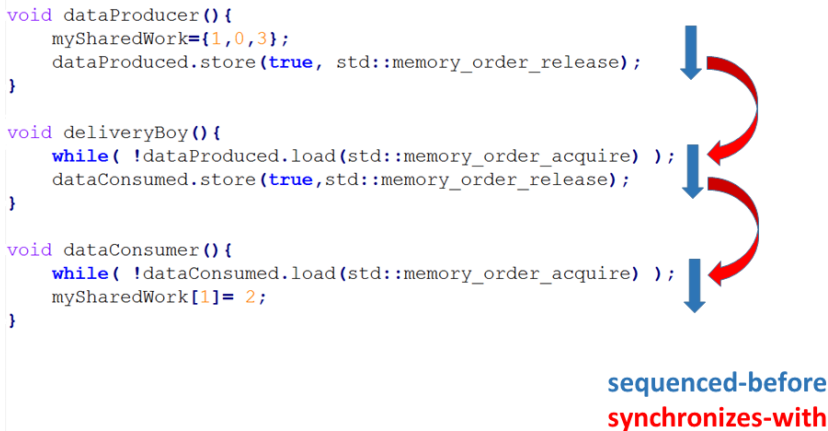
<div align="center">Transitivity of the acquire-release semantic</div>

The essential parts of the picture are the arrows.

- The **blue** arrows are the *sequenced-before* relations. This means that all operations in one thread are executed in source code order.

- The **red** arrows are the *synchronizes-with* relations. The reason is the acquire-release semantic of the atomic operations on the same atomic. The synchronization between the atomics, and therefore between the threads happen at specific points.

- *sequenced-before* establishes a *happens-before* and *synchronizes-with* a *inter-thread happens-before* relation.

The rest is pretty simple. The *happens-before* and *inter-thread happens-before* order of the instructions correspond to the arrows' direction from top to bottom. Finally, we guarantee that mySharedWork[1] == 2 is executed last.

A release operation *synchronizes-with* an acquire operation on the same atomic variable, so we can easily synchronize threads, **if** ... . The typical misunderstanding is about the **if**.

2.4.3.2 The Typical Misunderstanding

What is my motivation for writing about the typical misunderstanding of the acquire-release semantic? Many of my readers and students have already fallen into this trap. Let's look at the straightforward case.

2.4.3.2.1 Waiting Included

Here is a simple program as a starting point.

Acquire-release with waiting

```cpp
// acquireReleaseWithWaiting.cpp

#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> mySharedWork;
std::atomic<bool> dataProduced(false);

void dataProducer(){
    mySharedWork = {1, 0, 3};
    dataProduced.store(true, std::memory_order_release);
}

void dataConsumer(){
    while( !dataProduced.load(std::memory_order_acquire) );
    mySharedWork[1] = 2;
}

int main(){

  std::cout << '\n';

  std::thread t1(dataConsumer);
  std::thread t2(dataProducer);

  t1.join();
  t2.join();

  for (auto v: mySharedWork){
      std::cout << v << " ";
  }

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

}
```

The consumer thread t1 in line 17 waits until the producer thread t2 in line 13 sets dataProduced to true. dataProduced is the guard, and it guarantees that access to the non-atomic variable mySharedWork is synchronized. Synchronized means that the producer thread t2 initializes mySharedWork, then the consumer thread t1 finishes the work by setting mySharedWork[1] to 2. The program is well-defined.

Execution of the acquireReleaseWithWaiting program

The graphic shows the *happens-before* relation within the threads and the *synchronizes-with* relation between the threads. *synchronizes-with* establishes an *inter-thread happens-before* relation. The rest of the reasoning is the transitivity of the *happens-before* relation.

Finally it holds that mySharedWork = {1, 0, 3} *happens-before* mySharedWork[1] = 2.

Acquire-release semantic with waiting

What aspect is often missing in this reasoning? The **if**.

2.4.3.2.2 If ...

What happens if the consumer thread t1 in line 17 doesn't wait for the producer thread t2?
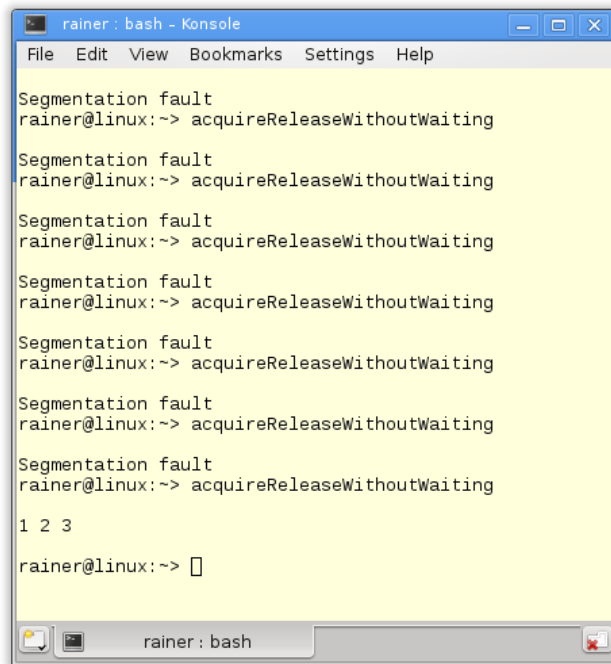
Acquire-release without waiting

```cpp
1   // acquireReleaseWithoutWaiting.cpp
2
3   #include <atomic>
4   #include <iostream>
5   #include <thread>
6   #include <vector>
7
8   std::vector<int> mySharedWork;
9   std::atomic<bool> dataProduced(false);
10
11  void dataProducer(){
12      mySharedWork = {1, 0, 3};
13      dataProduced.store(true, std::memory_order_release);
14  }
15
16  void dataConsumer(){
17      dataProduced.load(std::memory_order_acquire);
18      mySharedWork[1] = 2;
19  }
20
21  int main(){
22
23    std::cout << '\n';
24
25    std::thread t1(dataConsumer);
26    std::thread t2(dataProducer);
27
28    t1.join();
29    t2.join();
30
31    for (auto v: mySharedWork){
32        std::cout << v << " ";
33    }
34
35    std::cout << "\n\n";
36
37  }
```

The program has undefined behavior because there is a data race on the variable mySharedWork. When we let the program run, we get the following non-deterministic behavior.

Undefined behavior with the acquire-release semantic

What is the issue? It holds that `dataProduced.store(true, std::memory_order_release)` *synchronizes-with* `dataProduced.load(std::memory_order_acquire)`. But that doesn't mean the acquire operation waits for the release operation, and that is exactly what is displayed in the graphic. In the graphic the `dataProduced.load(std::memory_order_acquire)` instruction is performed before the instruction `dataProduced.store(true, std::memory_order_release)`. We have no *synchronizes-with* relation.

time
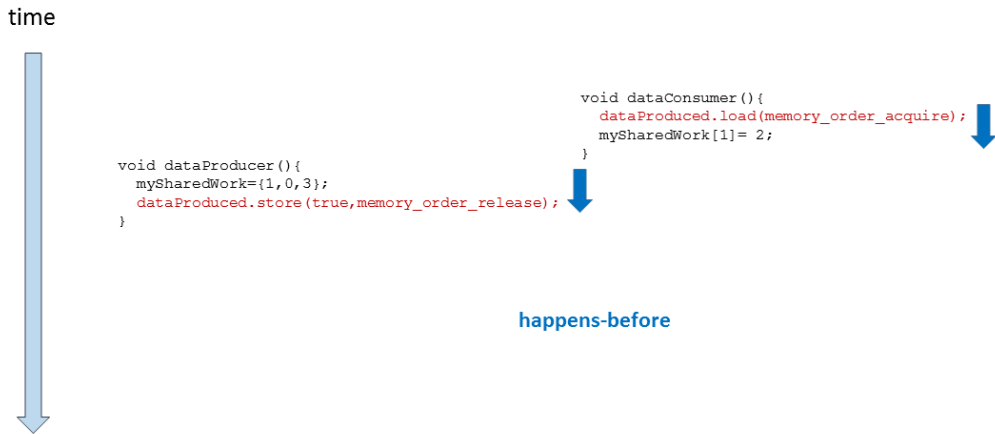
Acquire-release semantic without waiting

2.4.3.2.3 The Solution

synchronizes-with means: if `dataProduced.store(true, std::memory_order_release)` happens before `dataProduced.load(std::memory_order_acquire)`, then all visible effects of the operations before `dataProduced.store(true, std::memory_order_release)` are visible after `dataProduced.load(std::memory_order_acquire)`. The key is the word **if**. That **if** is guaranteed in the first program with the predicate (`while(!dataProduced.load(std::memory_order_acquire))`).

Here it comes once again, but more formally.

All operations before `dataProduced.store(true, std::memory_order_release)` *happens-before* all operations after `dataProduced.load(std::memory_order_acquire)`, if the following holds: `dataProduced.store(true, std::memory_order_release)` *happens-before* `dataProduced.load(std::memory_order_acquire)`.

2.4.3.3 Release Sequence

A release sequence is quite an advanced concept when dealing with acquire-release semantic. So let first start with the acquire-release semantic in the following example.

A release sequence

```cpp
// releaseSequence.cpp

#include <atomic>
#include <thread>
#include <iostream>
#include <mutex>

std::atomic<int> atom{0};
int somethingShared{0};

using namespace std::chrono_literals;

void writeShared(){
    somethingShared = 2011;
    atom.store(2, std::memory_order_release);
}

void readShared(){
    while ( !(atom.fetch_sub(1, std::memory_order_acquire) > 0) ){
        std::this_thread::sleep_for(100ms);
    }

    std::cout << "somethingShared: " << somethingShared << '\n';
}

int main(){

    std::cout << '\n';

    std::thread t1(writeShared);
    std::thread t2(readShared);
    // std::thread t3(readShared);

    t1.join();
    t2.join();
    // t3.join();

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

    std::cout << '\n';

}
```
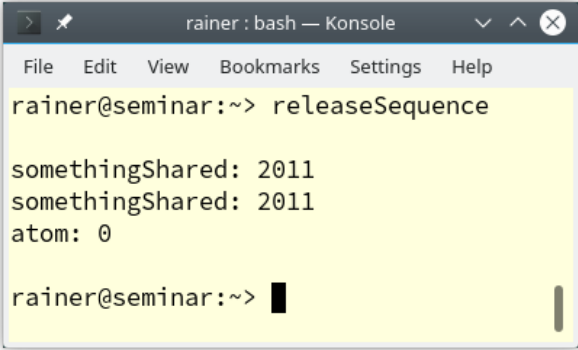
Let's first look at the example without thread `t3`. The atomic store on line 15 *synchronizes-with* the atomic load in line 19. The synchronization guarantees that all that happens before the store is available after the load. This means, in particular, that the access to the non-atomic variable `somethingShared` is not a data race.

What changes if I use the thread `t3`? Now there seems to be a data race. As I already mentioned, the first call to `atom.fetch_sub(1, std::memory_order_acquire)` (line 19) has an acquire-release semantic with `atom.store(2, std::memory_order_release)` (line 15); therefore, there is no data race on `somethingShared`.

This does not hold for the second call to `atom.fetch_sub(1, std::memory_order_acquire)`. It is a ready-modify-write operation without a `std::memory_order_release` tag. It means in particular, that the second call to `atom.fetch_sub(1, std::memory_order_acquire)` does not synchronize-with the first call, and a data race may occur on `sharedVariable`. May because thanks to the release sequence, this does not happen. The release sequence is extended to the second call to `atom.fetch_sub(1, std::memory_-order_acquire)`; therefore, the second call `atom.fetch_sub(1, std::memory_order_acquire)` *has a happens-before* relation with the first call.

Finally, here is the output of the program.

A release sequence

More formally, the N4659: Working Draft, Standard for Programming Language C++[19].

🔑 Release Sequence

A release sequence headed by a release operation A on an atomic object M is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first operation is A, and every subsequent operation * is performed by the same thread that performed A, or * is an atomic read-modify-write operation.

---

[19]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf

If you carefully follow my explanation such as the one in the subsection Challenges, you probably expect Relaxed Semantic to come next; however I'll look first at the memory model `std::memory_-order_consume` which is quite similar to `std::memory_order_acquire`.

2.4.4 `std::memory_order_consume`

`std::memory_order_consume` is the most legendary of the six memory orderings. For two reasons: first, `std::memory_order_consume` is extremely hard to understand, and second - and this may change in the future - no compiler supports it currently. With C++17 the situation gets even worse. Here is the official wording: "The specification of release-consume ordering is being revised, and the use of memory_order_consume is temporarily discouraged."

How can it be that a compiler that implements the C++11 standard doesn't support the memory model `std::memory_order_consume`? The answer is that the compiler maps `std::memory_order_consume` to `std::memory_order_acquire`. This mapping is acceptable because both are load or acquire operations. `std::memory_order_consume` requires weaker synchronization and ordering constraints than `std::memory_order_acquire`. Therefore, the release-acquire ordering is potentially slower than the release-consume ordering but - and this is the key point - *well-defined.*

To understand the release-consume ordering, it is a good idea to compare it with the release-acquire ordering. I speak in the following subsection explicitly about the release-acquire ordering and not about the acquire-release semantic to emphasize the strong relationship of `std::memory_order_consume` and `std::memory_order_acquire`.

2.4.4.1 Release-acquire ordering

As a starting point, let us use the following program with two threads `t1` and `t2`. `t1` plays the producer's role, `t2` the consumer's role. The atomic variable `ptr` helps to synchronize the producer and consumer.

Release-acquire ordering

```cpp
// acquireRelease.cpp

#include <atomic>
#include <thread>
#include <iostream>
#include <string>

using namespace std;

atomic<string*> ptr;
int data;
atomic<int> atoData;

void producer(){
    string* p  = new string("C++11");
```

```
16          data = 2011;
17          atoData.store(2014, memory_order_relaxed);
18          ptr.store(p, memory_order_release);
19      }
20
21      void consumer(){
22          string* p2;
23          while (!(p2 = ptr.load(memory_order_acquire)));
24          cout << "*p2: " << *p2 << '\n';
25          cout << "data: " << data << '\n';
26          cout << "atoData: " << atoData.load(memory_order_relaxed) << '\n';
27      }
28
29      int main(){
30
31          cout << '\n';
32
33          thread t1(producer);
34          thread t2(consumer);
35
36          t1.join();
37          t2.join();
38
39          cout << '\n';
40
41      }
```

Before analyzing the program, I want to introduce a small variation.

2.4.4.2 Release-consume ordering

I replace the memory order `std::memory_order_acquire` in line 23 with `std::memory_order_consume`.

Release-consume ordering

```
1   // acquireConsume.cpp
2
3   #include <atomic>
4   #include <thread>
5   #include <iostream>
6   #include <string>
7
8   using namespace std;
9
10  atomic<string*> ptr;
```

```
11    int data;
12    atomic<int> atoData;
13
14    void producer(){
15        string* p  = new string("C++11");
16        data = 2011;
17        atoData.store(2014,memory_order_relaxed);
18        ptr.store(p, memory_order_release);
19    }
20
21    void consumer(){
22        string* p2;
23        while (!(p2 = ptr.load(memory_order_consume)));
24        cout << "*p2: " << *p2 << '\n';
25        cout << "data: " << data << '\n';
26        cout << "atoData: " << atoData.load(memory_order_relaxed) << '\n';
27    }
28
29    int main(){
30
31        cout << '\n';
32
33        thread t1(producer);
34        thread t2(consumer);
35
36        t1.join();
37        t2.join();
38
39        cout << '\n';
40
41    }
```

Now the program has undefined behavior. This statement is very hypothetical because my GCC 5.4 compiler implements `std::memory_order_consume` by using `std::memory_order_acquire`. Under the hood, both programs do the same thing.

2.4.4.3 Release-acquire versus Release-consume ordering

The outputs of the programs are identical.

Release-acquire and release-consume ordering

At the risk of repeating myself, I want to add a few words explaining why the first program, `acquireRelease.cpp` is well-defined.

The store operation on line 17 *synchronizes-with* the load operation in line 23. The reason is that the store operation uses `std::memory_order_release` and the load operation uses `std::memory_order_-acquire`. This store/load relation is the synchronization. What are the ordering constraints for the release-acquire operations? The release-acquire ordering guarantees that all operations' results before the store operation (line 16) are available after the load operation (line 21). So also, the release-acquire operation orders access to the non-atomic variable `data` (line 14) and the atomic variable `atoData` (line 15). That holds although `atoData` uses the `std::memory_order_relaxed` memory ordering.

The crucial question is: what happens if I replace `std::memory_order_acquire` with `std::memory_-order_consume`?

2.4.4.4 Data dependencies with `std::memory_order_consume`

`std::memory_order_consume` deals with data dependencies on atomics. Data dependencies exist in two ways. First, let us look at *carries-a-dependency-to* in a thread and *dependency-ordered before* between two threads. Both dependencies introduce a *happens-before* relation. These are the kind of relations we are looking for. What does *carries-a-dependency-to* and *dependency-order-before* mean?

- *carries-a-dependency-to*: if the result of operation A is used as an operand in operation B, then: A *carries-a-dependency-to* B.
- *dependency-ordered-before*: a store operation (with *std::memory_order_release*, *std::memory_-order_acq_rel*, or *std::memory_order_seq_cst*) is *dependency-ordered-before* a load operation B (with *std::memory_order_consume*) if the result of load operation B is used in a further operation C in the same thread. It is important to note that operations B and C have to be in the same thread.

I know from personal experience that both definitions might not be easy to digest. Here is a graphic to visualize them.

```cpp
std::atomic<std::string*> ptr;
int data;
std::atomic<int> atoData;

void producer(){
    std::string* p  = new std::string("C++11");
    data = 2011;
    atoData.store(2014,std::memory_order_relaxed);
    ptr.store(p, std::memory_order_release);
}

void consumer(){
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_consume)));
    std::cout << "*p2: " << *p2 << std::endl;
    std::cout << "data: " << data << std::endl;
    std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
}
```

dependency-ordered-before
carries-a-dependency-to

Data dependencies of std::memory_order_consume

The expression `ptr.store(p, std::memory_order_release)` is *dependency-ordered-before* the expression `while (!(p2 = ptr.load(std::memory_order_consume)))`, because the following line `std::cout << "*p2: " << *p2 << '\n'` is be read as the result of the load operation. Furthermore it holds that: `while (!(p2 = ptr.load(std::memory_order_consume))` *carries-a-dependency-to* `std::cout << "*p2: " << *p2 << '\n'`, because the output of `*p2` uses the result of the `ptr.load` operation.

We have no guarantee regarding the output of `data` and `atoData`. That's because neither has a *carries-a-dependency* relation to the `ptr.load` operation. It gets even worse: since `data` is a non-atomic variable, there is a data race on the variable `data`. The reason is that both threads can access data at the same time, and thread t1 wants to modify `data`. Therefore the program has undefined behavior.

Finally, we have reached the relaxed semantic.

2.4.5 Relaxed Semantic

The relaxed semantic is the other end of the spectrum. The relaxed semantic is the weakest of all memory models and only guarantees the modification order of atomics. This means all modifications on an atomic happen in some particular total order.

2.4.5.1 No synchronization and ordering constraints

It is relatively easy. If there are no rules, we cannot violate them. However, that is too easy; the program should have *well-defined* behavior. *Well-defined* behavior means that you typically use synchronization and ordering constraints of stronger memory orderings to control operations with relaxed semantic. How does this work? A thread can see the effects of another thread in arbitrary order, so you have to make sure there are points in your program where all operations on all threads get synchronized.

A typical example of an atomic operation, in which the sequence of operations doesn't matter, is a counter. The critical observation for a counter is not how the different threads increment the counter; the critical observation in a counter is that all increments are atomic and that all increments are done at the end. Have a look at the following example.

A counter with relaxed semantic

```cpp
// relaxed.cpp

#include <vector>
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<int> count = {0};

void add()
{
    for (int n = 0; n < 1000; ++n) {
        count.fetch_add(1, std::memory_order_relaxed);
    }
}

int main()
{
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n) {
        v.emplace_back(add);
    }
    for (auto& t : v) {
        t.join();
    }
    std::cout << "Final counter value is " << count << '\n';
}
```

The three most exciting lines are 13, 24, and 26.

In line 13, the atomic number `count` is incremented using the relaxed semantic, so we have a guarantee that the operation is atomic. The `fetch_add` operation establishes an ordering on `count`. The function `add` (lines 10 - 15) is the work package of the threads. Each thread gets its work package on line 21.

Thread creation is one synchronization point - the other one being `t.join()` on line 24.

The creator thread synchronizes with all its children in line 24. It waits with the `t.join()` call until all its children are done. `t.join()` is why the results of the atomic operations are published. To say it more formally: `t.join()` is a release operation.

In conclusion, there is a *happens-before* relation between the increment operation in line 13 and the reading of the counter `count` in line 26.

The result is that the program always returns 10000. Boring? No, it's reassuring!

The atomic counter with relaxed semantic

A typical example of an atomic counter which uses the relaxed semantic is the reference counter of `std::shared_ptr`. The relaxed semantic only holds for the increment operation. The critical property for incrementing the reference counter is that the operation is atomic. The order of the increment operations does not matter. The relaxed semantic does not hold for the decrement of the reference counter. These operations need an acquire-release semantic for the destructor.

The `add` algorithm is wait-free

Have a closer look at the function `add` in line 10. There is no synchronization involved in the increment operation (line 13). The value 1 is just added to the atomic `count`.

Therefore, the algorithm is not only lock-free but it is also wait-free.

The fundamental idea of `std::atomic_thread_fence` is to establish synchronization and ordering constraints between threads without an atomic operation.

2.5 Fences

C++ support two kind of fences: a `std::atomic_thread_fence` and a `std::atomic_signal_fence`.

- `std::atomic_thread_fence`: synchronises memory accesses between threads.
- `std::atomic_signal_fence`: synchronizes between a signal handler and code running on the same thread.

2.5.1 `std::atomic_thread_fence`

A `std::atomic_thread_fence` prevents specific operations from crossing a fence.

`std::atomic_thread_fence` needs no atomic variable. They are frequently just referred to as fences or memory barriers. You quickly get an idea of what a `std::atomic_thread_fence` is all about.

2.5.1.1 Fences as Memory Barriers

What does that mean? Specific operations cannot cross a memory barrier. What kind of operations? From a bird's-eye view, we have two kinds of operations: read and write or load and store operations. The expression `if(resultRead) return result` is a load, followed by a store operation.

There are four different ways to combine load and store operations:

- **LoadLoad**: A load followed by a load.
- **LoadStore**: A load followed by a store.
- **StoreLoad**: A store followed by a load.
- **StoreStore**: A store followed by a store.

Of course, there are more complicated operations consisting of multiple load and stores (`count++`), and these operations fall into my general classification.

What about memory barriers? If you place memory barriers between two operations like LoadLoad, LoadStore, StoreLoad, or StoreStore, you have the guarantee that specific LoadLoad, LoadStore, StoreLoad, or StoreStore operations are not be reordered. The risk of reordering is always present if non-atomics or atomic operations with relaxed semantic are used.

2.5.1.2 The Three Fences

Typically, three kinds of fences are used. They are called a full fence, acquire fence, and release fence. As a reminder, acquire is a load, and release is a store operation. What happens if I place one of the three memory barriers between the four combinations of load and store operations?

- **Full fence**: A full fence `std::atomic_thread_fence()` between two arbitrary operations prevents these operations' reordering with one exception: StoreLoad operations can be reordered.

- **Acquire fence**: An acquire fence `std::atomic_thread_fence(std::memory_order_acquire)` prevents a read operation before an acquire fence from being reordered with a read or write operation after the acquire fence.

- **Release fence**: A release fence `std::atomic_thread_fence(std::memory_order_release)` prevents a write operation after a release fence from being reorderd with a read or write operation before a release fence.

A lot of energy goes into getting the definitions of the acquire and release fence and their consequences for lock-free programming right. Especially challenging to understand are the subtle differences between the acquire-release semantic of atomic operations. Before I get to that point, I illustrate the definitions with graphics.

Which kind of operations can cross a memory barrier? Have a look at the following three graphics. If the arrow is crossed with a red bar, the fence prevents this type of operation.

2.5.1.2.1 Full fence
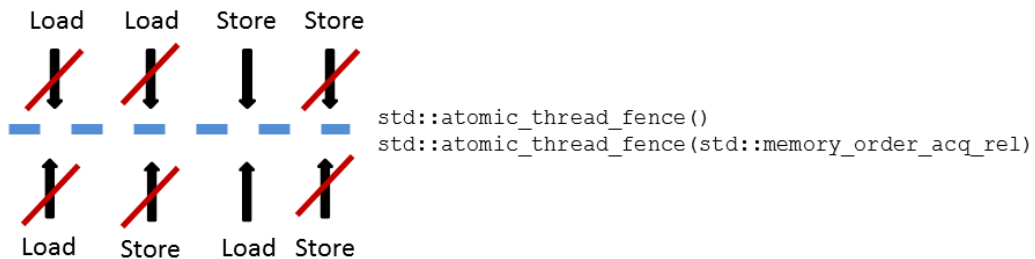
Full fence

Of course, instead of writing `std::atomic_thread_fence()` you can explicitly write `std::atomic_thread_fence(std::memory_order_seq_cst)`. Sequential consistency is applied to fences by default. If you use sequential consistency for a full fence, the `std::atomic_thread_fence` follows a global order.

2.5.1.2.2 Acquire fence
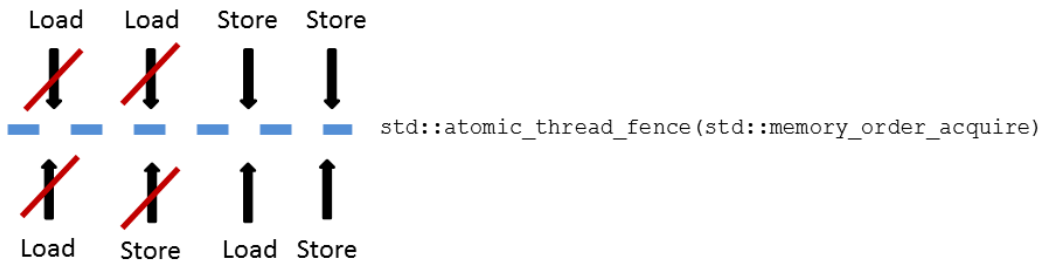
Acquire fence

2.5.1.2.3 Release fence
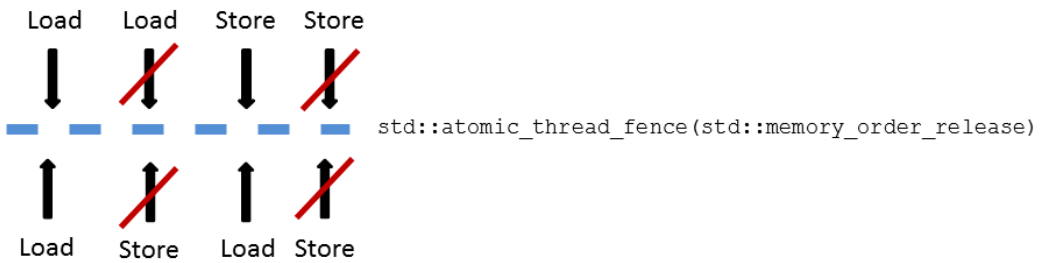
Release fence

The three memory barriers can be depicted even more concisely.

2.5.1.2.4 All Fences at a Glance
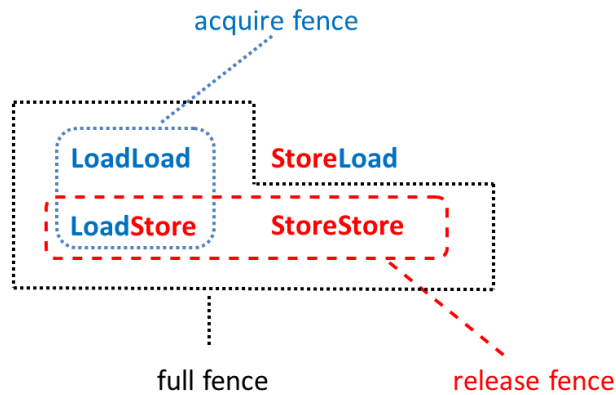
All fences at a glance

Acquire and release fences guarantee similar synchronization and ordering constraints as atomics with acquire-release semantic.

2.5.1.3 Acquire-Release Fences

The most apparent difference between acquire and release fences and atomics with acquire-release semantics is that fences need no atomics. There is also a more subtle difference; the acquire and release fences are more heavyweight than the corresponding atomics.

2.5.1.3.1 Atomic Operations versus Fences

For the sake of simplicity, I now refer to acquire operations when I use fences or atomic operations with acquire semantics. The same holds for release operations.

The main idea of an acquire, and a release operation is that it establishes synchronization and ordering constraints between threads. These synchronization and ordering constraints also hold for atomic operations with relaxed semantic or non-atomic operations. Note that acquire and release operations come in pairs. Besides, operations on atomic variables with acquire-release semantic must act on the same atomic variable. Having said that, I now look at these operations in isolation.

Let's start with the acquire operation.

2.5.1.3.2 Acquire Operation

A load (read) operation on an atomic variable with the memory-ordering set to `std::memory_order_-acquire` is an acquire operation.

```
int ready= var.load(std::memory_order_acquire);
// load and store operations
```

Atomic operation with acquire semantic

`std::atomic_thread_fence` with the memory order set to `std::memory_order_acquire` imposes stricter constraints on memory access reordering:

`std::atomic_thread_fence(std::memory_order_acquire)`
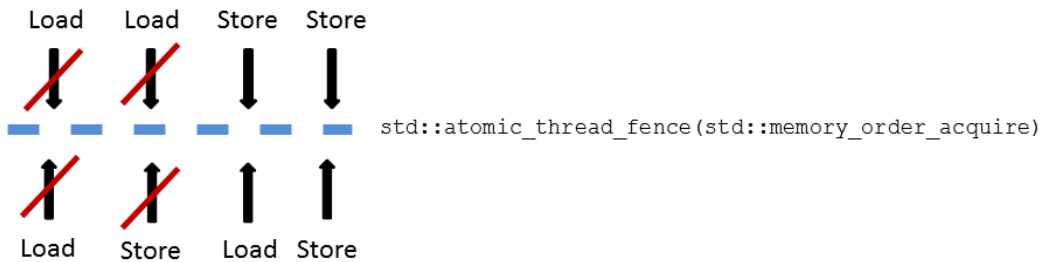
Fence with acquire semantic

This comparison emphasizes two points:

1. A fence with acquire semantic establishes stronger ordering constraints. Although the acquire operation on an atomic and on a fence requires no read or write operation to be moved before the acquire operation, there is an additional guarantee with the acquire fence. No read operation can be moved after the acquire fence.

2. The relaxed semantic is sufficient for the reading of the atomic variable `var`. Thanks to `std::atomc_thread_fence(std::memory_order_acquire)`, this operation cannot be moved after the acquire fence.

Similar observations can be made for the release fence.

2.5.1.3.3 Release Operation

The store (write) operation on an atomic variable with the memory-ordering set to `std::memory_-order_release` is a release operation.

```
// load and store operations
var.store(1,std::memory_order_release);
```

Atomic operation with release semantic

Here is the corresponding image for the release fence.
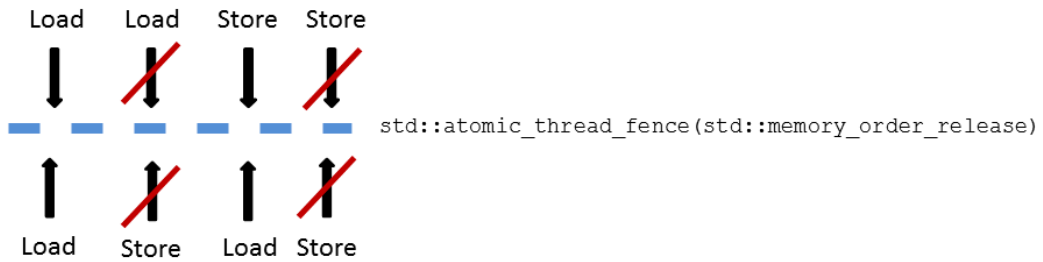
Fence with release semantic

In addition to the constraints imposed by the release operation on an atomic variable var, the release fence guarantees two properties:

1. Store operations can't be moved before the fence.

2. It's sufficient for the variable var to have relaxed semantic.

However, it's time to go one step further and build a program that uses fences.

2.5.1.4 Synchronization with Atomic Variables or Fences

As a starting point, I've implemented a typical consumer-producer workflow with the acquire-release semantic. Initially, I use atomics and then switch to fences.

2.5.1.4.1 Atomic Operations

Let's start with atomics because most of us are comfortable with them.

Acquire-release ordering with atomics

```cpp
// acquireRelease.cpp

#include <atomic>
#include <thread>
#include <iostream>
#include <string>

using namespace std;

atomic<string*> ptr;
int data;
atomic<int> atoData;

void producer(){
    string* p  = new string("C++11");
```

```
16        data = 2011;
17        atoData.store(2014, memory_order_relaxed);
18        ptr.store(p, memory_order_release);
19    }
20
21    void consumer(){
22        string* p2;
23        while (!(p2 = ptr.load(memory_order_acquire)));
24        cout << "*p2: " << *p2 << '\n';
25        cout << "data: " << data << '\n';
26        cout << "atoData: " << atoData.load(memory_order_relaxed) << '\n';
27    }
28
29    int main(){
30
31        cout << '\n';
32
33        thread t1(producer);
34        thread t2(consumer);
35
36        t1.join();
37        t2.join();
38
39        cout << '\n';
40
41    }
```

This program should be quite familiar to you. It is the classic example that I used in the subsection about `std::memory_order_consume`. The graphics emphasize exactly that the consumer thread t2 sees all values from the producer thread t1.

```
void producer(){
  std::string* p  = new std::string("C++11");
  data = 2011;
  atoData.store(2014,std::memory_order_relaxed);
  ptr.store(p, std::memory_order_release);
}

void consumer(){
  std::string* p2;
  while (!(p2 = ptr.load(std::memory_order_acquire)));
  std::cout << "*p2: " << *p2 << std::endl;
  std::cout << "data: " << data << std::endl;
  std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
}
```

happens-before
synchronizes-with

Acquire-release semantic with atomic operations

The program is well-defined because the happens-before relation is transitive. I only have to combine the three happens-before relations:

1. Lines 15 - 17 *happens-before* line 18 `ptr.store(p, std::memory_order_release)`.

2. Line 23 `while(!(p2= ptr1.load(std::memory_order_acquire)))` *happens-before* the lines 24 - 26.

3. Line 18 *synchronizes-with* line 23. ⇒ Line 18 *inter-thread happens-before* line 23.

However, the story becomes more interesting. Now I come to fences. They are almost completely ignored in the literature on the C++ memory model.

2.5.1.4.2 Fences

It's quite straightforward to port the program to use fences.

Acquire-release ordering with fences

```
1   // acquireReleaseFences.cpp
2
3   #include <atomic>
4   #include <thread>
5   #include <iostream>
6   #include <string>
7
8   using namespace std;
9
10  atomic<string*> ptr;
```

```
11    int data;
12    atomic<int> atoData;
13
14    void producer(){
15        string* p  = new string("C++11");
16        data = 2011;
17        atoData.store(2014, memory_order_relaxed);
18        atomic_thread_fence(memory_order_release);
19        ptr.store(p, memory_order_relaxed);
20    }
21
22    void consumer(){
23        string* p2;
24        while (!(p2 = ptr.load(memory_order_relaxed)));
25        atomic_thread_fence(memory_order_acquire);
26        cout << "*p2: " << *p2 << '\n';
27        cout << "data: " << data << '\n';
28        cout << "atoData: " << atoData.load(memory_order_relaxed) << '\n';
29    }
30
31    int main(){
32
33        cout << '\n';
34
35        thread t1(producer);
36        thread t2(consumer);
37
38        t1.join();
39        t2.join();
40
41        delete ptr;
42
43        cout << '\n';
44
45    }
```

The first step was to add fences with release and acquire semantic (lines 18 and 25). Next, I changed the atomic operations with acquire or release semantic to relaxed semantic (lines 19 and 24). That was straightforward. Of course, I can only replace an acquire or release operation with the corresponding fence. The critical point is that the release fence establishes a *synchronizes-with* relation with the acquire fence, and therefore an *inter-thread happens-before* relation.

Here is the output of the program.

Synchronization with fences

For the more visual reader, here's the entire relation graphically.

```cpp
void producer(){
  std::string* p  = new std::string("C++11");
  data = 2011;
  atoData.store(2014,std::memory_order_relaxed);
  std::atomic_thread_fence(std::memory_order_release);
  ptr.store(p, std::memory_order_relaxed);
}

void consumer(){
  std::string* p2;
  while (!(p2 = ptr.load(std::memory_order_relaxed)));
  std::atomic_thread_fence(std::memory_order_acquire);
  std::cout << "*p2: " << *p2 << std::endl;
  std::cout << "data: " << data << std::endl;
  std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
}
```

happens-before
synchronizes-with

Acquire-release semantic with fences

The key question is: why do the operations after the acquire fence see the effects of the operations before the release fence? This guarantee is interesting because `data` is a non-atomic variable, and `atoData.store` is used with relaxed semantic. This would suggest they can be reordered; however, thanks to the `std::atomic_thread_fence(std::memory_order_release)` as a release operation in combination with the `std::atomic_thread_fence(std::memory_order_acquire)`, neither can be reordered.

For clarity, the whole reasoning in a more concise form.

1. The acquire and release fences prevent the reordering of the atomic and non-atomic operations across the fences.

2. The consumer thread `t2` is waiting in the `while (!(p2= ptr.load(std::memory_order_-relaxed)))` loop, until the pointer `ptr.store(p,std::memory_order_relaxed)` is set in the

producer thread `t1`.

3. The release fence *synchronizes-with* the acquire fence.

4. In the end, all effects of relaxed or non-atomic operations that *happen-before* the release fence are visible after the acquire fence.

🔑 Synchronization between the release fence and the acquire fence

The words from the N4659: Working Draft, Standard for Programming Language C++[20] are quite difficult to get: "A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation."

Let me explain the last sentence with the help of the program `acquireReleaseFence.cpp`

- `atomic_thread_fence(memory_order_release)` (line 18) is the release fence A
- `atomic_thread_fence(memory_order_acquire)` (line 25) is the acquire fence B
- `ptr` (line 10) is the atomic object M
- `ptr.store(p, memory_order_relaxed)` (line 19) is the atomic store X
- `while (!(p2 = ptr.load(memory_order_relaxed)))` (line 24) is the atomic load Y

You can even mix the acquire and release operations on an atomic in the program `acquireRelease.cpp` with the acquire and release fence in the program `acquireReleaseFence.cpp` without affecting the *synchronize-with* relation.

2.5.2 `std::atomic_signal_fence`

`std::atomic_signal_fence` establishes memory synchronization ordering of non-atomic and relaxed atomic accesses between a thread and a signal handler executed on the same thread. The following program shows the usage of a `std::atomic_signal_fence`.

---

[20]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf

Synchronization with a signal handler

```
1   // atomicSignal.cpp
2
3   #include <atomic>
4   #include <cassert>
5   #include <csignal>
6
7   std::atomic<bool> a{false};
8   std::atomic<bool> b{false};
9
10  extern "C" void handler(int) {
11      if (a.load(std::memory_order_relaxed)) {
12          std::atomic_signal_fence(std::memory_order_acquire);
13          assert(b.load(std::memory_order_relaxed));
14      }
15  }
16
17  int main() {
18
19      std::signal(SIGTERM, handler);
20
21      b.store(true, std::memory_order_relaxed);
22      std::atomic_signal_fence(std::memory_order_release);
23      a.store(true, std::memory_order_relaxed);
24
25  }
```

First, I set in line 19 the signal handler `handler` for the particular signal SIGTERM. SIGTERM is a termination request for the program. Both `std::atomic_signal_handler` establish an acquire-release fence between the release operation `std::atomic_signal_fence(std::memory_order_release)` (line 22) and the acquire operation `std::atomic_signal_fence(std::memory_order_acquire)` (line 12). This means in particular that release operations can not be reordered across the release fence (line 22) and that acquire operations can not be reordered across the acquire fence (line 11). Consequently, the assertion in line 13 `assert(b.load(std::memory_order_relaxed))` never fires because if `a.store(true, std::memory_order_relaxed)` (line 23) happened, `b.store(true, std::memory_order_relaxed)` (line 21) must have happened before.

Distilled Information

- The memory model is a contract between the programmer and the system. The system consists of the compiler that generates machine code, the processor that executes the machine code and includes the different caches that store the program's state.
- The C++ memory model has to deal with the following questions:
  - Which operations are atomic?
  - Which ordering of operations is guaranteed?
  - What are the visible effects on shared variables to other threads?
- The foundation of the contract are operations on atomics that have two characteristics: They are by definition atomic, and they create synchronization and order constraints on the program execution.
- C++20 support atomics for integrals, floating-points, pointers, and smart pointers.
- The class template `std::atomic_ref` applies atomic operations to the referenced object.
- Roughly speakting, there are three different types of synchronization and ordering constraints in C++:
  - Sequential consistency (default)
  - Acquire-release semantic
  - Relaxed semantic

3. Multithreading

Cippi ties a braid

C++ has a multithreading interface since C++11 and this interface has all the basic building blocks for creating multithreaded programs. These are threads, synchronization primitives for shared data such as mutexes and locks, thread-local data, synchronization mechanism for threads such as condition variables, and tasks. Tasks usually called promises and futures, provide a higher abstraction than native threads.

C++11   **C++14**   **C++17**   **C++20**   **C++23**

| 2011 | 2014 | 2017 | 2020 | 2023 |

- Memory model
- Atomics

- Reader-writer locks

- Parallel STL

- Atomic smart pointers

  - std::jthread
  - Latches and barriers
  - Semaphores

  - Coroutines

- Executors
- `std::future` extensions
- Transactional memory
- Task blocks
- Data-parallel vector library

- Threads
- Mutexes and locks
- Thread local data
- Condition variables
- Tasks
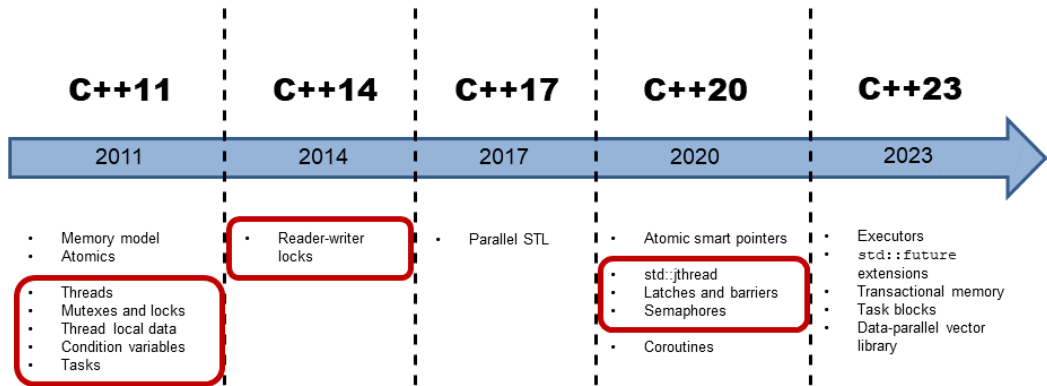
Multithreading in C++

C++ supports two kind of threads: the basic thread `std::thread` (C++11) and the improved thread `std::jthread` (C++20). First, I write about the basic thread, and at the end of this section, I discuss the improved thread.

3.1 The Basic Thread `std::thread`

To launch a thread in C++, you have to include the `<thread>` header.

3.1.1 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 thread is not copy-constructible or copy-assignable but move-constructible or move-assignable.

A callable unit is an entity that behaves like a function. Of course, it can be a function but also a function object, or a lambda function. The return value of the callable unit is ignored.

After discussing theory, here is a small example.

Creation of a thread with callable units

```cpp
// createThread.cpp

#include <iostream>
#include <thread>

void helloFunction(){
  std::cout << "Hello from a function." << '\n';
}

class HelloFunctionObject{
  public:
    void operator()() const {
      std::cout << "Hello from a function object." << '\n';
    }
};

int main(){

  std::cout << '\n';

  std::thread t1(helloFunction);

  HelloFunctionObject helloFunctionObject;
  std::thread t2(helloFunctionObject);
```
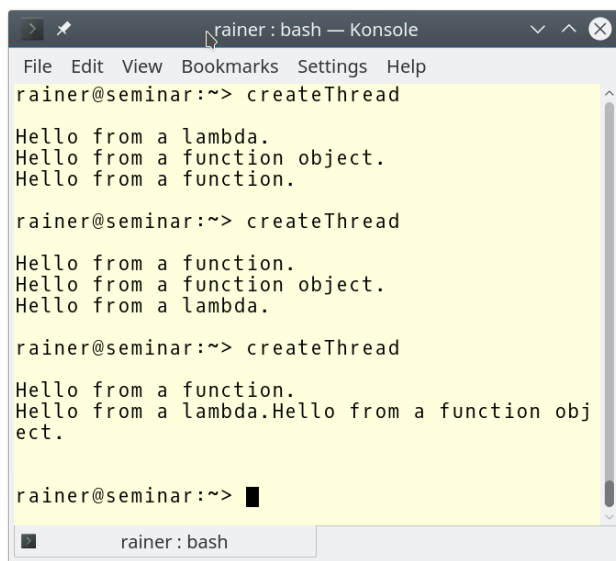
```
25
26    std::thread t3([]{std::cout << "Hello from a lambda." << '\n';});
27
28    t1.join();
29    t2.join();
30    t3.join();
31
32    std::cout << '\n';
33
34  };
```

All three threads (t1, t2, and t3) write their messages to the console. The work package of thread t2 is a function object (lines 10 - 15), the work package of thread t3 is a lambda function (line 26). In lines 28 - 30, the main thread is waiting until its children are done.

Let's have a look at the output. This is more interesting.

Creation of threads with various callables

The three threads are executed in an arbitrary order. Even the three output operations can interleave.

In our case, the child's creator is the main thread, is responsible for the child's lifetime.

3.1.2 Thread Lifetime

The parent has to take care of its children. This simple principle has big consequences for the lifetime of a thread. This small program starts a thread that displays its id.

Forget to join a thread

```
1   // threadWithoutJoin.cpp
2
3   #include <iostream>
4   #include <thread>
5
6   int main(){
7
8     std::thread t([]{std::cout << std::this_thread::get_id() << '\n';});
9
10  }
```

But the program does not print the id.
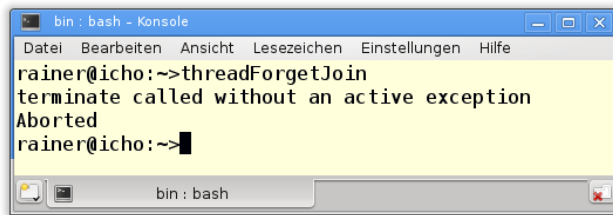
Forget to join a thread

What's the reason for the exception?

3.1.2.1 `join` **and** `detach`

The lifetime of a created thread `t` ends with its callable unit. The creator has two choices.

1. It can wait until its child is done: `t.join()`.

2. It can detach itself from its child: `t.detach()`.

A `t.join()` call is useful when the following code relies on the result of the calculation performed in the thread. `t.detach()` permits the thread to execute independently from the thread handle `t`; therefore, the detached thread runs for the lifetime of the executable. Typically you use a detached thread for a long-running background service such as a server.
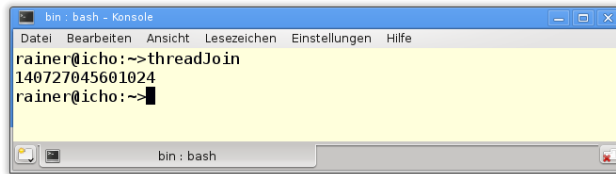
A thread `t` with a callable unit - you can create threads without a callable unit - is called joinable if neither a `t.join()` nor a `t.detach()` call happened. The destructor of a joinable thread throws an exception and `std::terminate` is called. This was the reason the program execution of `threadWithoutJoin.cpp` terminated with an exception. If you invoke `t.join()` or `t.detach()` more than once on a thread `t`, you get a `std::system_error` exception.

The solution to this problem is quite simple: call `t.join()`.

Joining a thread

```cpp
1   // threadWithJoin.cpp
2
3   #include <iostream>
4   #include <thread>
5
6   int main(){
7
8     std::thread t([]{std::cout << std::this_thread::get_id() << '\n';});
9
10    t.join();
11
12  }
```

Now we get the expected output.

Join a thread

The thread's id serves as a unique identifier of the std::thread.

⚠ The Challenge of detach

Of course, you can use `t.detach()` instead of `t.join()` in the last program. The thread `t` is not joinable anymore; therefore, its destructor didn't call `std::terminate`. But now you have another issue. The program behavior is undefined because the main program may complete before the thread `t` has time to complete its work package; therefore, its lifetime is too short to display the id. For more details see lifetime issues of variables.

scoped_thread **by Anthony Williams**

If it's too bothersome for you to take care of the lifetime of your threads t manually, you can encapsulate a std::thread in your wrapper class. This class should automatically call t.join() in its destructor if the thread is still joinable. Of course, you can go the other way and call t.detach(); but you know, there is an issue with detaching.

Anthony Williams created such a useful class and presented it in his excellent book C++ Concurrency in Action[1]. He called the wrapper scoped_thread. scoped_thread gets a thread t in its constructor and checks if t is still joinable. If the thread t passed into the constructor is not joinable anymore, there is no need for the scoped_thread. If t is joinable, the destructor calls t.join(). Because the copy constructor and copy assignment operator are declared as delete, instances of scoped_thread cannot be copied to or assigned from.

```cpp
// scoped_thread.cpp

#include <thread>
#include <utility>


class scoped_thread{
std::thread t;
public:
 explicit scoped_thread(std::thread t_): t(std::move(t_)){
   if (!t.joinable()) throw std::logic_error("No thread");
 }
 ~scoped_thread(){
   t.join();
 }
 scoped_thread(scoped_thread&)= delete;
 scoped_thread& operator=(scoped_thread const &)= delete;
};
```

3.1.3 Thread Arguments

A thread such as an arbitrary function can get its arguments by copy, by move, or by reference. std::thread is a variadic template[2]. This means it can get an arbitrary number of arguments.

If your thread gets its data by reference, you have to be extremely careful about the lifetime of the arguments and the sharing of data.

---

[1]https://www.manning.com/books/c-plus-plus-concurrency-in-action
[2]http://en.cppreference.com/w/cpp/language/parameter_pack

3.1.3.1 Copy or Reference

Let's have a look at a small code snippet.

```
std::string s{"C++11"}

std::thread t1([=]{ std::cout << s << '\n'; });
t1.join();

std::thread t2([&]{ std::cout << s << '\n'; });
t2.detach();
```

Thread t1 gets its argument by copy, thread t2 by reference.

Thread arguments by reference

To be honest, I cheated a little. The thread t2 gets its argument by reference, but the lambda function captures its argument by reference. If you need to pass the argument to a thread by reference, it must be wrapped in a reference wrapper[3]. This is quite straightforward with the helper function std::ref[4]. std::ref is defined in the header <functional>.

```
<functional>
   ...
void transferMoney(int amount, Account& from, Account& to){
   ...
}
...
std::thread thr1(transferMoney, 50, std::ref(account1), std::ref(account2));
```

Thread thr1 executes the function transferMoney. transferMoney gets its arguments by reference; therefore, thread thr1 gets its account1 and account2 by reference.

What issues are hiding in these lines of code? Thread t2 gets its string s by reference, and afterward is detached from its creator's lifetime. The lifetime of the string s is bound to the creator's lifetime; the lifetime of the global object std::cout is attached to the main thread's lifetime. So the lifetime of s or the lifetime of std::cout may be shorter than the lifetime of the thread t2. Now we are deep in the area of undefined behavior.

Are you not convinced? Let's take a closer look at what undefined behavior may look like.

---

[3]http://en.cppreference.com/w/cpp/utility/functional/reference_wrapper
[4]http://en.cppreference.com/w/cpp/utility/functional/ref

Passing the arguments to a thread by reference

```cpp
// threadArguments.cpp

#include <chrono>
#include <iostream>
#include <thread>

class Sleeper{
  public:
    Sleeper(int& i_): i{i_}{};
    void operator() (int k){
      for (unsigned int j = 0; j <= 5; ++j){
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        i += k;
      }
      std::cout << std::this_thread::get_id() << '\n';
    }
  private:
    int& i;
};


int main(){

  std::cout << '\n';

  int valSleeper = 1000;
  std::thread t(Sleeper(valSleeper), 5);
  t.detach();
  std::cout << "valSleeper = " << valSleeper << '\n';

  std::cout << '\n';

}
```

What value does valSleeper have in line 29? valSleeper is local to the main() function. Thread t gets as its work package a function object with the variable valSleeper and the number 5 (line 27). The crucial observation is that the thread gets valSleeper by reference (line 9) and is detached from the main thread (line 28). Next, it executes the call operator of the function object (lines 10 - 16). In this member function, it counts from 0 to 5, sleeps in each iteration 1/10 of a second, and increments i by k. In the end, it displays its id on the screen. Nach Adam Riese[5] (a German proverb), the result should be 1000 + 6 * 5 = 1030.

---

[5]https://de.wikipedia.org/wiki/Liste_gefl%C3%BCgelter_Worte/N#Nach_Adam_Riese

However, what happened? Something is going very wrong.
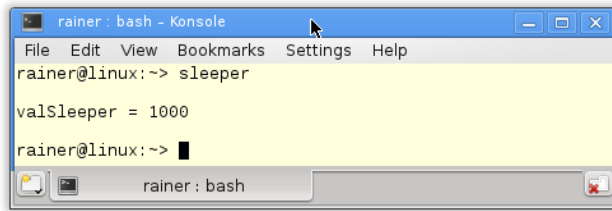
Undefined behavior with a reference

The program has two strange properties. First, `valSleeper` is 1000, and second, the id is not displayed.

The program has at least two issues.

1. `valSleeper` is shared by all threads. This is a data race because the threads may read and write `valSleeper` at the same time.
2. The main thread's lifetime ends before the child thread has performed its calculation or written its id to `std::cout`.

Both issues are race conditions because the program's result depends on the interleaving of the operations. The race condition is the cause of the data race.

Fixing the data race is pretty easy. `valSleeper` should be protected using either a lock or an atomic. To overcome the lifetime issues of `valSleeper` and `std::cout`, you have to join the thread instead of detaching it.

Here is the modified main function.
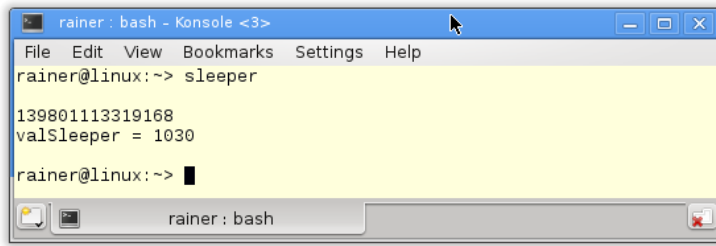
```cpp
int main(){

  std::cout << '\n';

  int valSleeper= 1000;
  std::thread t(Sleeper(valSleeper),5);
  t.join();
  std::cout << "valSleeper = " << valSleeper << '\n';

  std::cout << '\n';

}
```

Now we get the right result. Of course, the execution becomes slower.

Fixed the undefined behavior

To complete the story of `std::thread`, here are the remaining member functions.

3.1.4 Member Functions

Here is the interface of `std::thread  t` in a concise table. For additional details, please refer to cppreference.com[6].

Member functions of a thread `t`

Member functions	Description
`t.join()`	Waits until thread `t` has finished its execution.
`t.detach()`	Executes the created thread `t` independently of the creator.
`t.joinable()`	Returns true if thread `t` is still joinable.
`t.get_id()` and `std::this_thread::get_id()`	Returns the id of the thread.
`std::thread::hardware_concurrency()`	Indicates the number of threads that can run concurrently.
`std::this_thread::sleep_until(absTime)`	Puts thread `t` to sleep until the time point `absTime`.
`std::this_thread::sleep_for(relTime)`	Puts thread `t` to sleep for the time duration `relTime`.
`std::this_thread::yield()`	Enables the system to run another thread.
`t.swap(t2)`	Swaps the threads. Same as `std::swap(t, t2)`.

The static member function `std::thread::hardware_concurrency` returns the number of concurrent threads supported by the implementation, or 0 if the runtime can not determine the number. This is according to the C++ standard. The `sleep_until` and `sleep_for` operations need a time point or a time

---

[6]http://de.cppreference.com/w/cpp/thread/thread

duration as an argument.

🔑 Access to the system-specific implementation

The threading interface is a wrapper around the underlying implementation. You can use the member function `native_handle` to get access to the system-specific implementation. This handle to the underlying implementation is available for threads, mutexes, and condition variables.

Threads cannot be copied but can be moved. The swap member function performs a move when possible.

To conclude this subsection, here are a few of the mentioned member functions in practice.

Member functions of a thread
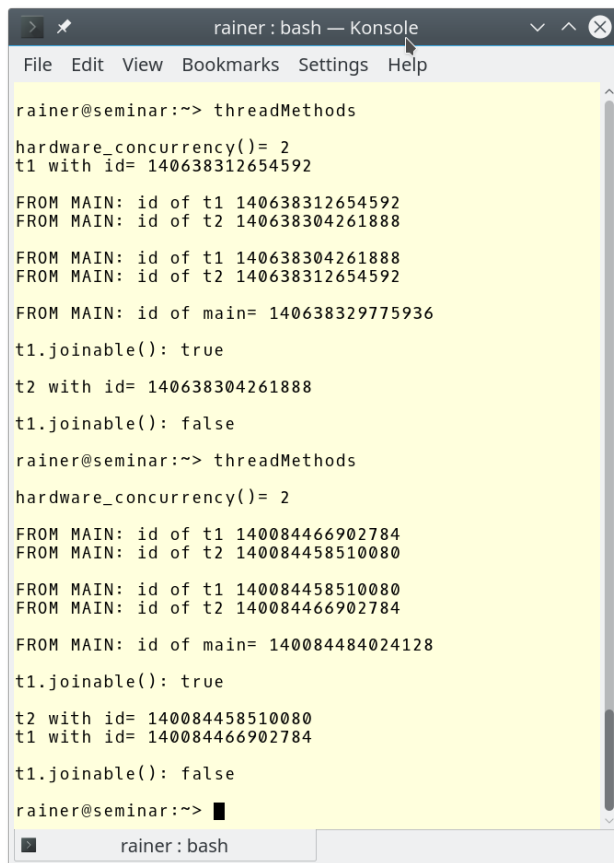
```cpp
// threadMethods.cpp

#include <iostream>
#include <thread>

using namespace std;

int main(){

  cout << boolalpha << '\n';

  cout << "hardware_concurrency()= "<< thread::hardware_concurrency() << '\n';

  thread t1([]{cout << "t1 with id= " << this_thread::get_id() << '\n';});
  thread t2([]{cout << "t2 with id= " << this_thread::get_id() << '\n';});

  cout << '\n';

  cout << "FROM MAIN: id of t1 " << t1.get_id() << '\n';
  cout << "FROM MAIN: id of t2 " << t2.get_id() << '\n';

  cout << '\n';
  swap(t1,t2);

  cout << "FROM MAIN: id of t1 " << t1.get_id() << '\n';
  cout << "FROM MAIN: id of t2 " << t2.get_id() << '\n';

  cout << '\n';
```

```
30      cout << "FROM MAIN: id of main= " << this_thread::get_id() << '\n';
31
32      cout << '\n';
33
34      cout << "t1.joinable(): " << t1.joinable() << '\n';
35
36      cout << '\n';
37
38      t1.join();
39      t2.join();
40
41      cout << '\n';
42
43      cout << "t1.joinable(): " << t1.joinable() << '\n';
44
45      cout << '\n';
46
47   }
```

In combination with the output, the program should be relatively easy to follow.

Member functions of a thread

Maybe it looks a little weird that threads `t1` and `t2` (lines 14 and 15) run at different points in time of the program execution. You have no guarantee when each thread runs; you only have the assurance that both threads run before `t1.join()` and `t2.join()` in lines 38 and 39.

The more mutable (non-const) variables threads share, the more challenging multithreading becomes.

3.2 The Improved Thread `std::jthread` (C++20)

`std::jthread` stands for joining thread. In addition to `std::thread` from C++11, `std::jthread` automatically joins in it destructor and can cooperatively be interrupted. Consequently, `std::jthread` extends the interface of `std::thread`. `std::jthread` models RAII and, therefore, also joins when an exception occurs. The `std::jthread` holds a private member of type `std::stop_source`. `std::jthread` thread constructor accept a function that takes `std::stop_token` as its first argument. This `std::stop_token`

is passed into the function and can, therefore, be used by the function to check if a stop request has be requested.

The following table gives you a concise overview of the additional `std::jthread t` functionality.

<div align="center">

Additional member functions of a `std::jthread t`

Member functions	Description
`t.get_stop_source()`	Returns a `std::stop_source` object associated with the shared stop state.
`t.get_stop_token()`	Returns a `std::stop_token` object associated with the shared stop state.
`t.request_stop()`	Requests execution stop via the shared stop state. Returns `true` if the stop request was successful.

</div>

3.2.1 Automatically Joining

This is the *non-intuitive* behavior of `std::thread`. If a `std::thread` is still joinable, `std::terminate` is called in its destructor. A thread `thr` is joinable if either `thr.join()` nor `thr.detach()` was called.

Terminating a still joinable `std::thread`

```cpp
// threadJoinable.cpp

#include <iostream>
#include <thread>

int main() {

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

    std::thread thr{[]{ std::cout << "Joinable std::thread" << '\n'; }};

    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';

}
```
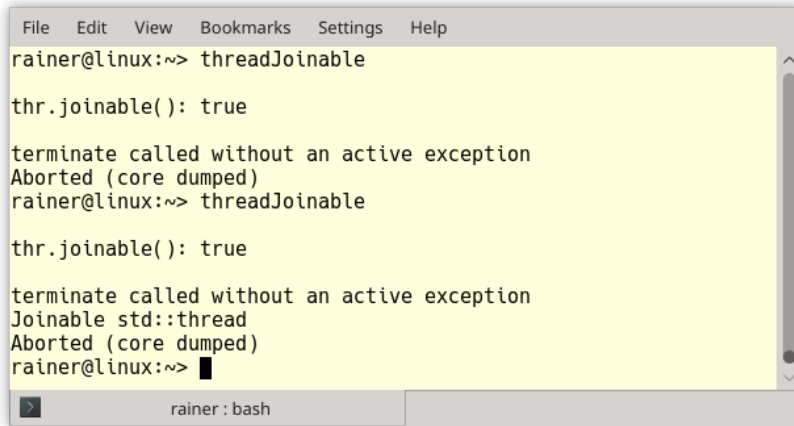
When executed, the program terminates.

Terminating a joinable `std::thread`

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

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

Terminating a still joinable `std::jthread`

```cpp
// jthreadJoinable.cpp

#include <iostream>
#include <thread>

int main() {

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

    std::jthread thr{[]{ std::cout << "Joinable std::thread" << '\n'; }};

    std::cout << "thr.joinable(): " << thr.joinable() << '\n';

    std::cout << '\n';


}
```
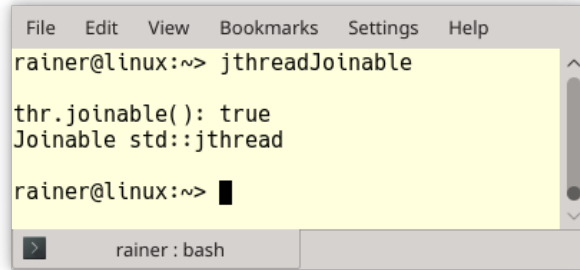
Now, the thread `thr` automatically joins in its destructor such if it's still joinable.

Using a `std::jthread` that joins automatically

Here is a typical implementation of `std::jthreads` destructor.

Typical implemenation of `std::jthreads` destructor

```
1  jthread::~jthread() {
2      if(joinable()) {
3          request_stop();
4          join();
5      }
6  }
```

First, the thread checks if it is still joinable (line 2). A thread is still joinable if neither `join()` or `detach()` was called on it. If the thread is still joinable, it asks for the stopping of the execution (line 3) and calls `join()` afterward (line 4). The join call blocks until the thread's execution is done.

3.2.2 Cooperative Interruption of a `std::jthread`

To get the general idea, let me present a simple example.

Interrupt an non-interruptable and interruptable `std::jthread`

```cpp
1   // interruptJthread.cpp
2
3   #include <chrono>
4   #include <iostream>
5   #include <thread>
6
7   using namespace::std::literals;
8
9   int main() {
10
11      std::cout << '\n';
12
```
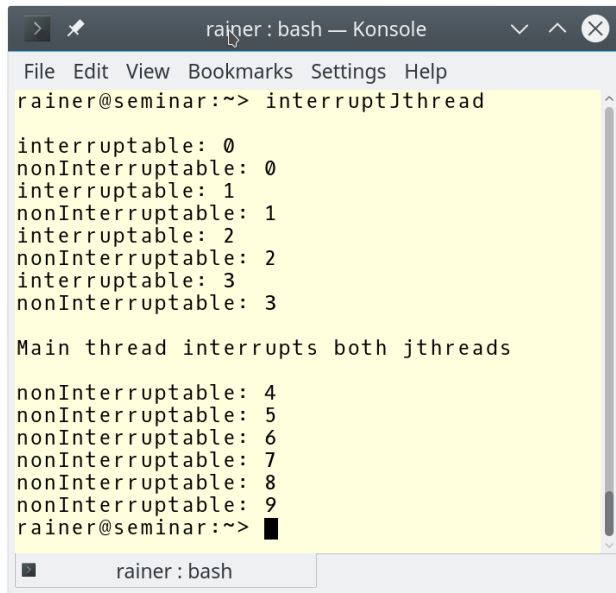
```
13        std::jthread nonInterruptable([]{
14            int counter{0};
15            while (counter < 10){
16                std::this_thread::sleep_for(0.2s);
17                std::cerr << "nonInterruptable: " << counter << '\n';
18                ++counter;
19            }
20        });
21
22        std::jthread interruptable([](std::stop_token stoken){
23            int counter{0};
24            while (counter < 10){
25                std::this_thread::sleep_for(0.2s);
26                if (stoken.stop_requested()) return;
27                std::cerr << "interruptable: " << counter << '\n';
28                ++counter;
29            }
30        });
31
32        std::this_thread::sleep_for(1s);
33
34        std::cerr << '\n';
35        std::cerr << "Main thread interrupts both jthreads" << '\n';
36        nonInterruptable.request_stop();
37        interruptable.request_stop();
38
39        std::cout << '\n';
40
41    }
```

In the main program, I start the two threads nonInterruptible and interruptible (lines 13 and 22). Unlike in the thread nonInterruptible, the thread interruptible gets a std::stop_token and uses it in line 26 to check if it was interrupted: stoken.stop_requested(). In case of a stop request, the lambda function returns and, therefore, the thread ends. The call interruptible.request_stop() (line 37) triggers the stop request. This does not hold for the previous call nonInterruptible.request_stop(). The call has no effect.

Interrupt a non-interruptable and interruptable `std::jthread`"

I provide the details about the `std::stop_token` in the section about [cooperative interruption.](#)

3.3 Shared Data

To make the point clear, you only need to think about synchronization if you have shared, mutable data because shared, mutable data is prone to data races. If you have concurrent non-synchronized read and write access to data, your program has undefined behavior.

The easiest way to visualize concurrent, unsynchronized read and write operations is to write something to std::cout.

Let's have a look.

Writing unsynchronized to `std::cout`

```cpp
// coutUnsynchronized.cpp

#include <chrono>
#include <iostream>
#include <thread>

class Worker{
public:
  Worker(std::string n):name(n){};
    void operator() (){
      for (int i = 1; i <= 3; ++i){
        // begin work
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        // end work
        std::cout << name << ": " << "Work " << i << " done !!!" << '\n';
      }
    }
private:
  std::string name;
};


int main(){

  std::cout << '\n';

  std::cout << "Boss: Let's start working.\n\n";

  std::thread herb= std::thread(Worker("Herb"));
  std::thread andrei= std::thread(Worker("  Andrei"));
  std::thread scott= std::thread(Worker("    Scott"));
  std::thread bjarne= std::thread(Worker("      Bjarne"));
  std::thread bart= std::thread(Worker("        Bart"));
```

```
34    std::thread jenne= std::thread(Worker("            Jenne"));
35
36
37    herb.join();
38    andrei.join();
39    scott.join();
40    bjarne.join();
41    bart.join();
42    jenne.join();
43
44    std::cout << "\n" << "Boss: Let's go home." << '\n';
45
46    std::cout << '\n';
47
48  }
```

The program describes a workflow. The boss has six workers (lines 29 - 34). Each worker has to take care of 3 work packages. The work package takes 1/5 second (line 13). After the worker is done with his work package, he screams out loudly to the boss (line 15). Once the boss received notifications from all workers, he sends them home (line 44).

What a mess for such a simple workflow!

Non-synchronized writing to `std::cout`

The most straightforward solution is to use a mutex.

3.3.1 Mutexes

Mutex stands for **mut**ual **ex**clusion. It ensures that only one thread can access a critical section at any one time.

By using a mutex, the mess of the workflow turns into harmony.

Writing synchronized to `std::cout`

```cpp
1   // coutSynchronized.cpp
2
3   #include <chrono>
4   #include <iostream>
5   #include <mutex>
6   #include <thread>
7
8   std::mutex coutMutex;
9
10  class Worker{
```
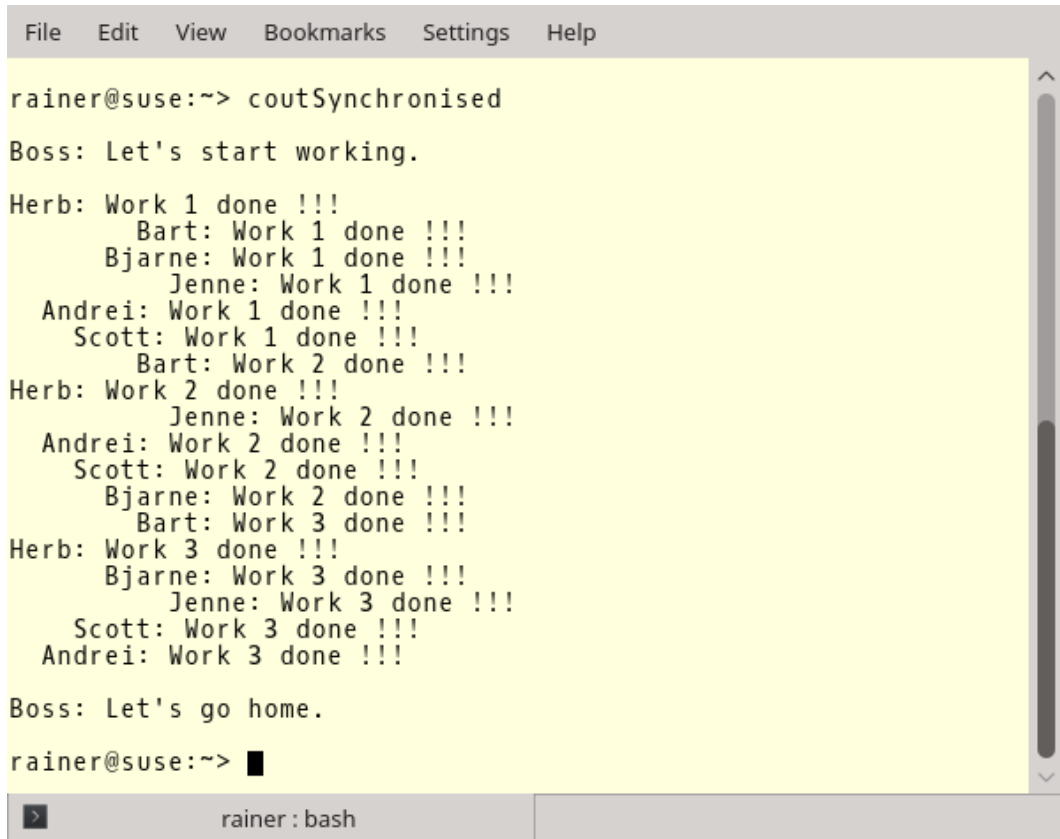
```cpp
11    public:
12      Worker(std::string n):name(n){};
13
14        void operator() (){
15          for (int i = 1; i <= 3; ++i){
16            // begin work
17            std::this_thread::sleep_for(std::chrono::milliseconds(200));
18            // end work
19            coutMutex.lock();
20            std::cout << name << ": " << "Work " << i << " done !!!" << '\n';
21            coutMutex.unlock();
22          }
23        }
24    private:
25      std::string name;
26    };
27
28
29    int main(){
30
31      std::cout << '\n';
32
33      std::cout << "Boss: Let's start working." << "\n\n";
34
35      std::thread herb= std::thread(Worker("Herb"));
36      std::thread andrei= std::thread(Worker("  Andrei"));
37      std::thread scott= std::thread(Worker("    Scott"));
38      std::thread bjarne= std::thread(Worker("      Bjarne"));
39      std::thread bart= std::thread(Worker("        Bart"));
40      std::thread jenne= std::thread(Worker("          Jenne"));
41
42      herb.join();
43      andrei.join();
44      scott.join();
45      bjarne.join();
46      bart.join();
47      jenne.join();
48
49      std::cout << "\n" << "Boss: Let's go home." << '\n';
50
51      std::cout << '\n';
52
53    }
```

std::cout is protected by the coutMutex in line 8. A simple lock() in line 19 and the corresponding unlock() call in line 21 ensure that the workers won't scream all at once.

```
File    Edit    View    Bookmarks    Settings    Help

rainer@suse:~> coutSynchronised

Boss: Let's start working.

Herb: Work 1 done !!!
        Bart: Work 1 done !!!
        Bjarne: Work 1 done !!!
            Jenne: Work 1 done !!!
  Andrei: Work 1 done !!!
    Scott: Work 1 done !!!
        Bart: Work 2 done !!!
Herb: Work 2 done !!!
            Jenne: Work 2 done !!!
  Andrei: Work 2 done !!!
    Scott: Work 2 done !!!
        Bjarne: Work 2 done !!!
        Bart: Work 3 done !!!
Herb: Work 3 done !!!
        Bjarne: Work 3 done !!!
            Jenne: Work 3 done !!!
    Scott: Work 3 done !!!
  Andrei: Work 3 done !!!

Boss: Let's go home.

rainer@suse:~> █

                     rainer : bash
```

Synchronized writing to std::cout

🔑 std::cout **is thread-safe**

The C++11 standard guarantees that you must not protect std::cout. Each character is written atomically. More output statements like those in the example may interleave. This is only a visual issue; the program is well-defined. This remark is valid for all global stream objects. Insertion to and extraction from global stream objects (std::cout, std::cin, std::cerr, and std::clog) is thread-safe.

To put it more formally: writing to std::cout is not a data race but a race condition. This means that the output depends on the interleaving of threads.

C++11 has four different exclusive mutexes that can lock recursively, tentative with and without time constraints.

Exclusive mutex variations

Member function	mutex	recursive_mutex	timed_mutex	recursive_timed_mutex
`m.lock`	yes	yes	yes	yes
`m.try_lock`	yes	yes	yes	yes
`m.try_lock_for`			yes	yes
`m.try_lock_until`			yes	yes
`m.unlock`	yes	yes	yes	yes

A recursive mutex allows the same thread to lock the mutex many times. The mutex stays locked until it was unlocked as many times as it was locked. The maximum number of times that a recursive mutex can be locked is unspecified. If the maximum number is reached, an std::system_error[7] exception is thrown.

With C++14 we have a std::shared_timed_mutex and with C++17 a std::shared_mutex. std::shared_mutex and std::shared_timed_mutex are quite similar. You can use both mutexes for exclusive or shared locking. Additionally, with std::shared_timed_mutex you can specify a time point or a time duration.

Shared mutex variations

Member function	shared_timed_mutex	shared_mutex
`m.lock`	yes	yes
`m.try_lock`	yes	yes
`m.try_lock_for`	yes	
`m.try_lock_until`	yes	
`m.unlock`	yes	yes
`m.lock_shared`	yes	yes
`m.try_lock_shared`	yes	yes
`m.try_lock_shared_for`	yes	
`m.try_lock_shared_until`	yes	

---

[7]http://en.cppreference.com/w/cpp/error/system_error

Shared mutex variations

Member function	shared_timed_mutex	shared_mutex
`m.unlock_shared`	yes	yes

The `std::shared_timed_mutex`(`std::shared_mutex`) enables you to implement reader-writer locks. This means you can use `std::shared_timed_mutex` (`std::shared_mutex`) for exclusive or for shared locking. You get an exclusive lock, if you put the `std::shared_timed_mutex` (`std::shared_mutex`) into a `std::lock_guard` or into a `std::unique_lock`; you get a shared lock, if you put the `std::shared_timed_-mutex` (`std::shared_lock`) into a `std::shared_lock`. The member functions `m.try_lock_for(relTime)` and `m.try_lock_shared_for(relTime)`) need a relative time duration; the member functions `m.try_-lock_until(absTime)` and `m.try_lock_shared_until(absTime)` need an absolute time point.

`m.try_lock` (`m.try_lock_shared`) tries to lock the mutex and returns immediately. On success, it returns true, otherwise false. In contrast the member functions `m.try_lock_for` (`m.try_lock_shared_for`) and `m.try_lock_until` (`m.try_lock_shared_until`) try to lock until the specified time out occurs or the lock is acquired, whichever comes first. You should use a steady clock for your time constraint. A steady clock can not be adjusted.

You should not use mutexes directly; you should put mutexes into locks. Here is the reason why.

3.3.1.1 Issues of Mutexes

The issues with mutexes boil down to one main concern: deadlocks.

Deadlock
> A deadlock is a state where two or more threads are blocked because each thread waits for the release of a resource before it releases its resource.

The result of a deadlock is a total standstill. The thread that tries to acquire the resource, and usually the whole program, is blocked forever. Producing a deadlock is easy. Curious?

3.3.1.1.1 Exceptions and Unknown Code

The small code snippet has many issues.

```
std::mutex m;
m.lock();
sharedVariable = getVar();
m.unlock();
```

Here are the issues:

1. If the function `getVar()` throws an exception, the mutex `m` is not released.

2. **Never ever** call an unknown function while holding a lock. If the function `getVar` tries to lock the mutex `m`, the program has undefined behavior because `m` is not a recursive mutex. Most of the time, undefined behavior causes a deadlock.

3. Avoid calling a function while holding a lock. Maybe the function is from a library, and you get a new version of the library, or the function is rewritten. There is always the danger of a deadlock.

The more locks your program needs, the more challenging it becomes. The dependency is very non-linear.

3.3.1.1.2 Lock Mutexes in Different Order

Here is a typical scenario of a deadlock resulting from locking in a different order.
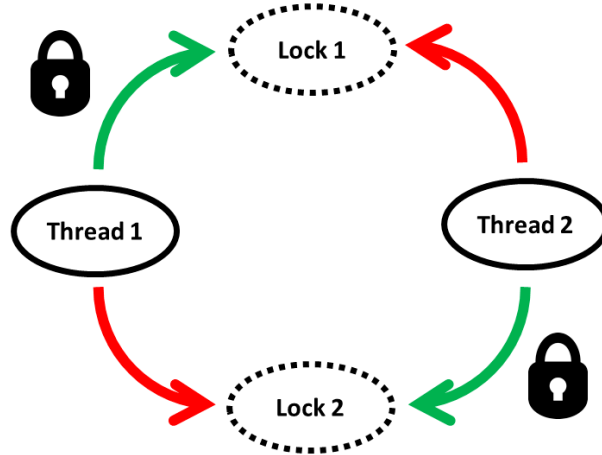
Deadlock of two threads

Thread 1 and thread 2 need access to two resources to finish their work. The problem arises when the requested resources are protected by two separate mutexes and are requested in different orders (Thread 1: Lock 1, Lock 2; Thread 2: Lock 2, Lock 1). In this case, the thread executions interleave so that thread 1 gets mutex 1, then thread 2 gets mutex 2, and we reach a standstill. Each thread wants to get the other's mutex, but the other thread has to release it first to get the other mutex. The expression "deadly embrace" describes this kind of deadlock very well.

Translating this picture into code is easy.

Locking mutexes in different order

```cpp
// deadlock.cpp

#include <iostream>
#include <chrono>
#include <mutex>
#include <thread>

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';
  // do something with a and b
  a.mut.unlock();
  b.mut.unlock();

}

int main(){

  CriticalData c1;
  CriticalData c2;

  std::thread t1([&]{deadLock(c1,c2);});
  std::thread t2([&]{deadLock(c2,c1);});

  t1.join();
  t2.join();

}
```

Threads `t1` and `t2` call `deadlock` (lines 12 - 23). The function `deadlock` needs variables `CriticalData` `c1` and `c2` (lines 27 and 28). Because objects `c1` and `c2` have to be protected from shared access, they internally hold a mutex (to keep this example short and simple, `CriticalData` doesn't have any other member functions or members apart from a mutex).

A short sleep of about one millisecond in line 16 is sufficient to produce the deadlock.
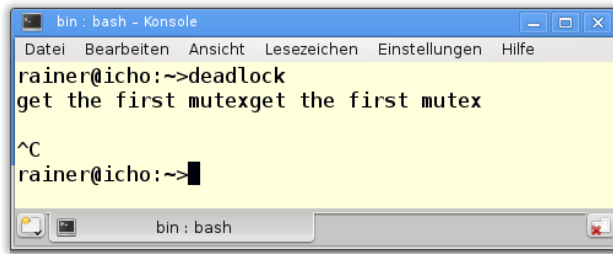
Deadlock of two threads

The only choice left is to press CTRL+C and kill the process.

Thanks to `std::lock`, it is easy to lock many Lockables such as mutexes or locks in an atomic step. Therefore you can overcome deadlocks by locking Lockables in a different order. The section `std::lock` provide more details about atomic locking with `std::lock`.

Locks do not solve all the issues with mutexes, but they come to our rescue in many cases.

3.3.2 Locks

Locks take care of their resource following the RAII idiom. A lock automatically binds its mutex in the constructor and releases it in the destructor. This considerably reduces the risk of a deadlock, because the runtime takes care of the mutex.

Locks are available in four different flavours: `std::lock_guard` for the simple use-cases; `std::unique_-lock` for the advanced use-cases. `std::shared_lock` is available since C++14 and can be used to implement reader-writer locks. With C++17 we got the `std::scoped_lock` which can lock more mutexes in an atomic step.

First, the simple use-case.

3.3.2.1 `std::lock_guard`

```
std::mutex m;
m.lock();
sharedVariable = getVar();
m.unlock();
```

The mutex `m` ensures that access to the critical section `sharedVariable = getVar()` is sequential. Sequential means in this special case that each thread gains access to the critical section after the other. This establishes a total order in the system. The code is simple but prone to deadlocks. A deadlock may appear if the critical section throws an exception or if the programmer forgets to unlock the mutex. With `std::lock_guard` we can do this more elegantly:

```
std::mutex m;
{
    std::lock_guard<std::mutex> lockGuard(m);
    sharedVariable = getVar();
}
```

That was easy, but what's the story with the opening and closing brackets? Its scope limits the lifetime of `std::lock_guard` and the scope is defined by the curly brackets[8]. This means that its lifetime ends when it passes the closing curly brackets. Exactly then, the `std::lock_guard` destructor is called, and - as you may have guessed - the mutex is released. This happens automatically and, it happens also if `getVar()` in `sharedVariable = getVar()` throws an exception. Function scope and loop scope also limit the lifetime of an object.

3.3.2.2 `std::scoped_lock`

With C++17 we got the `std::scoped_lock`. It's very similar to `std::lock_guard`, but `std::scoped_lock` can, additionally, lock an arbitrary number of mutexes atomically. You have to keep a few facts in mind.

1. If `std::scoped_lock` is invoked with one mutex `m` it behaves such as a `std::lock_guard` and locks the mutex m: `m.lock`. If the `std::scoped_lock` is invoked with more than one mutex (`std::scoped_lock(MutexTypes& ... m)`) it uses the function `std::lock(m ...)`.

2. If the current threads already owns one of the mutexes and the mutex is not recursive, the behavior is undefined. With high probability, you get a deadlock.

3. You can just take the ownership of the mutex without locking them. In this case, you have to provide the `std::adopt_lock_t` flag to the constructor: `std::scoped_lock(std::adopt_lock_t, MutexTypes& ... m)`.

You can quite elegantly solve the previous deadlock by using a `std::scoped_lock`. I discuss the resolution of the deadlock in the following section.

3.3.2.3 `std::unique_lock`

A `std::unique_lock` is stronger but more expensive than its little brother `std::lock_guard`.

In addition to what's offered by a `std::lock_guard`, a `std::unique_lock` enables you to

- create it without an associated mutex.
- create it without locking the associated mutex.
- explicitly and repeatedly set or release the lock of the associated mutex.
- recursively lock its mutex.
- move the mutex.

---

[8]http://en.cppreference.com/w/cpp/language/scope#Block_scope

- try to lock the mutex.
- delay the lock on the associated mutex.

The following table shows the member functions of a `std::unique_lock` `lk`.

The interface of `std::unique_lock`

Member function	Description
`lk.lock()`	Locks the associated mutex.
`lk.try_lock()` and `lk.try_lock_for(relTime)` and `lk.try_lock_until(absTime)`	Tries to lock the associated mutex. Tries to lock the associated mutex for the time duration `relTime`. Tries to lock the associated mutex until the time point `absTime`.
`lk.unlock()`	Unlocks the associated mutex.
`lk.release()`	Release the mutex. The mutex remains locked.
`lk.swap(lk2)`	Swaps the locks. Same as `std::swap(lk, lk2)`.
`lk.mutex()`	Returns a pointer to the associated mutex.
`lk.owns_lock()` and `operator bool`	Checks if the lock `lk` has a locked mutex.

`lk.try_lock_for(relTime)` needs a relative time duration; `lk.try_lock_until(absTime)` an absolute time point. `lk.try_lock_for(lk.try_lock_until)` calls effectively the member function `mut.try_lock_-for(mut.try_lock_until)` on the associated mutex `mut`. The associated mutex has to support exclusive timed blocking. You should use a steady clock for your time constraint. A steady clock can not be adjusted.

`lk.try_lock` tries to lock the mutex and returns immediately. On success, it returns true, otherwise false. In contrast, the member functions `lk.try_lock_for` and `lk.try_lock_until` the lock `lk` blocks until the specified timeout occurs or the lock is acquired, whichever comes first. All three member functions `lk.try_lock`, `lk.try_lock_for`, and `lk.try_lock_until` throw a `std::system_error` exception if there is no associated mutex or if the mutex is already locked by this `std::unique_lock`.

`lk.release()` removes the underlying mutex from the lock object's control and returns a pointer to that mutex. After calling this member function, the caller is responsible for releasing it because the mutex is not associated with the lock object anymore.

3.3.2.4 `std::shared_lock`

With C++14, C++ adds support for `std::shared_lock`.

A `std::shared_lock` has the same interface as a `std::unique_lock` but behaves differently when used with a `std::shared_timed_mutex` or a `std::shared_mutex`. Many threads can share one `std::shared_-timed_mutex` (`std::shared_mutex`) and, therefore, implement a reader-writer lock. The idea of reader-writer locks is straightforward and extremely useful. An arbitrary number of threads executing read operations can access the critical region simultaneously, but only one thread is allowed to write.

Reader-writer locks do not solve the fundamental problem - threads competing for access to a critical region, but they help minimize the bottleneck.

A telephone book is a typical example of using a reader-writer lock. Usually, many people want to look up a telephone number, but only a few want to change them. Let's look at an example.

Reader-writer locks

```cpp
1   // readerWriterLock.cpp
2
3   #include <iostream>
4   #include <map>
5   #include <shared_mutex>
6   #include <string>
7   #include <thread>
8
9   std::map<std::string,int> teleBook{{"Dijkstra", 1972}, {"Scott", 1976},
10                                      {"Ritchie", 1983}};
11
12  std::shared_timed_mutex teleBookMutex;
13
14  void addToTeleBook(const std::string& na, int tele){
15    std::lock_guard<std::shared_timed_mutex> writerLock(teleBookMutex);
16    std::cout << "\nSTARTING UPDATE " << na;
17    std::this_thread::sleep_for(std::chrono::milliseconds(500));
18    teleBook[na]= tele;
19    std::cout << " ... ENDING UPDATE " << na << '\n';
20  }
21
22  void printNumber(const std::string& na){
23    std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
24    std::cout << na << ": " << teleBook[na];
25  }
26
27  int main(){
28
29    std::cout << '\n';
30
31    std::thread reader1([]{ printNumber("Scott"); });
32    std::thread reader2([]{ printNumber("Ritchie"); });
33    std::thread w1([]{ addToTeleBook("Scott",1968); });
```

```
34    std::thread reader3([]{ printNumber("Dijkstra"); });
35    std::thread reader4([]{ printNumber("Scott"); });
36    std::thread w2([]{ addToTeleBook("Bjarne",1965); });
37    std::thread reader5([]{ printNumber("Scott"); });
38    std::thread reader6([]{ printNumber("Ritchie"); });
39    std::thread reader7([]{ printNumber("Scott"); });
40    std::thread reader8([]{ printNumber("Bjarne"); });
41
42    reader1.join();
43    reader2.join();
44    reader3.join();
45    reader4.join();
46    reader5.join();
47    reader6.join();
48    reader7.join();
49    reader8.join();
50    w1.join();
51    w2.join();
52
53    std::cout << '\n';
54
55    std::cout << "\nThe new telephone book" << '\n';
56    for (auto teleIt: teleBook){
57      std::cout << teleIt.first << ": " << teleIt.second << '\n';
58    }
59
60    std::cout << '\n';
61
62  }
```

The telephone book in line 9 is the shared variable, which has to be protected. Eight threads want to read the telephone book; two threads want to modify it (lines 31 - 40). To access the telephone book concurrently, the reading threads use the `std::shared_lock<std::shared_timed_mutex>>` in line 23. This is in contrast to the writing threads, which need exclusive access to the critical section. The exclusivity is given by the `std::lock_guard<std::shared_timed_mutex>>` in line 15. In the end, the program displays the updated telephone book (lines 55 - 58).

Reader-writer lock for reading and writing of a telephone book

The screenshot shows that the reading threads' output overlaps while the writing threads are executed one after the other. This means that the reading operations are performed at the same time.
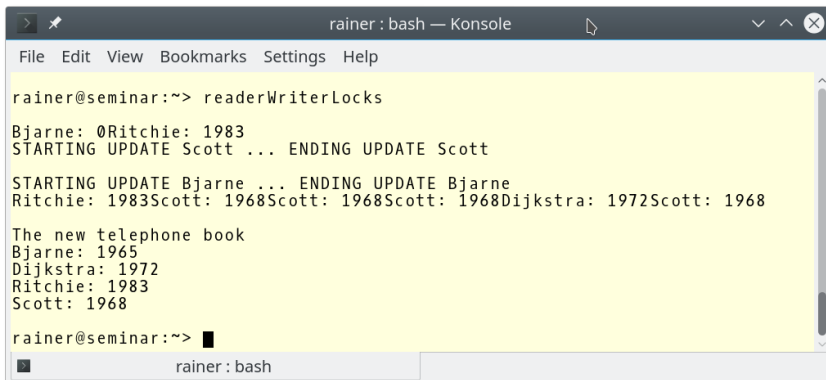
That was easy. Too easy. The telephone book has undefined behavior.

3.3.2.4.1 Undefined behavior

The program has undefined behavior. To be more precise it has a data race. What? Before you continue, stop for a few seconds and think. By the way, the concurrent access to `std::cout` is not the issue.

The characteristic of a data race is that at least two threads access the shared variable simultaneously, and at least one of them is a writer. This exact scenario may occur during program execution. One of the associative containers' features is that reading of the container using the index operator can modify it. This happens if the element is not available in the container. If "Bjarne" is not found in the telephone book, a pair ("Bjarne", 0) is created from the read access. You can force the data race by putting the printing of Bjarne in line 40 in front of all the threads (lines 31 - 40). Let's have a look.

You can see it right at the top. Bjarne has the value 0.

The program has a data race

An obvious way to fix this issue is to use only reading operations in the function `printNumber`:
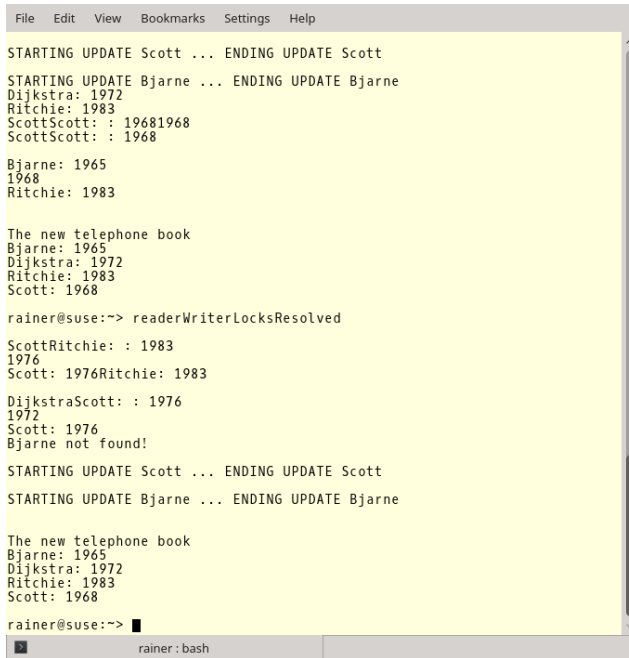
Reader-writer locks resolved

```cpp
// readerWriterLocksResolved.cpp

...

void printNumber(const std::string& na){
  std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
  auto searchEntry = teleBook.find(na);
  if(searchEntry != teleBook.end()){
    std::cout << searchEntry->first << ": " << searchEntry->second << '\n';
  }
  else {
    std::cout << na << " not found!" << '\n';
  }
}

...
```

If a key is not in the telephone book, I just write the key and the text `not found!` to the console.

Data race resolved

You can see the message `Bjarne not found!` in the second program execution. In the first program execution, `addToTeleBook` is executed first; therefore, Bjarne is found.

3.3.3 `std::lock`

`std::lock` can lock its Lockables[9] such as a mutex or a lock in an atomic step. A Lockable is a data type that supports the member functions `lock`, `unlock`, and `try_lock`.`std::lock` is a variadic template and can, therefore, accept an arbitrary number of arguments. `std::lock` tries to get all locks in one atomic step using a deadlock avoidance algorithm. The Lockable are locked by an unspecified series of calls to `lock`, `try_lock`, and `unlock`. If a call to `lock` causes an exception, `unlock` is called for any locked Lockable before rethrowing.

Remember the deadlock from the subsection Issues of Mutexes?

---

[9]https://en.cppreference.com/w/cpp/named_req/Lockable

Locking mutexes in different order

```cpp
1   // deadlock.cpp
2
3   #include <iostream>
4   #include <chrono>
5   #include <mutex>
6   #include <thread>
7
8   struct CriticalData{
9     std::mutex mut;
10  };
11
12  void deadLock(CriticalData& a, CriticalData& b){
13
14    a.mut.lock();
15    std::cout << "get the first mutex" << '\n';
16    std::this_thread::sleep_for(std::chrono::milliseconds(1));
17    b.mut.lock();
18    std::cout << "get the second mutex" << '\n';
19    // do something with a and b
20    a.mut.unlock();
21    b.mut.unlock();
22
23  }
24
25  int main(){
26
27    CriticalData c1;
28    CriticalData c2;
29
30    std::thread t1([&]{deadLock(c1,c2);});
31    std::thread t2([&]{deadLock(c2,c1);});
32
33    t1.join();
34    t2.join();
35
36  }
```

Let's solve the issue. The function deadLock has to lock its mutexes atomically, and that's exactly what happens in the following example.

Delayed locking of mutexes
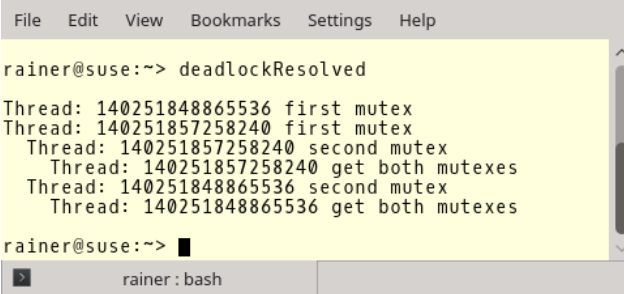
```cpp
1   // deadlockResolved.cpp
2
3   #include <iostream>
4   #include <chrono>
5   #include <mutex>
6   #include <thread>
7
8   using namespace std;
9
10  struct CriticalData{
11    mutex mut;
12  };
13
14  void deadLock(CriticalData& a, CriticalData& b){
15
16    unique_lock<mutex> guard1(a.mut,defer_lock);
17    cout << "Thread: " << this_thread::get_id() << " first mutex" << '\n';
18
19    this_thread::sleep_for(chrono::milliseconds(1));
20
21    unique_lock<mutex> guard2(b.mut,defer_lock);
22    cout << "  Thread: " << this_thread::get_id() << " second mutex" <<  '\n';
23
24    cout << "    Thread: " << this_thread::get_id() << " get both mutex" << '\n';
25    lock(guard1,guard2);
26    // do something with a and b
27  }
28
29  int main(){
30
31    cout << '\n';
32
33    CriticalData c1;
34    CriticalData c2;
35
36    thread t1([&]{deadLock(c1,c2);});
37    thread t2([&]{deadLock(c2,c1);});
38
39    t1.join();
40    t2.join();
41
42    cout << '\n';
43
44  }
```

If you call the constructor of `std::unique_lock` with `std::defer_lock`, the underlying mutex is not locked automatically. At this point (lines 16 and 21), the `std::unique_lock` is just the mutex owner. Thanks to the variadic template `std::lock`, the lock operation is performed in an atomic step (line 25).

In this example, `std::unique_lock` manages the lifetime of the resources and `std::lock` locks the associated mutex. You can do it the other way around. In the first step, the mutexes are locked. In the second `std::unique_lock` manages the lifetime of resources. Here is an example of the second approach.

```
std::lock(a.mut, b.mut);
std::lock_guard<std::mutex> guard1(a.mut, std::adopt_lock);
std::lock_guard<std::mutex> guard2(b.mut, std::adopt_lock);
```

Both variants resolve the deadlock.

Deadlock resolved with `std::unique_lock`

Resolving the deadlock with a `std::scoped_lock`

With C++17, the resolution of the deadlock becomes quite easy. We have the `std::scoped_lock` that can lock an arbitrary number of mutexes atomically. You only have to use a `std::scoped_lock` instead of the `std::lock` call. That's all. Here is the modified function `deadlock`.

Deadlock resolved with `std::scoped_lock`

```cpp
1   // deadlockResolvedScopedLock.cpp
2
3   ...
4   void deadLock(CriticalData& a, CriticalData& b){
5
6     cout << "Thread: " << this_thread::get_id() << " first mutex" << '\n';
7     this_thread::sleep_for(chrono::milliseconds(1));
8     cout << "  Thread: " << this_thread::get_id() << " second mutex" <<  '\n';
9     cout << "    Thread: " << this_thread::get_id() << " get both mutex" << '\n';
10
11    std::scoped_lock(a.mut, b.mut);
12    // do something with a and b
13  }
14
15  ...
```

3.3.4 Thread-safe Initialization

If the variable is never modified, synchronization is no need by using an expensive lock or an atomic. You only have to ensure that it is initialized in a thread-safe way.

There are three ways in C++ to initialize variables in a thread-safe way.

- Constant expressions
- The function `std::call_once` in combination with the flag `std::once_flag`
- A static variable with block scope

Thread-safe Initialization in the main thread

The easiest way to initialize a variable in a thread-safe way is to initialize the variable in the main-thread before you create any child threads.

3.3.4.1 Constant Expressions

Constant expressions are expressions that the compiler can evaluate at compile time. They are implicitly thread-safe. Placing the keyword constexpr in front of a variable makes the variable a constant expression. The constant expression must be initialized immediately.

```
constexpr double pi = 3.14;
```

Besides, user-defined types can also be constant expressions. For those types, there are a few restrictions that must meet to initialize it at compile-time.

- They must not have virtual member functions or a virtual base class.
- Their constructor must be a constant expression.
- Every base class and each non-static member must be initialized.
- Their member functions, which should be callable at compile time, must be constant expressions.

Instances of MyDouble satisfy all these requirements. So it is possible to instantiate them at compile time. This instantiation is thread-safe.

User defined constant expressions

```
1   // constexpr.cpp
2
3   #include <iostream>
4
5   class MyDouble{
6     private:
7       double myVal1;
8       double myVal2;
9     public:
10      constexpr MyDouble(double v1, double v2): myVal1(v1), myVal2(v2){}
11      constexpr double getSum() const { return myVal1 + myVal2; }
12  };
13
14  int main() {
15
16    constexpr double myStatVal = 2.0;
17    constexpr MyDouble myStatic(10.5, myStatVal);
18    constexpr double sumStat = myStatic.getSum();
19
20  }
```

3.3.4.2 `std::call_once` and `std::once_flag`

By using the `std::call_once` function, you can register a callable. The `std::once_flag` ensures that only one registered function is invoked. You can register additional functions via the same `std::once_flag`. Only one function from that group is called.

`std::call_once` obeys the following rules:

- Exactly one execution of exactly one of the functions is performed. It is undefined which function is selected for execution. The selected function runs in the same thread as the `std::call_once` invocation it was passed to.

- No invocation in the group returns before the execution mentioned above of the selected function completes successfully.

- If the selected function exits via an exception, it is propagated to the caller. Another function is then selected and executed.

The short example demonstrates the application of `std::call_once` and the `std::once_flag`. Both of them are declared in the header `<mutex>`.

Use of `std::call_once` and the `std::once_flag`

```cpp
1   // callOnce.cpp
2
3   #include <iostream>
4   #include <thread>
5   #include <mutex>
6
7   std::once_flag onceFlag;
8
9   void do_once(){
10    std::call_once(onceFlag, [](){ std::cout << "Only once." << '\n'; });
11  }
12
13  void do_once2(){
14    std::call_once(onceFlag, [](){ std::cout << "Only once2." << '\n'; });
15  }
16
17  int main(){
18
19    std::cout << '\n';
20
21    std::thread t1(do_once);
22    std::thread t2(do_once);
23    std::thread t3(do_once2);
24    std::thread t4(do_once2);
25
```

```
26     t1.join();
27     t2.join();
28     t3.join();
29     t4.join();
30
31     std::cout << '\n';
32
33  }
```

The program starts with four threads (lines 21 - 24). Two of them invoke do_once and the other two do_once2. The expected result is that the string "Only once" or the string "Only once2" is displayed only once.
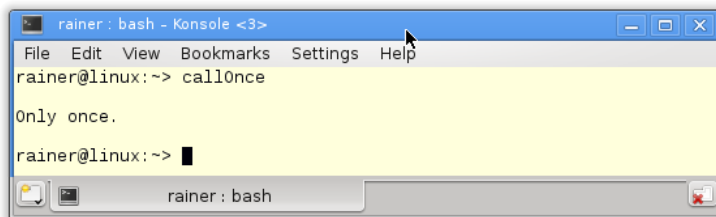
Usage of `std::call_once` and `std::once_flag`

The famous singleton pattern guarantees that only one instance of a class is created. This is a challenging task in multithreading environments. Thanks to std::call_once and std::once_flag the job is a piece of cake.

Now the singleton is initialized in a thread-safe way.

Singleton pattern with `std::call_once` and the `std::once_flag`

```cpp
1   // singletonCallOnce.cpp
2
3   #include <iostream>
4   #include <mutex>
5
6   using namespace std;
7
8   class MySingleton{
9
10    private:
11      static once_flag initInstanceFlag;
12      static MySingleton* instance;
13      MySingleton() = default;
14      ~MySingleton() = default;
15
```

```
16        static void initSingleton(){
17           instance = new MySingleton();
18        }
19
20     public:
21        MySingleton(const MySingleton&) = delete;
22        MySingleton& operator=(const MySingleton&) = delete;
23
24        static MySingleton* getInstance(){
25           call_once(initInstanceFlag, MySingleton::initSingleton);
26           return instance;
27        }
28
29  };
30
31  MySingleton* MySingleton::instance = nullptr;
32  once_flag MySingleton::initInstanceFlag;
33
34  int main(){
35
36     cout << '\n';
37
38     cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << '\n';
39     cout << "MySingleton::getInstance(): "<< MySingleton::getInstance() << '\n';
40
41     cout << '\n';
42
43  }
```

Let's first review the static flag initInstanceFlag. It is declared in line 11 and initialized in line 17. The static member function getInstance (lines 24 - 27) uses the flag initInstanceFlag to ensure that the static member function initSingleton (line 16 - 18) is executed exactly once. The singleton is created in the body of the member function.
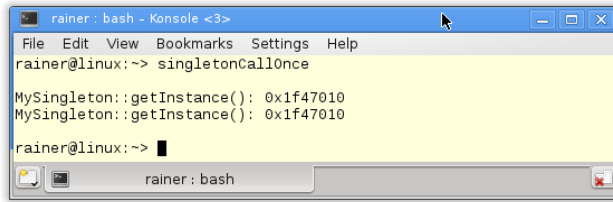
default **and** delete

You can request special member functions from the compiler by using the keyword default. These member functions are special because the compiler can create them for us.

The result of annotating a member function with delete is that the compiler-generated member function is not available and, therefore, cannot be called. If you try to use them, you'll get a compile-time error. Here are the details for the keywords default and delete[10].

---

[10]https://isocpp.org/wiki/faq/cpp11-language-classes

The `MySingleton::getIstance()` member function displays the address of the singleton.

Thread-safe implementation of the singleton with `std::call_once` and `std::once_flag`

3.3.4.3 Static Variables with Block Scope

Static variables with block scope are created exactly once and lazily. Lazily means that they are created just at the moment of usage. This characteristic is the basis of the so-called Meyers Singleton, named after Scott Meyers[11]. This is by far the most elegant implementation of the singleton pattern in C++. With C++11, static variables with block scope have an additional guarantee; they are initialized in a thread-safe way.

Here is the thread-safe Meyers Singleton pattern.

The thread-safe Meyers Singleton pattern

```cpp
// meyersSingleton.cpp

class MySingleton{
public:
  static MySingleton& getInstance(){
    static MySingleton instance;
    return instance;
  }
private:
  MySingleton();
  ~MySingleton();
  MySingleton(const MySingleton&)= delete;
  MySingleton& operator=(const MySingleton&)= delete;

};

MySingleton::MySingleton()= default;
MySingleton::~MySingleton()= default;

int main(){

```

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

```
22    MySingleton::getInstance();
23
24  }
```

Know your Compiler support for static

If you use the Meyers Singleton in a concurrent environment, be sure that your compiler implements static variables with the C++11 thread-safe semantic. It happens quite often that programmers rely on the C++11 semantic of static variables, but their compiler does not support it. The result may be that more than one instance of a singleton is created.

`thread_local` data has no sharing issues.

3.4 Thread-Local Data

Thread-local data, also known as thread-local storage, is created for each thread separately. It behaves like static data because it's bound for the thread's lifetime and is created at its first usage. This means that thread-local variables at namespace scope or as static class members are created before its first use, and those thread-local variables declared in a function are created with its first use. Thread-local data belongs exclusively to the thread.

Thread-local data

```cpp
// threadLocal.cpp

#include <iostream>
#include <string>
#include <mutex>
#include <thread>

std::mutex coutMutex;

thread_local std::string s("hello from ");

void addThreadLocal(std::string const& s2){

  s += s2;
  // protect std::cout
  std::lock_guard<std::mutex> guard(coutMutex);
  std::cout << s << '\n';
  std::cout << "&s: " << &s << '\n';
  std::cout << '\n';

}

int main(){

  std::cout << '\n';

  std::thread t1(addThreadLocal,"t1");
  std::thread t2(addThreadLocal,"t2");
  std::thread t3(addThreadLocal,"t3");
  std::thread t4(addThreadLocal,"t4");

  t1.join();
  t2.join();
  t3.join();
  t4.join();
```

```
36
37    std::cout << s << '\n';
38    std::cout << "&s: " << &s << '\n';
39
40  }
```

Using the keyword `thread_local` in line 10, the `thread-local` string s is created. Threads t1 - t4 (lines 27 - 30) use the function `addThreadLocal` (lines 12 - 21) as their work package. Threads get as their argument the strings t1 to t4 respectively and add them to the thread-local string s. Also, `addThreadLocal` displays the address of s in line 18. The main thread is also a thread and it gets its own copy of s (line 37).

Thread-local data

The output of the program shows it implicitly in line 17 and explicitly in line 18. The thread-local string is created for each thread t1 - t4, and the main thread. First, each output shows a new thread-local string. Second, each string s has a different address.

I often discuss in my seminars: What is the difference between a `static`, a `thread_local`, and a local variable? A `static` variable is bound to the lifetime of the main thread, a `thread_local` variable is bound to the lifetime of its thread, and a local variable is bound to the lifetime of the scope in which it was created. Here is a variation of the previous program `threadLocal.cpp` to make my point clear.

State between function calls
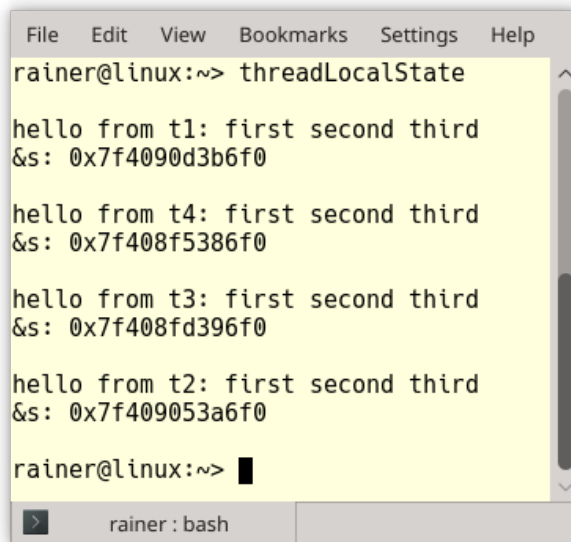
```cpp
1    // threadLocalState.cpp
2
3    #include <iostream>
4    #include <string>
5    #include <mutex>
6    #include <thread>
7
8    std::mutex coutMutex;
9
10   thread_local std::string s("hello from ");
11
12   void first(){
13       s += "first ";
14   }
15
16   void second(){
17       s += "second ";
18   }
19
20   void third(){
21       s += "third";
22   }
23
24   void addThreadLocal(std::string const& s2){
25
26     s += s2;
27
28     first();
29     second();
30     third();
31     // protect std::cout
32     std::lock_guard<std::mutex> guard(coutMutex);
33     std::cout << s << '\n';
34     std::cout << "&s: " << &s << '\n';
35     std::cout << '\n';
36
37   }
38
39   int main(){
40
41     std::cout << '\n';
42
43     std::thread t1(addThreadLocal,"t1: ");
44     std::thread t2(addThreadLocal,"t2: ");
```

```
45    std::thread t3(addThreadLocal,"t3: ");
46    std::thread t4(addThreadLocal,"t4: ");
47
48    t1.join();
49    t2.join();
50    t3.join();
51    t4.join();
52
53  }
```

In this variation to the previous program, the function addThreadLocal (line 24) invokes the functions first, second, and third. Each of the function uses the thread_local string s to add its function name. The key point of this variation is that the string s is used as a thread-local state between the function called first, second, and third (lines 28 - 30). The output shows that the strings are independent.

Thread-local data

From a Single-Threaded to a Multithreaded Program.

Thread-local data helps to port a single-threaded program to a multithreaded environment. If the global variables are thread-local, there is the guarantee that each thread gets its copy of the data. Due to this fact, there is no shared mutable state which may cause a data race resulting in undefined behavior.

In contrast to thread-local data, condition variables are not easy to use.

3.5 Condition Variables

Condition variables enable threads to be synchronized via messages. They need the `<condition_-variable>` header. One thread acts as a sender, and the other as a receiver of the message. The receiver waits for the notification from the sender. Typical use cases for condition variables are sender-receiver or producer-consumer workflows.

A condition variable can be the sender but also the receiver of the message.

The member functions of the condition variable cv

Member function	Description
`cv.notify_one()`	Notifies a waiting thread.
`cv.notify_all()`	Notifies all waiting threads.
`cv.wait(lock, ...)`	Waits for the notification.
`cv.wait_for(lock, relTime, ...)`	Waits for a time duration for the notification.
`cv.wait_until(lock, absTime, ...)`	Waits until a time point for the notification.
`cv.native_handle()`	Returns the native handle of this condition variable.

The subtle difference between `cv.notify_one` and `cv.notify_all` is that `cv.notify_all` notifies all waiting threads. In contrast, `cv.notify_one` notifies only one of the waiting threads. The other condition variables do stay in the wait state. Here is an example before we cover the gory details - which are the three dots in the wait operations - of condition variables.

First usage of condition variables

```cpp
// conditionVariable.cpp

#include <iostream>
#include <condition_variable>
#include <mutex>
#include <thread>

std::mutex mutex_;
std::condition_variable condVar;

bool dataReady{false};

void doTheWork(){
  std::cout << "Processing shared data." << '\n';
}

void waitingForWork(){
    std::cout << "Worker: Waiting for work." << '\n';
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
```

```
21        doTheWork();
22        std::cout << "Work done." << '\n';
23    }
24
25    void setDataReady(){
26        {
27            std::lock_guard<std::mutex> lck(mutex_);
28            dataReady = true;
29        }
30        std::cout << "Sender: Data is ready."  << '\n';
31        condVar.notify_one();
32    }
33
34    int main(){
35
36      std::cout << '\n';
37
38      std::thread t1(waitingForWork);
39      std::thread t2(setDataReady);
40
41      t1.join();
42      t2.join();
43
44      std::cout << '\n';
45
46    }
```

The program has two child threads: `t1` and `t2`. They get their work package `waitingForWork` and `setDataRead` in lines 38 and 39. `setDataReady` notifies - using the condition variable `condVar` - that it is done with the preparation of the work: `condVar.notify_one()`. While holding the lock, thread `t1` waits for its notification: `condVar.wait(lck, []{ return dataReady; })`. The sender and receiver need a lock. In the case of the sender, a `std::lock_guard` is sufficient because it calls lock and unlock only once. In the receiver's case, a `std::unique_lock` is necessary because it frequently locks and unlocks its mutex.

Here is the output of the program.

Synchronization of two threads with condition variables

std::condition_variable_any

std::condition_variable can only wait on an object of type std::unique_lock<mutex>
but std::condition_variable_any can wait on an user-supplied lock type that meets the
concept of BasicLockable[12]. The generalised std::condition_variable_any supports the
same interface such as std::condition_variable.

3.5.1 The Predicate

Maybe you wonder why you need a predicate for the wait call because you can invoke wait without
a predicate. Let's try it out.

Blocking condition variables

```
// conditionVariableBlock.cpp

#include <iostream>
#include <condition_variable>
#include <mutex>
#include <thread>

std::mutex mutex_;
std::condition_variable condVar;

void waitingForWork(){

    std::cout << "Worker: Waiting for work." << '\n';

    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck);
```

---

[12]http://en.cppreference.com/w/cpp/concept/BasicLockable

```
17        // do the work
18        std::cout << "Work done." << '\n';
19
20    }
21
22    void setDataReady(){
23
24        std::cout << "Sender: Data is ready."  << '\n';
25        condVar.notify_one();
26
27    }
28
29    int main(){
30
31      std::cout << '\n';
32
33      std::thread t1(setDataReady);
34      std::thread t2(waitingForWork);
35
36      t1.join();
37      t2.join();
38
39      std::cout << '\n';
40
41    }
```

The first invocation of the program seems to work fine. The second invocation locks because the notification call (line 25) happens before thread t2 (line 34) enters its waiting state (line 16).

Deadlock with condition variables

Now it is clear. The predicate is a kind of memory for the stateless condition variable; therefore, the wait call always checks the predicate at first. Condition variables are a victim to two known phenomena: lost wakeup and spurious wakeup.

3.5.2 Lost Wakeup and Spurious Wakeup

Lost Wakeup

The lost wakeup phenomenon is that the sender sends its notification before the receiver gets to its wait state. The consequence is that the notification is lost. The C++ standard describes condition variables as a simultaneous synchronization mechanism: "The condition_variable class is a synchronization primitive that can be used to block a thread, or multiple threads **at the same time**, ...". So the notification gets lost, and the receiver is waiting and waiting and ... .

Spurious Wakeup

The receiver may wake up, although no notification happened. At a minimum POSIX Threads[13] and the Windows API[14] can be victims of these phenomena. One reason for a spurious wakeup can be a stolen wakeup. This means, before the awoken thread gets the chance to run, another thread kicks in and runs.

3.5.3 The Wait Workflow

The waiting thread has quite a complicated workflow.

Here are the two fundamental lines 19 and 20 from the previous example `conditionVariable.cpp`.

```
std::unique_lock<std::mutex> lck(mutex_);
condVar.wait(lck, []{ return dataReady; });
```

The two lines are equivalent to the following four lines:

```
std::unique_lock<std::mutex> lck(mutex_);
while ( ![]{ return dataReady; }() ) {
    condVar.wait(lck);
}
```

First, you have to distinguish between the first call of `std::unique_lock<std::mutex> lck(mutex_)` and the notification of the condition variable: `condVar.wait(lck)`.

- **`std::unique_lock<std::mutex> lck(mutex_)`:** In the initial processing, the thread locks the mutex and then check the predicate `[]{ return dataReady; }`.
  - If the call of the predicated evaluates to
    - **`true`:** the thread continues its work.
    - **`false`:** `condVar.wait()` unlocks the mutex and puts the thread in a waiting (blocking) state

---

[13]https://en.wikipedia.org/wiki/POSIX_Threads
[14]https://en.wikipedia.org/wiki/Windows_API

- **condVar.wait(lck)**: If the condition_variable condVar is waiting and gets a notification or a spurious wakeup the following steps happen.

  - The thread is unblocked and reacquires the lock on the mutex.
  - The thread checks the predicate.
  - If the call of the predicated evaluates to
    * **true**: the thread continues its work.
    * **false**: condVar.wait() unlocks the mutex and puts the thread in a waiting (blocking) state.

Even if the shared variable is atomic, it must be modified under the mutex to correctly publish the modification to the waiting thread.

⚠ Use a mutex to protect the shared variable

Even if you make dataReady an atomic, it must be modified under the mutex; if not the modification to the waiting thread may be published but not correctly synchronized. This race condition may cause a deadlock. What does that mean: published, but not correctly synchronized. Let's have a closer look at the wait workflow and assume that deadReady is atomic and is modified, not protected by the mutex mutex_.

```
1   std::unique_lock<std::mutex> lck(mutex_);
2   while ( ![]{ return dataReady.load(); }() ) {
3       // time window
4       condVar.wait(lck);
5   }
```

Let me assume the notification is send while the condition variable condVar is not waiting. This means the thread's execution is in the source snippet between line 2 and 4 (see the comment time window). The result is that the notification is lost. Afterward, the thread goes back waiting and presumably sleeps forever.

This wouldn't have happened if a mutex had protected ' dataReady'. Because of the synchronization with the mutex, the notification would only be sent if the condition variable and, therefore, the receiver thread is waiting.

In most of the use-cases, tasks are the less error-prone way to synchronize threads. I compare condition variables and tasks in the section Returning a Notification.

3.6 Cooperative Interruption (C++20)

The additional functionality of the cooperative interruption thread is based on the `std::stop_source`, `std::stop_token`, and the `std::stop_callback` classes. `std::jthread` and `std::condition_variable_-any` support cooperative interruption by design.

First, why it is not a good idea to kill a thread?

Killing a Thread is Dangerous

Killing a thread is dangerous because you don't know the state of the thread. Here are two possible malicious outcomes.

- The thread is only half-done with its job. Consequently, you don't know the state of its job and, hence, the state of your program. You end with undefined behavior, and all bets are off.
- The thread may be in a critical section and having locked a mutex. Killing a thread while it locks a mutex ends with a high probability in a deadlock.

The `std::stop_source`, `std::stop_token`, and the `std::stop_callback` classes allows a thread to asynchronously request an execution to stop or ask if an execution got a stop signal. The `std::stop_token` can be passed to an operation and afterward be used to actively poll the token for a stop request or to register a callback via `std::stop_callback`. The stop request is sent by a `std::stop_source`. This signal affects all associated `std::stop_token`. The three classes `std::stop_source`, `std::stop_token`, and the `std::stop_callback` share the ownership of an associated stop state.

In the next subsecions, I provide more details about cooperative interruption.

3.6.1 `std::stop_source`

You can construct a `std::stop_source` in two ways:

Constructors of `std::stop_source`

```
1  std::stop_source();
2  explicit std::stop_source(std::nostopstate_t) noexcept;
```

The default constructor (line 1) constructs a `std::stop_source` with a new stop state. The constructor taking `std::nostopstate_t` (line 2) constructs an empty `std::stop_source` without associated stop state.

The component `std::stop_source src` provides the following member functions for handling stop requests.

<div align="center">Member functions of `std::stop_source src`</div>

Member function	Description
`std::stop_source src`	The default constructor creates a stop source with an associated stop state.
`std::stop_source src{nostopstate}`	Creates a stop_source without associated stop state.
`src.get_token()`	If `src.stop_possible()`, returns a `stop_token` for the associated stop state. Otherwise, returns a default-constructed (empty) `stop_token`.
`src.stop_possible()`	`true` if `src` can be requested to stop.
`src.stop_requested()`	`true` if `stop_possible()` and `request_stop()` was called by one of the owners.
`src.request_stop()`	Calls a stop request if `src.stop_possible()` and `!src.stop_requested()`. Otherwise, the call has no effect.

The call `src.get_token()` returns the stop token `stoken`. Thanks to `stoken` you can check if a stop request has been made or can be made by its associated stop source `src`. The stop token `stoken` observes the stop source `src`.

`src.stop_requested()` returns `true` when `src` has an associated stop state and was asked to stop earlier.

`src.stop_possible()` return `false` if there is no associated stop stare or no stop source anymore and stop was never requested before.

The calls `src.stop_possible()`, `src.stop_requested()`, and `src.request_stop()` are thread-safe.

`src.request_stop()` of a stop source `src` is visible to all `std::stop_token` and registered callback of the same associated stop state. Also, any `std::condiction_variable_any` waiting on the associated `std::stop_token() will be awoken. When a stop is requested, it cannot be withdrawn.` src.request_-stop() `is successful and returns` true `if` src` has an associated stop state and it was not requested to stop before.

3.6.2 `std::stop_token`

`std::stop_token` is essentially a thread-safe "view" of the associated stop state. It is typically retrieved from a `std::jthread` or a `std::stop_source src` via `src.get_token()`. This causes them share the same associated stop state as the `std::jthread` or `std::stop_source`.

Thanks to the `std::stop_token`, you can check for the associated `std::stop_source` if a stop request has been made.

The `std::stop_token` can also be passed to the constructor of `std::stop_callback`, or to the interruptible waiting functions of `std::condition_variable_any`.

<div align="center">**Member functions of `std::stop_token stoken`**</div>

Member function	Description
`std::stop_token stoken`	The default constructor creates a stop token with an associated stop state.
`stoken.stop_possible()`	Returns `true` if `stoken` has an associated stop state.
`stoken.stop_requested()`	`true` if `request_stop()` was called on the associated `std::stop_source src`, otherwise `false`.

`stoken.stop_possible()` also returns `true` if the stop request has already been made.

`stoken.stop_requested()` returns `true` when the stop token has an associated stop state and has already received a stop request.

If the `std::stop_token` should be temporarily disabled, you can replace it with a default-constructed token. A default-constructed token has no associated stop state. The following code snippet shows how to disable and enable a thread's capability to accept stop requests.

Temporarily disable a stop token

```
1   std::jthread jthr([](std::stop_token stoken) {
2       ...
3       std::stop_token interruptDisabled;
4       std::swap(stoken, interruptDisabled);
5       ...
6       std::swap(stoken, interruptDisabled);
7       ...
8   }
```

`std::stop_token interruptDisabled` has no associated stop state. This means the thread `jthr` can accept stop requests in all lines except 4 and 5.

3.6.3 `std::stop_callback`

A `std::stop_callback` models RAII. It's constructor registers a callable for a stop token and it's destructor unregisters it. The following example shows the use of `std::stop_callback`.

Use of callbacks

```cpp
1    // invokeCallback.cpp
2
3    #include <atomic>
4    #include <chrono>
5    #include <iostream>
6    #include <thread>
7    #include <vector>
8
9    using namespace std::literals;
10
11   auto func = [](std::stop_token stoken) {
12           int counter{0};
13           auto thread_id = std::this_thread::get_id();
14           std::stop_callback callBack(stoken, [&counter, thread_id] {
15               std::cout << "Thread id: " << thread_id
16                         << "; counter: " << counter << '\n';
17           });
18           while (counter < 10) {
19               std::this_thread::sleep_for(0.2s);
20               ++counter;
21           }
22       };
23
24   int main() {
25
26       std::cout << '\n';
27
28       std::vector<std::jthread> vecThreads(10);
29       for(auto& thr: vecThreads) thr = std::jthread(func);
30
31       std::this_thread::sleep_for(1s);
32
33       for(auto& thr: vecThreads) thr.request_stop();
34
35       std::cout << '\n';
36
37   }
```

Each of the ten threads invokes the lambda function `func` (lines 11 - 22). The callback in lines 14 - 17 displays the thread id and the local counter. Due to the 1-second sleeping of the main thread and the sleeping of the child threads, the counter is four when the callbacks are invoked. The call `thr.request_stop()` triggers the callback on each thread using the `std::stop_token`. The chapter The Improved Thread `std::jthread` provides more details about cooperative interruption of a thread.

Use of callbacks

The `std::stop_callback` constructor registers the callback function for the `std::stop_token` given by the associated `std::stop_source`. This callback function is either invoked in the thread invoking `request_stop()` or the thread constructing the `std::stop_callback`. If the request to stop happens prior to the registration of the `std::stop_callback`, the callback is invoked in the thread constructing the `std::stop_callback`. Otherwise, the callback is invoked in the thread invoking `request_stop`.

You can register more than one callback for one or more threads using the same `std::stop_token`. The C++ standard provides no guarantee in which order they are executed.

Stop request arrives before the callback registration

```cpp
1   // invokeCallbacks.cpp
2
3   #include <chrono>
4   #include <iostream>
5   #include <thread>
6
7   using namespace std::literals;
8
9   void func(std::stop_token stopToken) {
10      std::this_thread::sleep_for(100ms);
11      for (int i = 0; i <= 9; ++i) {
12          std::stop_callback cb(stopToken, [i] { std::cout << i; });
13      }
14      std::cout << '\n';
15  }
16
17  int main() {
```

```
18
19        std::cout << '\n';
20
21        std::jthread thr1 = std::jthread(func);
22        std::jthread thr2 = std::jthread(func);
23        thr1.request_stop();
24        thr2.request_stop();
25
26        std::cout << '\n';
27
28    }
```
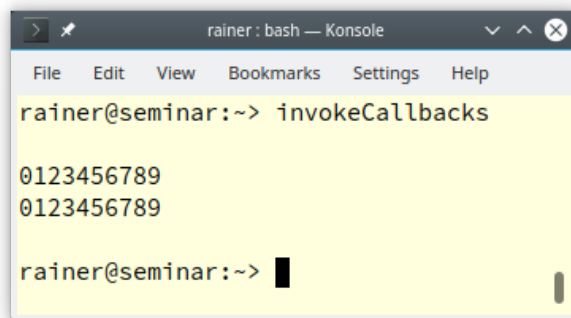

Stop request arrives before the callback registration

The previous program `invokeCallbacks.cpp` illustrates when the stop request arrives before the callback registration, and thus the child thread has to execute each handler during registration.

The following program, `invokeCallbacksLoop.cpp`, illustrates when the callback has been registered before the stop request.

Stop request arrives after the callback registration

```cpp
1     // invokeCallbacksLoop.cpp
2
3     #include <chrono>
4     #include <functional>
5     #include <iostream>
6     #include <list>
7     #include <stop_token>
8     #include <thread>
9
10    using namespace std::literals;
11
```
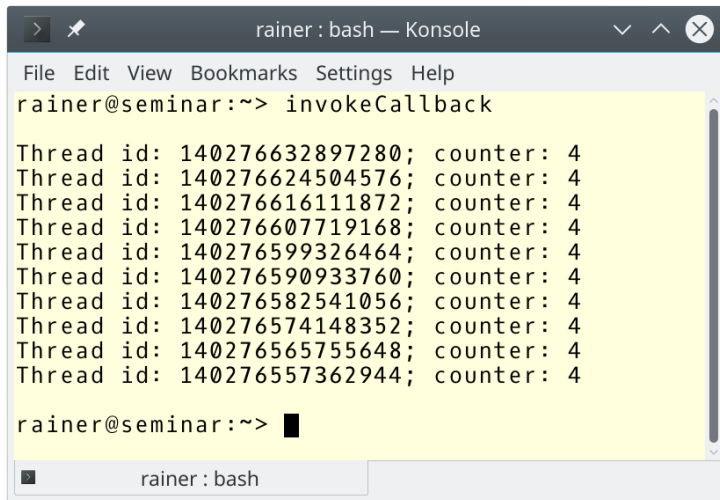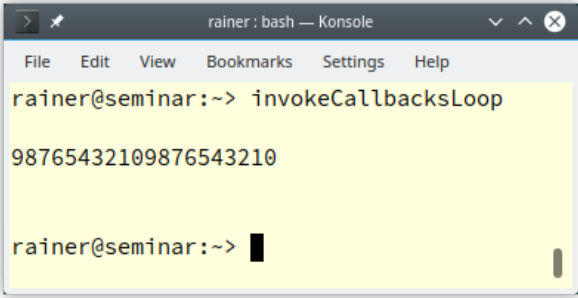
```cpp
12    using handler_t = std::function<void()>;
13    using callback_t = std::stop_callback<handler_t>;
14
15    void func(std::stop_token stopToken) {
16        std::list<callback_t> callbacks;
17        for (int i = 0; i <= 9; ++i) {
18            callbacks.emplace_back(stopToken, [i] { std::cout << i; });
19        }
20        std::this_thread::sleep_for(100ms);
21        std::cout << '\n';
22    }
23
24    int main() {
25
26        std::cout << '\n';
27
28        std::jthread thr1 = std::jthread(func);
29        std::jthread thr2 = std::jthread(func);
30        std::this_thread::sleep_for(50ms);
31        thr1.request_stop();
32        thr2.request_stop();
33
34        std::cout << '\n';
35
36    }
```


Stop request arrives after the callback registration

3.6.4 A General Mechanism to Send Signals

The pair std::stop_source and std::stop_token can be considered as a general mechanism to send a signal. By copying the std::stop_token, you can send the signal to any entity executing something.

In the following example, I use `std::async`, `std::promise`, `std::thread`, and `std::jthread` in various combinations.

Sending a signal to various executing entities

```cpp
1   // signalStopRequests.cpp
2
3   #include <iostream>
4   #include <thread>
5   #include <future>
6
7   using namespace std::literals;
8
9   void function1(std::stop_token stopToken, const std::string& str){
10      std::this_thread::sleep_for(1s);
11      if (stopToken.stop_requested()) std::cout << str << ": Stop requested\n";
12  }
13
14  void function2(std::promise<void> prom,
15                 std::stop_token stopToken, const std::string& str) {
16      std::this_thread::sleep_for(1s);
17      std::stop_callback callBack(stopToken, [&str] {
18          std::cout << str << ": Stop requested\n";
19      });
20      prom.set_value();
21  }
22
23  int main() {
24
25      std::cout << '\n';
26
27      std::stop_source stopSource;
28
29      std::stop_token stopToken = std::stop_token(stopSource.get_token());
30
31      std::thread thr1 = std::thread(function1, stopToken, "std::thread");
32
33      std::jthread jthr = std::jthread(function1, stopToken, "std::jthread");
34
35      auto fut1 = std::async([stopToken] {
36          std::this_thread::sleep_for(1s);
37          if (stopToken.stop_requested()) std::cout << "std::async: Stop requested\n";
38      });
39
40      std::promise<void> prom;
41      auto fut2 = prom.get_future();
```

```
42      std::thread thr2(function2, std::move(prom), stopToken, "std::promise");

43

44      stopSource.request_stop();
45      if (stopToken.stop_requested()) std::cout << "main: Stop requested\n";

46

47      thr1.join();
48      thr2.join();

49

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

51

52  }
```

Thanks to the `stopSource` (line 27), I can create the `stopToken` (line 29). Each running entity such as `std::thread` (line 31), `std::jthread` (line 33), `std::async` (line 35), or `std::promise` (line 42). A `std::stop_token` is cheap to copy. Line 44 triggers `stopSource.request_stop`. Also the `main`-thread (line 45) gets the signal. I use in this example `std::jthread`. `std::jthread` has explicit member functions to deal with cooperative interruption more conveniently. Read more about it the following section Joining Thread.

Sending a signal to various executing entities

You may wonder why the various executing entities sleep for one second (lines 10, 16, and 36) in the previous program `signalStopRequests.cpp`? I want to be sure that the call `stopSource.request_stop()` in line 44 has an effect. The execution entity such as the `std::thread` (line 31), the `std::jthread` (line 33), `std::async` (line 35), or `std::promise` (line 42) can be in one the following states, when the request to stop is signaled.

- Not started: The call `stopToken.stop_requested` returns `true` when executed. The callback is executed when `stopSource.request_stop` is signaled.
- Executing: The execution entity receives the signal. To take an effect, the `stopSource.request_-stop` must happen before the running entity calls `stopToken.stop_requested`. Accordingly, the `stopSource.request_stop` must happen before the callback is initialized.

- Finished: The call `stopSource.request_stop` has no effect. The callback is not executed.

Let's see what happens when I join the threads `thr1` and `thr2` before the call `stopSource.request_stop` in the previous program `signalStopRequests.cpp`? Here are the lines 44 and 45 swapped with the lines 47 and 48.

Sending the signal too late

```
44      thr1.join();
45      thr2.join();
46
47      stopSource.request_stop();
48      if (stopToken.stop_requested()) std::cout << "main: Stop requested\n";
```

The swap of the lines affects that only the `main`-thread reacts to the signal.

Ignoring the signal if it is too late

3.6.5 Additional Functionality of `std::jthread`

A `std::jthread` is a `std::thread` with the additional functionality of cooperative interruption. To support this functionality it has a `std::stop_source`.

The member functions of `std::jthread jthr` for stop-token handling

Member Function	Description
`jthr.get_stop_source()`	Returns a `std::stop_source` object associated with the shared stop state.
`jthr.get_stop_token()`	Returns a `std::stop_token` object associated with the shared stop state.
`jthr.request_stop()`	Requests execution stop via the shared stop state. Returns `true` if the stop request was successful.

The section `std::jthread` provides the details about cooperative interruption of a `std::jthread`.

3.6.6 New `wait` Overloads for the `condition_variable_any`

`std::condition_variable_any` is a generalization of `std::condition_variable`[15]. `std::condition_-variable` requires a `std::unique_lock<std::mutex>`, but `std::condition_variable_any` can operate on any lock `lo`, supporting `lo.lock()` and `lo.unlock`.

The three wait variations to `wait`, `wait_for`, and `wait_until` of the `std::condition_variable_any` get new overloads. They take a `std::stop_token`.

Three new wait overloads

```
1   template <class Predicate>
2   bool wait(Lock& lock,
3             stop_token stoken,
4             Predicate pred);
5
6   template <class Rep, class Period, class Predicate>
7   bool wait_for(Lock& lock,
8                 stop_token stoken,
9                 const chrono::duration<Rep, Period>& rel_time,
10                Predicate pred);
11
12  template <class Clock, class Duration, class Predicate>
13  bool wait_until(Lock& lock,
14                  stop_token stoken,
15                  const chrono::time_point<Clock, Duration>& abs_time,
16                  Predicate pred);
```

These new overloads require a predicate. The presented versions ensure that the threads are notified if a stop request for the passed `std::stop_token` `stoken` is signaled. The functions return a boolean that indicates whether the predicate evaluates to `true`. This returned boolean is independent of whether a stop was requested or whether the timeout was triggered. The three overloads are equivalent to the following expressions:

---

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

Equivalent expression for the three overloads

```cpp
// wait in lines 1 - 4
while (!stoken.stop_requested()) {
    if (pred()) return true;
    wait(lock);
}
return pred();

// wait_for in lines 6 - 10
return wait_until(lock,
                  std::move(stoken),
                  chrono::steady_clock::now() + rel_time,
                  std::move(pred)
                  );

// wait_until in lines 12 - 16
while (!stoken.stop_requested()) {
    if (pred()) return true;
    if (wait_until(lock, timeout_time) == std::cv_status::timeout) return pred();
}
return pred();
```

After the wait calls, you can check if a stop request happened.

Handle interrupts with `wait`

```cpp
cv.wait(lock, stoken, predicate);
if (stoken.stop_requested()){
    // interrupt occurred
}
```

The following example shows the use of a condition variable with a stop request.

Use of condition variable with a stop request

```cpp
1  // conditionVariableAny.cpp
2
3  #include <condition_variable>
4  #include <thread>
5  #include <iostream>
6  #include <chrono>
7  #include <mutex>
8  #include <thread>
9
10 using namespace std::literals;
```

```
11
12   std::mutex mut;
13   std::condition_variable_any condVar;
14
15   bool dataReady;
16
17   void receiver(std::stop_token stopToken) {
18
19       std::cout << "Waiting" << '\n';
20
21       std::unique_lock<std::mutex> lck(mut);
22       bool ret = condVar.wait(lck, stopToken, []{ return dataReady; });
23       if (ret) {
24           std::cout << "Notification received: " << '\n';
25       }
26       else{
27            std::cout << "Stop request received" << '\n';
28       }
29   }
30
31   void sender() {
32
33       std::this_thread::sleep_for(5ms);
34       {
35           std::lock_guard<std::mutex> lck(mut);
36           dataReady = true;
37           std::cout << "Send notification"  << '\n';
38       }
39       condVar.notify_one();
40
41   }
42
43   int main(){
44
45       std::cout << '\n';
46
47       std::jthread t1(receiver);
48       std::jthread t2(sender);
49
50       t1.request_stop();
51
52       t1.join();
53       t2.join();
54
55       std::cout << '\n';
```

```
56
57   }
```

The `receiver` thread (lines 17 - 29) is waiting for the notification of the `sender` thread (lines 31 - 41). Before the `sender` thread sends its notification in line 39, the `main` thread triggered a stop request in line 50. The output of the program shows that the stop request happened before the notification.

```
Waiting
Stop request received
Send notification
```

Sending a stop request to a condition variable

3.7 Semaphores (C++20)

Semaphores are a synchronization mechanism used to control concurrent access to a shared resource. A counting semaphore is a semaphore that has a counter that is bigger than zero. The constructor initializes the counter. Acquiring the semaphore decreases the counter, and releasing the semaphore increases the counter. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

🔑 Edsger W. Dijkstra invented semaphores

The Dutch computer scientist Edsger W. Dijkstra[16] presented in 1965 the concept of a semaphore. A semaphore is a data structure with a queue and a counter. The counter is initialized to a value equal to or greater than zero. It supports the two operations `wait` and `signal`. Operation `wait` acquires the semaphore and decreases the counter. It blocks the thread from acquiring the semaphore if the counter is zero. Operation `signal` releases the semaphore and increases the counter. Blocked threads are added to the queue to avoid starvation[17].

Originally, a semaphore was a railway signal.
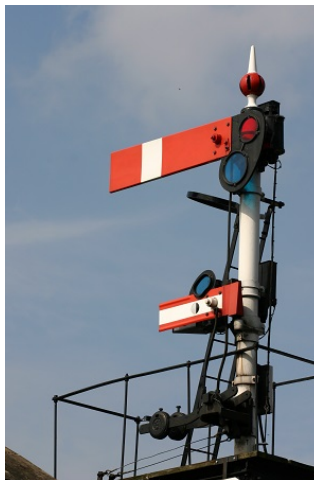
Semaphore

The original uploader was AmosWolfe at English Wikipedia. - Transferred from en.wikipedia to Commons., CC BY 2.0,[18]

C++20 supports a `std::binary_semaphore`, which is an alias for a `std::counting_semaphore<1>`. In this

---

[16]https://en.wikipedia.org/wiki/Edsger_W._Dijkstra
[17]https://en.wikipedia.org/wiki/Starvation_(computer_science)
[18]https://commons.wikimedia.org/w/index.php?curid=1972304

case, the least maximal value is 1. `std::binary_semaphores` can be used to implement locks[19].

```
using binary_semaphore = std::counting_semaphore<1>;
```

In contrast to a `std::mutex`, a `std::counting_semaphore` is not bound to a thread. This means that the acquire and release of a semaphore call can happen on different threads. The following table presents the interface of a `std::counting_semaphore`.

Member functions of a `std::counting_semaphore sem`

Member function	Description
`std::semaphore sem{num}`	Creates a semaphore with the counter `num`.
`sem.max()` (static)	Returns the maximum value of the `counter`.
`sem.release(upd = 1)`	Increases `counter` by `upd` and subsequently unblocks threads acquiring the semaphore `sem`.
`sem.acquire()`	Decrements the `counter` by 1 or blocks until the `counter` is greater than 0.
`sem.try_acquire()`	Tries to decrement the `counter` by 1 if it is greater than 0.
`sem.try_acquire_for(relTime)`	Tries to decrement the `counter` by 1 or blocks for at most `relTime` if the counter is 0.
`sem.try_acquire_until(absTime)`	Tries to decrement the `counter` by 1 or blocks at most until `absTime` if the counter is 0.

The constructor call `std::counting_semaphore<10> sem(5)` creates a semaphore `sem` with an at least maximal value of 10 and a counter of 5. The call `sem.max()` returns the maximum possible value of the internal counter. The following realations must hold for `upd` in `sem.release(upd = 1)`: `update >= 0` and `update + counter <= sem.max()`. `sem.try_aquire_for(relTime)` needs a time duration; the member function `sem.try_acquire_until(absTime)` needs a time point. The three calls `sem.try_acquire`, `sem.try_acquire_for`, and `sem.try_acquire_until` return a boolean indicating the success of the calls.

Semaphores are typically used in sender-receiver workflows. For example, initializing the semaphore `sem` with 0 will block the receiver's `sem.acquire()` call until the sender calls `sem.release()`. Consequently, the receiver waits for the notification of the sender. One-time synchronization of threads can easily be implemented using semaphores.

---

[19]https://en.cppreference.com/w/cpp/named_req/BasicLockable
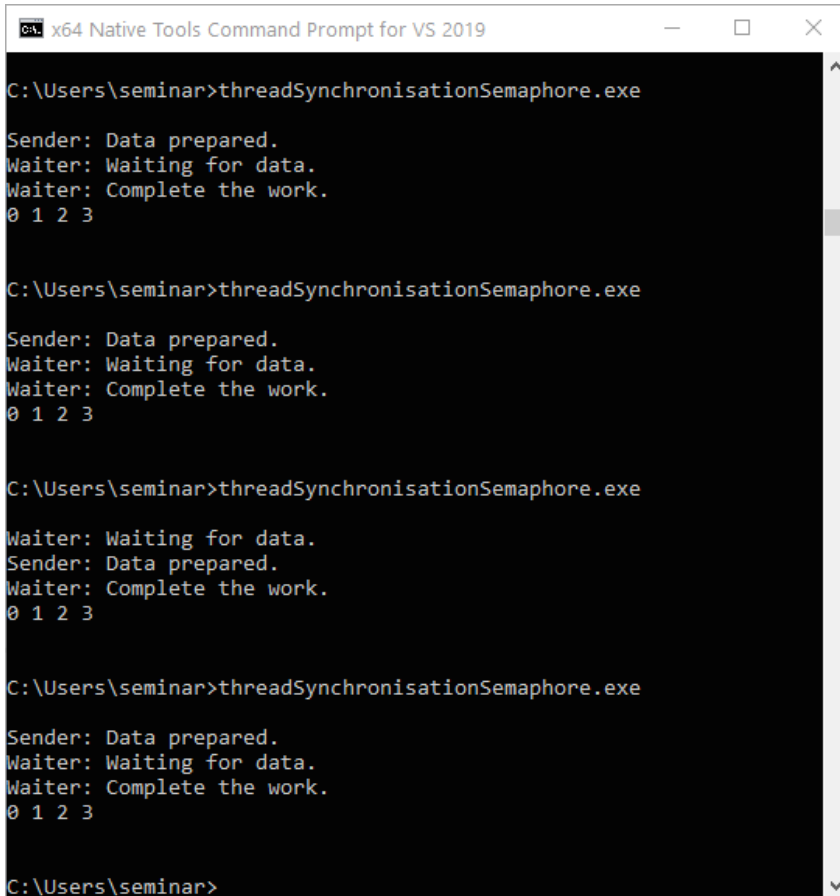
Thread synchronization with a `std::counting_semaphore`

```cpp
1    // threadSynchronizationSemaphore.cpp
2
3    #include <iostream>
4    #include <semaphore>
5    #include <thread>
6    #include <vector>
7
8    std::vector<int> myVec{};
9
10   std::counting_semaphore<1> prepareSignal(0);
11
12   void prepareWork() {
13
14       myVec.insert(myVec.end(), {0, 1, 0, 3});
15       std::cout << "Sender: Data prepared."  << '\n';
16       prepareSignal.release();
17   }
18
19   void completeWork() {
20
21       std::cout << "Waiter: Waiting for data." << '\n';
22       prepareSignal.acquire();
23       myVec[2] = 2;
24       std::cout << "Waiter: Complete the work." << '\n';
25       for (auto i: myVec) std::cout << i << " ";
26       std::cout << '\n';
27
28   }
29
30   int main() {
31
32       std::cout << '\n';
33
34       std::thread t1(prepareWork);
35       std::thread t2(completeWork);
36
37       t1.join();
38       t2.join();
39
40       std::cout << '\n';
41
42   }
```

The `std::counting_semaphore prepareSignal` (line 10) can have the values 0 and 1. In the concrete example, it's initialized with 0 (line 10). This means, that the call `prepareSignal.release()` sets the value to 1 (line 16) and unblocks the call `prepareSignal.acquire()` (line 22).

Thread synchronization with semaphores

3.8 Latches and Barriers (C++20)

Latches and barriers are coordination types that enable some threads to block until a counter becomes zero. At first, don't confuse the new barriers with memory barriers, also known as fences. In C++20 we get latches and barriers in two variations: `std::latch`, and `std::barrier`. Concurrent invocations of the member functions of a `std::latch` or a `std::barrier` produce no data race.

First, there are two questions:

1. What are the differences between these two mechanisms to coordinate threads? You can use a `std::latch` only once, but you can use a `std::barrier` more than once. A `std::latch` helps to manage one task by multiple threads. A `std::barrier` helps to manage repeated tasks by multiple threads. Additionally, a `std::barrier` enables you to execute a function in the so-called completion step. The completion step is the state when the counter becomes zero.

2. What use cases do latches and barriers support that cannot be done in C++11 and C++14 with futures, threads, or condition variables combined with locks? Latches and barriers address no new use cases, but they are a lot easier to use. They are also more performant because they often use a lock-free mechanism internally.

3.8.1 `std::latch`

Now, let us have a closer look at the interface of a `std::latch`.

Member functions of a `std::latch` `lat`

Member function	Description
`std::latch lat{cnt}`	Creates a `std::latch` with counter `cnt`.
`lat.count_down(upd = 1)`	Atomically decrements the counter by `upd` without blocking the caller.
`lat.try_wait()`	Returns `true` if `counter == 0`.
`lat.wait()`	Returns immediately if `counter == 0`. If not blocks until `counter == 0`.
`lat.arrive_and_wait(upd = 1)`	Equivalent to `count_down(upd); wait();`.
`std::latch::max`	Returns the maximum value of the counter supported by the implementation

The default value for `upd` is 1. When `upd` is greater than the counter or negative, the behavior is undefined. The call `lat.try_wait()` never actually waits, as its name suggests.

The following program `bossWorkers.cpp` uses two `std::latch` to build a boss-workers workflow. I synchronized the output to `std::cout` using the function `synchronizedOut` (line 13). This synchronization

makes it easier to follow the workflow.

A boss-worker workflow using two `std::latch`

```cpp
1    // bossWorkers.cpp
2
3    #include <iostream>
4    #include <mutex>
5    #include <latch>
6    #include <thread>
7
8    std::latch workDone(6);
9    std::latch goHome(1);
10
11   std::mutex coutMutex;
12
13   void synchronizedOut(const std::string s) {
14       std::lock_guard<std::mutex> lo(coutMutex);
15       std::cout << s;
16   }
17
18   class Worker {
19    public:
20        Worker(std::string n): name(n) { };
21
22        void operator() (){
23            // notify the boss when work is done
24            synchronizedOut(name + ": " + "Work done!\n");
25            workDone.count_down();
26
27            // waiting before going home
28            goHome.wait();
29            synchronizedOut(name + ": " + "Good bye!\n");
30        }
31    private:
32        std::string name;
33   };
34
35   int main() {
36
37       std::cout << '\n';
38
39       std::cout << "BOSS: START WORKING! " << '\n';
40
41       Worker herb("  Herb");
42       std::thread herbWork(herb);
```

```
43
44      Worker scott("     Scott");
45      std::thread scottWork(scott);
46
47      Worker bjarne("      Bjarne");
48      std::thread bjarneWork(bjarne);
49
50      Worker andrei("        Andrei");
51      std::thread andreiWork(andrei);
52
53      Worker andrew("          Andrew");
54      std::thread andrewWork(andrew);
55
56      Worker david("            David");
57      std::thread davidWork(david);
58
59      workDone.wait();
60
61      std::cout << '\n';
62
63      goHome.count_down();
64
65      std::cout << "BOSS: GO HOME!" << '\n';
66
67      herbWork.join();
68      scottWork.join();
69      bjarneWork.join();
70      andreiWork.join();
71      andrewWork.join();
72      davidWork.join();
73
74  }
```

The idea of the workflow is straightforward. The six workers herb, scott, bjarne, andrei, andrew, and david (lines 41 - 57) have to fulfill their job. When each has finished his job, it counts down the std::latch workDone (line 25). The boss (main-thread) is blocked in line 59 until the counter becomes 0. When the counter is 0, the boss uses the second std::latch goHome to signal its workers to go home. In this case, the initial counter is 1 (line 9). The call goHome.wait() blocks until the counter becomes 0.

A boss-worker workflow using two `std::latch`

When you think about this workflow, you may notice that it can be done without a boss. Here it is.

A worker's workflow using a `std::latch`

```cpp
// workers.cpp

#include <iostream>
#include <barrier>
#include <mutex>
#include <thread>

std::latch workDone(6);
std::mutex coutMutex;

void synchronizedOut(const std::string& s) {
    std::lock_guard<std::mutex> lo(coutMutex);
    std::cout << s;
}

class Worker {
 public:
    Worker(std::string n): name(n) { };
```
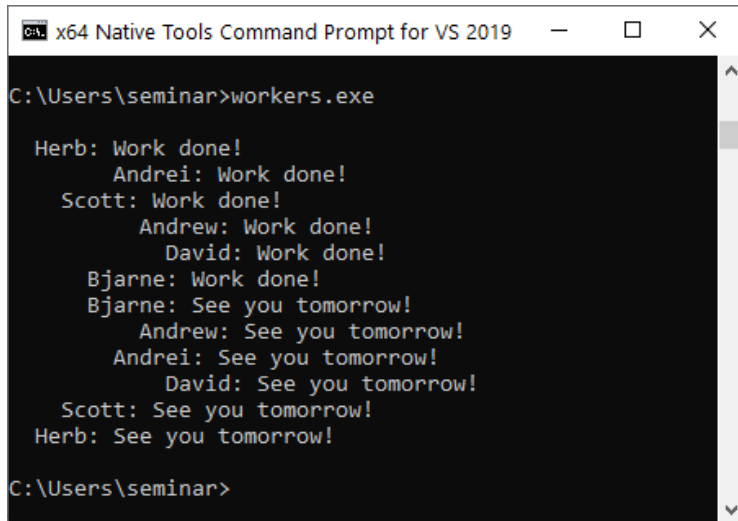
```
19
20        void operator() () {
21            synchronizedOut(name + ": " + "Work done!\n");
22            workDone.arrive_and_wait();  // wait until all work is done
23            synchronizedOut(name + ": " + "See you tomorrow!\n");
24        }
25    private:
26        std::string name;
27    };
28
29    int main() {
30
31        std::cout << '\n';
32
33        Worker herb("  Herb");
34        std::thread herbWork(herb);
35
36        Worker scott("    Scott");
37        std::thread scottWork(scott);
38
39        Worker bjarne("      Bjarne");
40        std::thread bjarneWork(bjarne);
41
42        Worker andrei("        Andrei");
43        std::thread andreiWork(andrei);
44
45        Worker andrew("          Andrew");
46        std::thread andrewWork(andrew);
47
48        Worker david("            David");
49        std::thread davidWork(david);
50
51        herbWork.join();
52        scottWork.join();
53        bjarneWork.join();
54        andreiWork.join();
55        andrewWork.join();
56        davidWork.join();
57
58    }
```

There is not much to add to this simplified workflow. The call `wordDone.arrive_and_wait()` (line 22) is equivalent to the calls `count_down(upd); wait();`. This means the workers coordinate themselves, and the boss is no longer necessary, as was the case in the previous program `bossWorkers.cpp`.

A workers workflow using a `std::latch`

A `std::barrier` is similar to a `std::latch`.

3.8.2 `std::barrier`

There are two differences between a `std::latch` and a `std::barrier`. First, you can use a `std::barrier` more than once, and second, you can adjust the counter for the next phase. The counter is set in the constructor of `std::barrier bar`. Calling `bar.arrive()`, `bar.arrive_and_wait()`, and `bar.arrive_and_-drop()` decrements the counter in the current phase. Additionally, `bar.arrive_and_drop()` decrements the counter for the next phase. Immediately after the current phase is finished and the counter becomes zero, the so-called completion step starts. In this completion step, a callable is invoked. The `std::barrier` gets its callable in its constructor.

The completion step performs the following steps:

1. All threads are blocked.
2. An arbitrary thread is unblocked and executes the callable. The callable must not throw and has to be `noexcept`.
3. If the completion step is done, all threads are unblocked.

Member functions of a `std::barrier bar`

Member function	Description
`std::barrier bar{cnt}`	Creates a `std::batcj` with counter `cnt`.
`std::barrier bar{cnt, call}`	Creates a `std::barrier` with counter `cnt` and callable `call`.
`bar.arrive(upd)`	Atomically decrements counter by `upd`.
`bar.wait()`	Blocks at the synchronization point until the completion step is done.
`bar.arrive_and_wait()`	Equivalent to `wait(arrive())`
`bar.arrive_and_drop()`	Decrements the counter for the current and the subsequent phase by one.
`std::barrier::max`	Maximum value supported by the implementation

The call `bar.arrive_and_drop()` means essentially that the counter is decremented by one for the next phase.

The program `fullTimePartTimeWorkers.cpp` halves the number of workers in the second phase.

Full-time and part-time workers

```cpp
// fullTimePartTimeWorkers.cpp

#include <iostream>
#include <barrier>
#include <mutex>
#include <string>
#include <thread>

std::barrier workDone(6);
std::mutex coutMutex;

void synchronizedOut(const std::string& s) noexcept {
    std::lock_guard<std::mutex> lo(coutMutex);
    std::cout << s;
}

class FullTimeWorker {
 public:
    FullTimeWorker(std::string n): name(n) { };

    void operator() () {
        synchronizedOut(name + ": " + "Morning work done!\n");
```
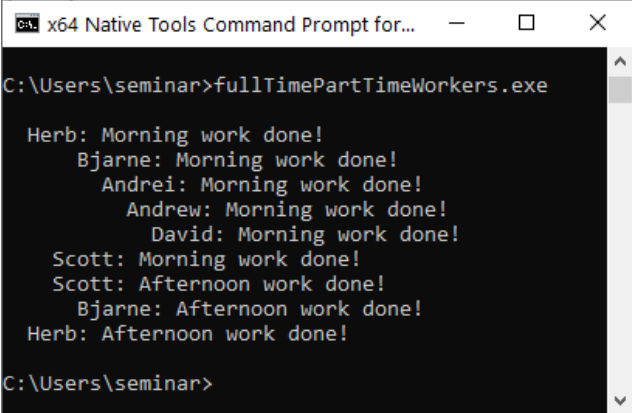
```
23          workDone.arrive_and_wait();  // Wait until morning work is done
24          synchronizedOut(name + ": " + "Afternoon work done!\n");
25          workDone.arrive_and_wait();  // Wait until afternoon work is done
26
27      }
28  private:
29      std::string name;
30  };
31
32  class PartTimeWorker {
33  public:
34      PartTimeWorker(std::string n): name(n) { };
35
36      void operator() () {
37          synchronizedOut(name + ": " + "Morning work done!\n");
38          workDone.arrive_and_drop();  // Wait until morning work is done
39      }
40  private:
41      std::string name;
42  };
43
44  int main() {
45
46      std::cout << '\n';
47
48      FullTimeWorker herb("  Herb");
49      std::thread herbWork(herb);
50
51      FullTimeWorker scott("    Scott");
52      std::thread scottWork(scott);
53
54      FullTimeWorker bjarne("      Bjarne");
55      std::thread bjarneWork(bjarne);
56
57      PartTimeWorker andrei("        Andrei");
58      std::thread andreiWork(andrei);
59
60      PartTimeWorker andrew("          Andrew");
61      std::thread andrewWork(andrew);
62
63      PartTimeWorker david("            David");
64      std::thread davidWork(david);
65
66      herbWork.join();
67      scottWork.join();
```

```
68        bjarneWork.join();
69        andreiWork.join();
70        andrewWork.join();
71        davidWork.join();
72
73   }
```

This workflow consists of two kinds of workers: full-time workers (line 17) and part-time workers (line 32). The part-time worker works in the morning, the full-time worker in the morning and the afternoon. Consequently, the full-time workers call workDone.arrive_and_wait() (lines 23 and 25) two times. On the contrary, the part-time workers call workDone.arrive_and_drop() (line 38) only once. This workDone.arrive_and_drop() call causes the part-time worker to skip the afternoon work. Accordingly, the counter has in the first phase (morning) the value 6, and in the second phase (afternoon) the value 3.

Full-time and part-time workers

3.9 Tasks

In addition to threads, C++ has tasks to perform work asynchronously. Tasks need the `<future>` header. A task is parameterised with a work package and consists of 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. It is special that the future can pick up the result later; 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 between communication endpoints

Tasks behave like data channels between communication endpoints. One endpoint of the data channel is called the promise. The other endpoint of the data channel is called the future. These endpoints can exist in the same or different threads. The promise puts its result in the data channel. The future picks it up later.
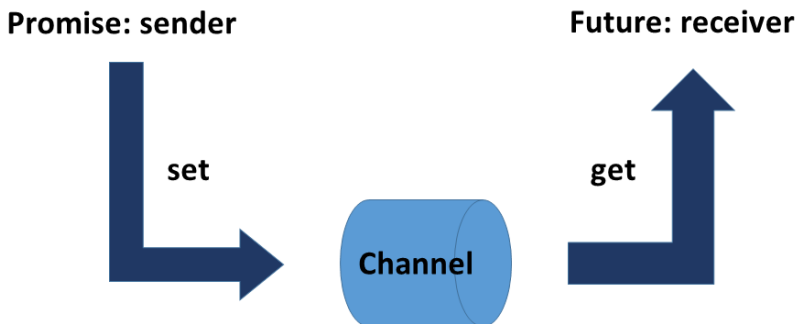
Tasks as data channels between communication endpoints

3.9.1 Tasks versus Threads

Tasks are very different threads.

`std:async` versus threads
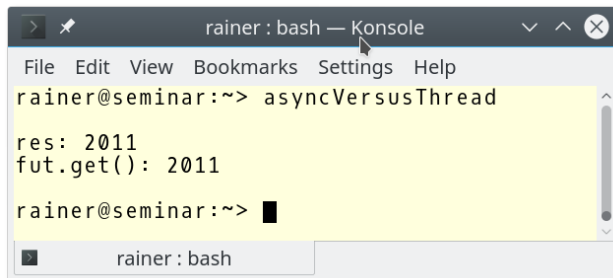
```cpp
1    // asyncVersusThread.cpp
2
3    #include <future>
4    #include <thread>
5    #include <iostream>
6
7    int main(){
8
9        std::cout << '\n';
10
11       int res;
12       std::thread t([&]{ res = 2000 + 11; });
13       t.join();
14       std::cout << "res: " << res << '\n';
15
16       auto fut= std::async([]{ return 2000 + 11; });
17       std::cout << "fut.get(): " << fut.get() << '\n';
18
19       std::cout << '\n';
20
21   }
```

The child thread `t` and the asynchronous function call `std::async` calculate both the sum of `2000` and `11`. The creator thread gets the result from its child thread `t` via the shared variable `res` and displays it in line 14. The call `std::async` in line 16 creates the data channel between the sender (promise) and the receiver (future). The future asks the data channel with `fut.get()` (line 17) for the calculation. This `fut.get` call is blocking.

Here is the output of the program.

Tasks versus threads

Based on this program, I want to emphasize the differences between threads and tasks explicitly.

Tasks versus threads

Criteria	Threads	Tasks
Participants	creator and child thread	promise and future
Communication	shared variable	communication channel
Thread creation	obligatory	optional
Synchronization	via `join()` (waits)	`get` call blocks
Exception in child thread	child and creator threads terminates	return value of the promise
Kinds of communication	values	values, notifications, and exceptions

Threads need the `<thread>` header; tasks the `<future>` header.

Communication between the creator thread and the created thread requires the use of a shared variable. The task communicates via its implicitly protected data channel; therefore, a task must not use a protection mechanism like a mutex.

While you can *misuse* a shared mutable variable to communicate between the child and its creator, a task's communication is more explicit. The future can request the task's result only once (by calling `fut.get()`). Calling it more than once results in undefined behavior. This is not true for a `std::shared_-future`, which can be queried multiple times.

The creator thread waits for its child with the call to `join`. The future `fut` uses the `fut.get()` call which blocks until the result is available.

If an exception is thrown in the created thread, the created thread terminates and so the creator and the whole process. In contrast, the promise can send the exception 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 just a notification. You can use them as a safe replacement for a condition variable.

`std::async` is the easiest way to create a future.

3.9.2 `std::async`

`std::async` behaves like an asynchronous function call. This function call takes a callable together with its arguments. `std::async` is a variadic template and can, therefore, take an arbitrary number of arguments. The call to `std::async` returns a future object `fut`. That's your handle on getting the result via `fut.get()`.

> ## `std::async` should be your first choice
>
> The C++ runtime decides if `std::async` is executed in a separate thread or not. The decision of the C++ runtime may depend on the number of CPU cores available, the utilization of your system, or the size of your work package. By using `std::async` you only specify the task that should run. The C++ runtime automatically manages the creation and also the lifetime of the thread.

Optionally you can specify a start policy for `std::async`.

3.9.2.1 The Start Policy

With the start policy, you can explicitly specify whether the asynchronous call should be executed in the same thread (`std::launch::deferred`) or in another thread (`std::launch::async`).

> ## Eager versus lazy evaluation
>
> Eager and lazy evaluation are two orthogonal strategies to calculate the result of an expression. In the case of eager evaluation[20], the expression is evaluated immediately - in the case of lazy evaluation[21] the expression is only be evaluated if needed. Eager evaluation is often called greedy evaluation, and lazy evaluation is often called call-by-need. With lazy evaluation, you save time and compute power because there is no evaluation on suspicion.

It is special about the call `auto fut = std::async(std::launch::deferred, ... )` that the promise is not be executed immediately. The call `fut.get()` starts the promise lazily. This means that the promise only runs if the future asks via `fut.get()` for the result.

Eager and lazy evaluation of a future

```cpp
// asyncLazy.cpp

#include <chrono>
#include <future>
#include <iostream>

int main(){

  std::cout << '\n';

  auto begin= std::chrono::system_clock::now();

```

---

[20]https://en.wikipedia.org/wiki/Eager_evaluation
[21]https://en.wikipedia.org/wiki/Lazy_evaluation

```
13    auto asyncLazy=std::async(std::launch::deferred,
14                              []{ return std::chrono::system_clock::now(); });
15
16    auto asyncEager=std::async(std::launch::async,
17                              []{ return std::chrono::system_clock::now(); });
18
19    std::this_thread::sleep_for(std::chrono::seconds(1));
20
21    auto lazyStart= asyncLazy.get() - begin;
22    auto eagerStart= asyncEager.get() - begin;
23
24    auto lazyDuration= std::chrono::duration<double>(lazyStart).count();
25    auto eagerDuration=  std::chrono::duration<double>(eagerStart).count();
26
27    std::cout << "asyncLazy evaluated after : " << lazyDuration
28              << " seconds." << '\n';
29    std::cout << "asyncEager evaluated after: " << eagerDuration
30              << " seconds." << '\n';
31
32    std::cout << '\n';
33
34 }
```

Both `std::async` calls (lines 13 and 16) return the current time point. However, the first call is lazy while the second eager. The short sleep of one second in line 19 makes that obvious. The call `asyncLazy.get()` in line 21 triggers the execution of the promise in line 13 - the result is available after a short nap of one second (line 19). This is not true for `asyncEager`, which gets the result from the immediately executed work package.

Here are the numbers.

Lazy versus eager evaluation

You do not have to bind a future to a variable.

3.9.2.2 Fire and Forget

Fire and forget futures are special futures. They execute just in place because their future is not bound to a variable. It is necessary for a fire-and-forget future that the promise runs in a separate thread to immediately start its work. This is done by the `std::launch::async` policy.

Let's compare an ordinary future with a fire and forget future.

```cpp
auto fut= std::async([]{ return 2011; });
std::cout << fut.get() << '\n';

std::async(std::launch::async,
          []{ std::cout << "fire and forget" << '\n'; });
```

Fire and forget futures look very promising but have a big drawback. A future that is created by `std::async` waits on its destructor, until its promise is done. In this context, waiting is not very different from blocking. The future blocks the progress of the program in its destructor. This becomes more evident when you use fire and forget futures. What seems to be concurrent runs sequentially.

Fire and forget futures

```cpp
1   // fireAndForgetFutures.cpp
2
3   #include <chrono>
4   #include <future>
5   #include <iostream>
6   #include <thread>
7
8   int main(){
9
10    std::cout << '\n';
11
12    std::async(std::launch::async, []{
13        std::this_thread::sleep_for(std::chrono::seconds(2));
14        std::cout << "first thread" << '\n';
15      });
16
17    std::async(std::launch::async, []{
18        std::this_thread::sleep_for(std::chrono::seconds(1));
19        std::cout << "second thread" << '\n';}
20      );
21
22    std::cout << "main thread" << '\n';
23
24    std::cout << '\n';
```
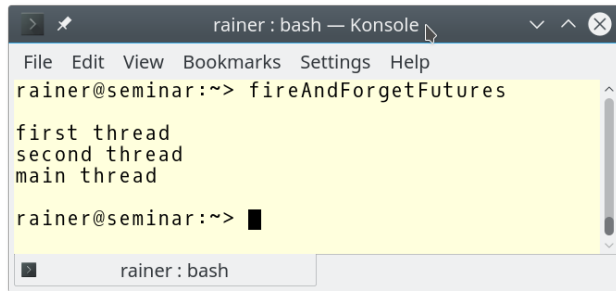
```
25
26   }
```

The program executes two promises in their threads. The resulting futures are fire and forget futures. These futures block in their destructors until the associated promise is done. The result is that the promises are executed in the sequence which you find in the source code. The execution sequence is independent of the execution time. This is what you see in the output of the program.

Fire and forget futures

`std::async` is a convenient mechanism used to distribute a bigger compute job on more shoulders.

3.9.2.3 Concurrent Calculation

The calculation of the scalar product can be spread across four asynchronous function calls.

Dot product with four futures

```cpp
1    // dotProductAsync.cpp
2
3    #include <iostream>
4    #include <future>
5    #include <random>
6    #include <vector>
7    #include <numeric>
8
9    using namespace std;
10
11   static const int NUM= 100000000;
12
13   long long getDotProduct(vector<int>& v, vector<int>& w){
14
15     auto vSize = v.size();
16
17     auto future1 = async([&]{
```
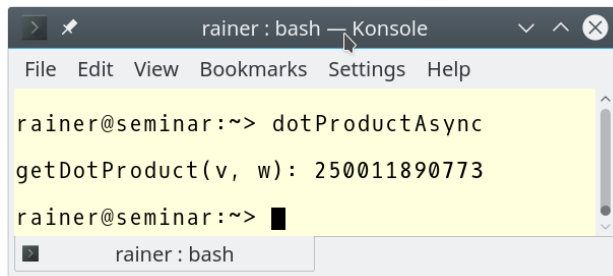
```
18      return inner_product(&v[0], &v[vSize/4], &w[0], 0LL);
19    });
20
21    auto future2 = async([&]{
22      return inner_product(&v[vSize/4], &v[vSize/2], &w[vSize/4], 0LL);
23    });
24
25    auto future3 = async([&]{
26      return inner_product(&v[vSize/2], &v[vSize* 3/4], &w[vSize/2], 0LL);
27    });
28
29    auto future4 = async([&]{
30      return inner_product(&v[vSize * 3/4], &v[vSize], &w[vSize * 3/4], 0LL);
31    });
32
33    return future1.get() + future2.get() + future3.get() + future4.get();
34  }
35
36
37  int main(){
38
39    cout << '\n';
40
41    random_device seed;
42
43    // generator
44    mt19937 engine(seed());
45
46    // distribution
47    uniform_int_distribution<int> dist(0, 100);
48
49    // fill the vectors
50    vector<int> v, w;
51    v.reserve(NUM);
52    w.reserve(NUM);
53    for (int i=0; i< NUM; ++i){
54      v.push_back(dist(engine));
55      w.push_back(dist(engine));
56    }
57
58    cout << "getDotProduct(v, w): " << getDotProduct(v, w) << '\n';
59
60    cout << '\n';
61
62  }
```

The program uses the functionality of the random library that is part of C++11. The two vectors v and w are created and filled with random numbers (lines 50 - 56). Each of the vectors gets (lines 53 - 56) one hundred million elements. `dist(engine)` in lines 54 and 55 generates the random numbers, which are uniformly distributed in the range 0 to 100. The calculation of the scalar product takes place in `getDotProduct` (lines 13 - 34). The standard algorithm `std::inner_product` is executed asynchronously four times: one asynchronous invocation for each quarter of the vectors' length. The return statement sums up the results of the futures.

Scalar product with four asynchronous function calls

`std::packaged_task` is also usually used to perform a concurrent computation.

3.9.3 `std::packaged_task`

`std::packaged_task pack` is a wrapper for a callable in order to be invoked asynchronously. By calling `pack.get_future()` you get the associated future. Invoking the call operator on pack (`pack()`) executes the `std::packaged_task` and, therefore, executes the callable.

Dealing with `std::packaged_task` usually consists of four steps:

I. Wrap your work:

```cpp
std::packaged_task<int(int, int)> sumTask([](int a, int b){ return a + b; });
```

II. Create a future:

```cpp
std::future<int> sumResult= sumTask.get_future();
```

III. Perform the calculation:

```cpp
sumTask(2000, 11);
```

IV. Query the result:

```
        sumResult.get();
```

Here is an example showing the four steps.

Concurrency with `std::packaged_task`

```cpp
1   // packagedTask.cpp
2
3   #include <utility>
4   #include <future>
5   #include <iostream>
6   #include <thread>
7   #include <deque>
8
9   class SumUp{
10    public:
11      int operator()(int beg, int end){
12        long long int sum{0};
13        for (int i = beg; i < end; ++i ) sum += i;
14        return sum;
15      }
16  };
17
18  int main(){
19
20    std::cout << '\n';
21
22    SumUp sumUp1;
23    SumUp sumUp2;
24    SumUp sumUp3;
25    SumUp sumUp4;
26
27    // wrap the tasks
28    std::packaged_task<int(int, int)> sumTask1(sumUp1);
29    std::packaged_task<int(int, int)> sumTask2(sumUp2);
30    std::packaged_task<int(int, int)> sumTask3(sumUp3);
31    std::packaged_task<int(int, int)> sumTask4(sumUp4);
32
33    // create the futures
34    std::future<int> sumResult1 = sumTask1.get_future();
35    std::future<int> sumResult2 = sumTask2.get_future();
36    std::future<int> sumResult3 = sumTask3.get_future();
37    std::future<int> sumResult4 = sumTask4.get_future();
38
39    // push the tasks on the container
40    std::deque<std::packaged_task<int(int,int)>> allTasks;
```

```
41    allTasks.push_back(std::move(sumTask1));
42    allTasks.push_back(std::move(sumTask2));
43    allTasks.push_back(std::move(sumTask3));
44    allTasks.push_back(std::move(sumTask4));
45
46    int begin{1};
47    int increment{2500};
48    int end = begin + increment;
49
50    // perform each calculation in a separate thread
51    while (not allTasks.empty()){
52      std::packaged_task<int(int, int)> myTask = std::move(allTasks.front());
53      allTasks.pop_front();
54      std::thread sumThread(std::move(myTask), begin, end);
55      begin = end;
56      end += increment;
57      sumThread.detach();
58    }
59
60    // pick up the results
61    auto sum = sumResult1.get() + sumResult2.get() +
62              sumResult3.get() + sumResult4.get();
63
64    std::cout << "sum of 0 .. 10000 = " << sum << '\n';
65
66    std::cout << '\n';
67
68  }
```

The program's purpose is to calculate the sum of all numbers from 0 to 10000 - with the help of four `std::packaged_task` each running in a separate thread. The associated futures are used, to sum up the final result. Of course, you can use the Gaußschen Summenformel[22]. Strange, I didn't find an English web page.
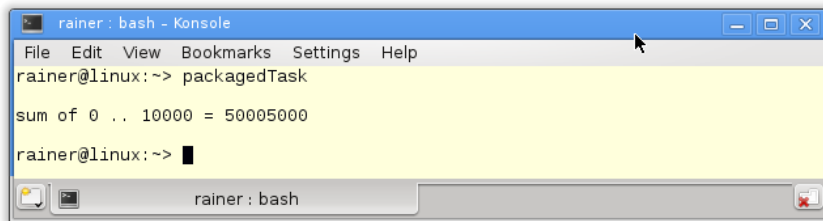
I. **Wrap the tasks**: I pack the work packages in `std::packaged_task` (lines 28 - 31) objects. Work packages are instances of the class SumUp (lines 9 - 16). The work is done in the call operator (lines 11 - 15). The call operator sums up all numbers from beg to end - 1 and returns the sum as a result. `std::packaged_task` in lines 28 - 31 can deal with callables that need two ints and return an int: `int(int, int)`.

II. **Create the futures**: I have to create the future objects with the help of `std::packaged_task` objects. This is done in the lines 34 to 37. The packaged_task is the promise in the communication channel. The type of the future is defined explicitly: `std::future<int> sumResult1 = sumTask1.get_future()`, but the compiler can do that job for me: `auto sumResult1 = sumTask1.get_future()`.

---

[22]https://de.wikipedia.org/wiki/Gau%C3%9Fsche_Summenformel

III. **Perform the calculations**: Now, the calculation takes place. The `packaged_task` are moved onto the `std::deque`[23] (lines 40 - 44). In the while loop, each `packaged_task` (lines 51 - 58) is executed. For doing that, I move the head of the `std::deque` in a `std::packaged_task` (line 52), move the `packaged_task` in a new thread (line 54), and let it run in the background (line 57). `std::packaged_task` objects are not copyable. That's the reason I used the move semantic in lines 52 and 54. This restriction holds for all promises, but also futures and threads. There is one exception to this rule: std::shared_future.

IV. **Pick up the results**: In the final step, I ask all futures for their value and sum them up (line 61).

Here is the overall result of the concurrent computation.

Summation with `std::packaged_task`

The following table shows the interface of an `std::packaged_task pack`.

The member functions of the `std::packaged_task`

Member function	Description
`pack.swap(pack2)`	Swaps the task objects. Same as `std::swap(pack, pack2)`.
`pack.valid()`	Checks if the task object has a valid function.
`pack.get_future()`	Returns the future.
`pack.make_ready_at_thread_exit(ex)`	Executes the function. The result is available if the current thread exits.
`pack.reset()`	Resets the state of the task. Abandons the stored results from previous executions.

A `std::packaged_task` can be in contrast to a `std::async`, or a `std::promise` reset and resused. The following example shows this special use-case on a `std::packaged_task`.

---

[23]http://en.cppreference.com/w/cpp/container/deque

Reuse of a `std::packaged_task`
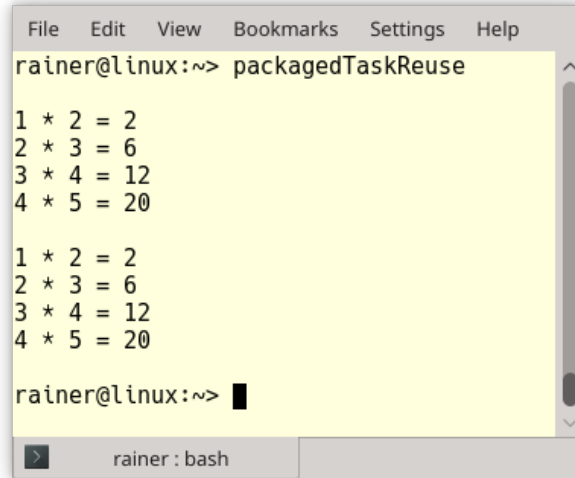
```cpp
// packagedTaskReuse.cpp

#include <functional>
#include <future>
#include <iostream>
#include <utility>
#include <vector>

void calcProducts(std::packaged_task<int(int, int)>& task,
                  const std::vector<std::pair<int, int>>& pairs){
    for (auto& pair: pairs){
        auto fut = task.get_future();
        task(pair.first, pair.second);
        std::cout << pair.first << " * " << pair.second << " = " <<  fut.get()
                  << '\n';
        task.reset();
    }
}

int main(){

    std::cout << '\n';

    std::vector<std::pair<int, int>> allPairs;
    allPairs.push_back(std::make_pair(1, 2));
    allPairs.push_back(std::make_pair(2, 3));
    allPairs.push_back(std::make_pair(3, 4));
    allPairs.push_back(std::make_pair(4, 5));

    std::packaged_task<int(int, int)> task{[](int fir, int sec){
        return fir * sec; }
    };

    calcProducts(task, allPairs);

    std::cout << '\n';

    std::thread t(calcProducts, std::ref(task), allPairs);
    t.join();

    std::cout << '\n';

}
```

The function `calcProducts` (line 9) gets two arguments: the task and the int-pairs vector. It uses the task to calculate for each int-pair the product (line 13). In the end, the task is reset in line 16. `calcProducts` runs in the main-thread (line 34) and a separate thread (line 38). Here is the output of the program.

Reuse of a `std::packaged_task`

3.9.4 `std::promise` **and** `std::future`

The class templates `std::promise` and `std::future` provide you with full control over the task.

Promise and future are a mighty pair. A promise can put a value, an exception, or simply a notification into the shared data channel. One promise can serve many `std::shared_future` futures. With C++23 we may get extended futures that are composable.

Here is an introductory example of the usage of `std::promise` and `std::future`. Both communication endpoints can be moved to separate threads, so the communication takes place between threads.

Usage of `std::promise` and `std::future`

```cpp
// promiseFuture.cpp

#include <future>
#include <iostream>
#include <thread>
#include <utility>

void product(std::promise<int>&& intPromise, int a, int b){
  intPromise.set_value(a*b);
```
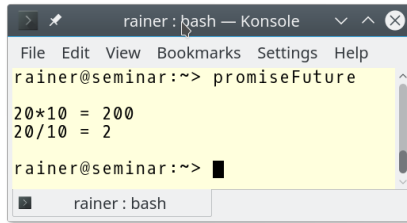
```
10   }
11
12   struct Div{
13
14     void operator() (std::promise<int>&& intPromise, int a, int b) const {
15       intPromise.set_value(a/b);
16     }
17
18   };
19
20   int main(){
21
22     int a = 20;
23     int b = 10;
24
25     std::cout << '\n';
26
27     // define the promises
28     std::promise<int> prodPromise;
29     std::promise<int> divPromise;
30
31     // get the futures
32     std::future<int> prodResult = prodPromise.get_future();
33     std::future<int> divResult = divPromise.get_future();
34
35     // calculate the result in a separate thread
36     std::thread prodThread(product, std::move(prodPromise), a, b);
37     Div div;
38     std::thread divThread(div, std::move(divPromise), a, b);
39
40     // get the result
41     std::cout << "20*10 = " << prodResult.get() << '\n';
42     std::cout << "20/10 = " << divResult.get() << '\n';
43
44     prodThread.join();
45
46     divThread.join();
47
48     std::cout << '\n';
49
50   }
```

Thread `prodThread` (line 36) gets the function `product` (lines 8 -10), the `prodPromise` (line 32) and the numbers `a` and `b`. To understand the arguments of `prodThread`, you have to look at the signature of the

function. `prodThread` needs as its first argument a callable. This is the previously mentioned function `product`. The function `product` requires a promise of the kind rvalue reference (`std::promise<int>&& intPromise`) and two numbers. These are the last three arguments of `prodThread`. `std::move` in line 36 creates an rvalue reference. The rest is a piece of cake. `divThread` (line 38) divides the two numbers `a` and `b`. For its work, it uses the instance `div` of the class `Div` (lines 12 - 18). `div` is an instance of a function object.

The future picks up the results by calling `prodResult.get()` and `divResult.get()`.

Division and multiplication with tasks

3.9.4.1 `std::promise`

`std::promise` enables you to set a value, a notification, or an exception. Besides, the promise can provide its result in a delayed fashion.

The member functions of the `std::promise` prom

Member function	Description
`prom.swap(prom2)`	Swaps the promises. Same as `std::swap(prom, prom2)`.
`prom.get_future()`	Returns the future.
`prom.set_value(val)`	Sets the value.
`prom.set_exception(ex)`	Sets the exception.
`prom.set_value_at_thread_exit(val)`	Stores the value and makes it ready if the promise exits.
`prom.set_exception_at_thread_exit(ex)`	Stores the exception and makes it ready if the promise exits.

If the promise sets the value or the exception more than once, a `std::future_error` exception is thrown.

3.9.4.2 `std::future`

A `std::future` enables you to

- pick up the value from the promise.
- ask the promise if the value is available.
- wait for the notification of the promise. This waiting can be done with a relative time duration or an absolute time point.
- create a shared future (std::shared_future).

The member functions of the future `fut`

Member function	Description
`fut.share()`	Returns a `std::shared_future`. After calling `fut.share()`, `fut.valid()` returns `false`.
`fut.get()`	Returns the result, which can be a value or an exception.
`fut.valid()`	Checks if the shared state is available. After calling `fut.get()` it returns `false`.
`fut.wait()`	Waits for the result.
`fut.wait_for(relTime)`	Waits for the result, but not longer than for a `relTime`. Returns a `std::future_status`.
`fut.wait_until(absTime)`	Waits for the result, but not longer than until `abstime`. Returns a `std::future_status`.

In contrast to the `wait` call, the `wait_for` and `wait_until` variants return the status of the future.

3.9.4.2.1 `std::future_status`

The `wait_for` and `wait_until` calls of the future and a shared future return its state. Three states are possible:

State of a future or a shared future

```
enum class future_status {
    ready,
    timeout,
    deferred
};
```

The following table describe each possible state:

The state of a future or a shared future

State	Description
deferred	The function has not been started.
ready	The result is ready.
timeout	The timeout has expired.

Thanks to the `wait_for` or `wait_until` variants of a future, your future can wait until its associated promise is done.

Waiting for the promise

```cpp
// waitFor.cpp

#include <iostream>
#include <future>
#include <thread>
#include <chrono>

using namespace std::literals::chrono_literals;

void getAnswer(std::promise<int> intPromise){
    std::this_thread::sleep_for(3s);
    intPromise.set_value(42);
}

int main(){

    std::cout << '\n';

    std::promise<int> answerPromise;
    auto fut = answerPromise.get_future();

        std::thread prodThread(getAnswer, std::move(answerPromise));

        std::future_status status{};
        do {
    status = fut.wait_for(0.2s);
                std::cout << "... doing something else" << '\n';
        } while (status != std::future_status::ready);

    std::cout << '\n';

```

```
32        std::cout << "The Answer: " << fut.get() << '\n';

33

34        prodThread.join();

35

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

37

38    }
```

While the future `fut` is waiting for the promise, it can perform *something else*.

Waiting for the promise

If a future `fut` asks for the result more than once, a `std::future_error` exception is thrown.

There is a one-to-one relationship between the promise and the future. In contrast, `std::shared_future` supports a one-to-many relationship between a promise and many futures.

3.9.5 `std::shared_future`

There are two ways to create a `std::shared future`.

1. You can take the future from the Promise `prom` by a `std::shared_future`: `std::shared_-future<int> fut = prom.get_future()`.

2. You can either invoke `fut.share()` on a future `fut`. After the `fut.share()` call `fut.valid()` returns `false`.

A shared future is associated with its promise and can independently ask for the result. A `std::shared future` has the same interface as a `std::future`.

In addition to the `std::future`, a `std::shared_future` enables you to query the promise independently of the other associated futures.

The handling of a `std::shared_future` is special. The following program creates directly a `std::shared_-future`.

Taking a future with a `std::shared_future`

```cpp
1   // sharedFuture.cpp
2
3   #include <future>
4   #include <iostream>
5   #include <thread>
6   #include <utility>
7
8   std::mutex coutMutex;
9
10  struct Div{
11
12    void operator()(std::promise<int>&& intPromise, int a, int b){
13      intPromise.set_value(a/b);
14    }
15
16  };
17
18  struct Requestor{
19
20    void operator ()(std::shared_future<int> shaFut){
21
22      // lock std::cout
23      std::lock_guard<std::mutex> coutGuard(coutMutex);
24
25      // get the thread id
26      std::cout << "threadId(" << std::this_thread::get_id() << "): " ;
27
28      std::cout << "20/10= " << shaFut.get() << '\n';
29
30    }
31
```

```
32    };
33
34    int main(){
35
36      std::cout << '\n';
37
38      // define the promises
39      std::promise<int> divPromise;
40
41      // get the futures
42      std::shared_future<int> divResult = divPromise.get_future();
43
44      // calculate the result in a separat thread
45      Div div;
46      std::thread divThread(div, std::move(divPromise), 20, 10);
47
48      Requestor req;
49      std::thread sharedThread1(req, divResult);
50      std::thread sharedThread2(req, divResult);
51      std::thread sharedThread3(req, divResult);
52      std::thread sharedThread4(req, divResult);
53      std::thread sharedThread5(req, divResult);
54
55      divThread.join();
56
57      sharedThread1.join();
58      sharedThread2.join();
59      sharedThread3.join();
60      sharedThread4.join();
61      sharedThread5.join();
62
63      std::cout << '\n';
64
65    }
```

Both work packages of the promise and the future are function objects in this current example. In line 46 divPromise is moved and executed in thread divThread. Accordingly, std::shared_future's are copied in all five threads (lines 49 - 53). It's important to emphasize it once more. In contrast to a std::future object that can only be moved, you can copy a std::shared_future object.

The main thread waits in lines 57 to 61 for its child threads to finish their jobs and to display their the results.

Taking a `std::shared_future`

As I previously mentioned, you can create a `std::shared_future` from a `std::future` by using its member function `share`.

Creating a `std::shared_future` from a `std::future`

```cpp
// sharedFutureFromFuture.cpp

#include <future>
#include <iostream>
#include <thread>
#include <utility>

std::mutex coutMutex;

struct Div{

  void operator()(std::promise<int>&& intPromise, int a, int b){
    intPromise.set_value(a/b);
  }

};

struct Requestor{

  void operator ()(std::shared_future<int> shaFut){

    // lock std::cout
    std::lock_guard<std::mutex> coutGuard(coutMutex);

    // get the thread id
    std::cout << "threadId(" << std::this_thread::get_id() << "): " ;

```

```
28        std::cout << "20/10= " << shaFut.get() << '\n';

29

30    }

31

32  };

33

34  int main(){

35

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

37

38    // define the promises
39    std::promise<int> divPromise;

40

41    // get the future
42    std::future<int> divResult = divPromise.get_future();

43

44    std::cout << "divResult.valid(): " << divResult.valid() << '\n';

45

46    // calculate the result in a separat thread
47    Div div;
48    std::thread divThread(div, std::move(divPromise), 20, 10);

49

50    std::cout << "divResult.valid(): " << divResult.valid() << '\n';

51

52    std::shared_future<int> sharedResult = divResult.share();

53

54    std::cout << "divResult.valid(): " << divResult.valid() << "\n\n";

55

56    Requestor req;
57    std::thread sharedThread1(req, sharedResult);
58    std::thread sharedThread2(req, sharedResult);
59    std::thread sharedThread3(req, sharedResult);
60    std::thread sharedThread4(req, sharedResult);
61    std::thread sharedThread5(req, sharedResult);

62

63    divThread.join();

64

65    sharedThread1.join();
66    sharedThread2.join();
67    sharedThread3.join();
68    sharedThread4.join();
69    sharedThread5.join();

70

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

72
```
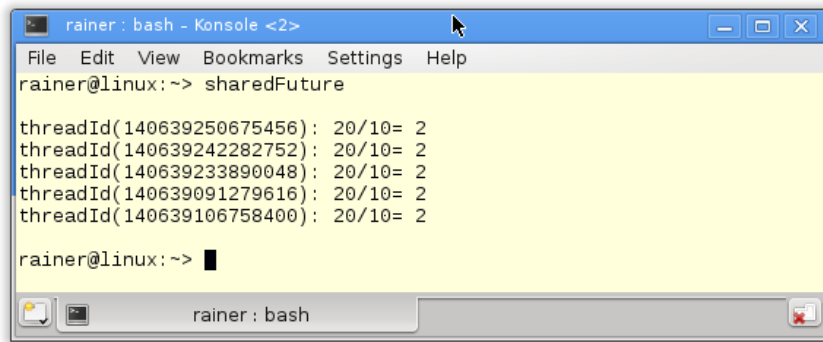
```
73    }
```

The first two calls of `divResult.valid()` on the `std::future` (lines 44 and 50) return `true`. This change after the call `divResult.share()` in line 52 because this call transfers the shared state.

Creating a `std::shared_future`

3.9.6 Exceptions

If the callable used by `std::async` or by `std::packaged_task` throws an error, the exception is store in the shared state. When the future `fut` then calls `fut.get()`, the exception is rethrown, and the code using the future has to handle it.

A `std::promise prom` provides the same facility but has the member function `prom.set_exception(std::current_-exception())` to set the exception as shared state.

Dividing a number by 0 is undefined behavior. The function `executeDivision` displays the result of the calculation or the exception.

Returning an exception

```cpp
1   // promiseFutureException.cpp
2
3   #include <exception>
4   #include <future>
5   #include <iostream>
6   #include <thread>
7   #include <utility>
8
9   struct Div{
```
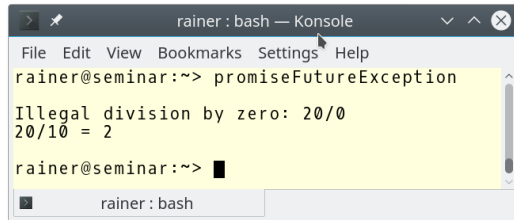
```cpp
10     void operator()(std::promise<int>&& intPromise, int a, int b){
11       try{
12         if ( b==0 ){
13             std::string errMess = std::string("Illegal division by zero: ") +
14                               std::to_string(a) + "/" + std::to_string(b);
15             throw std::runtime_error(errMess);
16         }
17         intPromise.set_value(a/b);
18       }
19       catch (...){
20         intPromise.set_exception(std::current_exception());
21       }
22     }
23   };
24
25   void executeDivision(int nom, int denom){
26     std::promise<int> divPromise;
27     std::future<int> divResult= divPromise.get_future();
28
29     Div div;
30     std::thread divThread(div, std::move(divPromise), nom, denom);
31
32     // get the result or the exception
33     try{
34       std::cout << nom << "/" << denom << " = " << divResult.get() << '\n';
35     }
36     catch (std::runtime_error& e){
37       std::cout << e.what() << '\n';
38     }
39
40     divThread.join();
41   }
42
43   int main(){
44
45     std::cout << '\n';
46
47     executeDivision(20, 0);
48     executeDivision(20, 10);
49
50     std::cout << '\n';
51
52   }
```

The promise deals with the issue that the denominator is 0. If the denominator is 0, it sets the exception as return value: `intPromise.set_exception(std::current_exception())` in line 20. The future has to deal with the exception in its try-catch block (lines 33 - 38).

Here is the output of the program.

Exception handling with `std::promise` and `std::future`

---

std::current_exception and std::make_exception_ptr

`std::current_exception()` captures the current exception object and creates a `std::exception_ptr`. `std::exception_ptr` holds either a copy or a reference to the exception object. If the function is called when no exception is being handled, an empty `std::exception_ptr`[24] is returned.

Instead of retrieving the thrown exception in an try/catch block with `intPromise.set_exception(std::current_exception());`, you can directly call `intPromise.set_exception(std::make_exception_ptr(std::runtime_error(errMess)));`.

If you destroy the `std::promise` without calling the `set`-member function or a `std::packaged_task` before invoking it, a `std::future_error` exception with an error code `std::future_errc::broken_-promise` would be stored in the shared state.

3.9.7 Notifications

Tasks are a save replacement for condition variables. If you use promises and futures to synchronize threads, they have a lot in common with condition variables. Most of the time, promises and futures are the better choices.

Before I present you an example, here is the big picture.

---

[24]http://en.cppreference.com/w/cpp/error/current_exception

Condition variables versus tasks

Criteria	Condition Variables	Tasks
Multiple synchronizations	Yes	No
Critical section	Yes	No
Error handling in receiver	No	Yes
Spurious wakeup	Yes	No
Lost wakeup	Yes	No

The advantage of a condition variable to a promise and future is that you can use condition variables to synchronize threads multiple times. In contrast to that, a promise can send its notification only once, so you have to use more promise and future pairs to get the condition variable's functionality. If you use the condition variable for only one synchronization, the condition variable is a lot more challenging to use in the right way. A promise and future pair needs no shared variable and therefore no lock and is not prone to spurious or lost wakeups. In addition to that, tasks can handle exceptions. There are lots of reasons to prefer tasks to condition variables.

Do you remember how difficult it was to use condition variables? If not, here are the key parts required to synchronize two threads.

```cpp
void waitingForWork(){
    std::cout << "Worker: Waiting for work." << '\n';

    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
    doTheWork();
    std::cout << "Work done." << '\n';
}

void setDataReady(){
    std::lock_guard<std::mutex> lck(mutex_);
    dataReady=true;
    std::cout << "Sender: Data is ready."  << '\n';
    condVar.notify_one();
}
```

The function `setDataReady` performs the notification part of the synchronization - the function `waitingForWork` the waiting part of the synchronization.

Here is the same workflow with tasks.

Thread synchronization with tasks
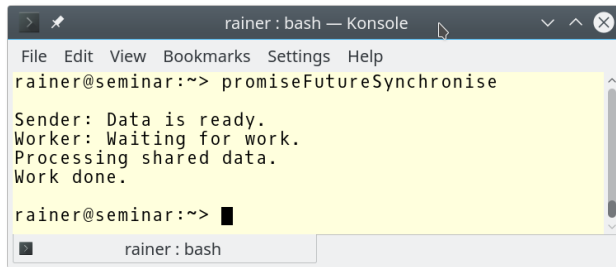
```cpp
1    // promiseFutureSynchronize.cpp
2
3    #include <future>
4    #include <iostream>
5    #include <utility>
6
7
8    void doTheWork(){
9        std::cout << "Processing shared data." << '\n';
10   }
11
12   void waitingForWork(std::future<void>&& fut){
13
14       std::cout << "Worker: Waiting for work." << '\n';
15       fut.wait();
16       doTheWork();
17       std::cout << "Work done." << '\n';
18
19   }
20
21   void setDataReady(std::promise<void>&& prom){
22
23       std::cout << "Sender: Data is ready."  << '\n';
24       prom.set_value();
25
26   }
27
28   int main(){
29
30       std::cout << '\n';
31
32       std::promise<void> sendReady;
33       auto fut = sendReady.get_future();
34
35       std::thread t1(waitingForWork, std::move(fut));
36       std::thread t2(setDataReady, std::move(sendReady));
37
38       t1.join();
39       t2.join();
40
41       std::cout << '\n';
42
43   }
```

That was quite easy.

Thanks to `sendReady` (line 32) you get a future `fut` (line 33). The promise communicates using its return value `void` (`std::promise<void> sendReady`) that it is only capable of sending notifications. Both communication endpoints are moved into threads `t1` and `t2` (lines 35 and 36). The future waits using the call `fut.wait()` (line 15) for the notification of the promise: `prom.set_value()` (line 24).

The program's structure and output match the corresponding program in the section condition variable.

std::promise and std::future as condition variable

3.10 Synchronized Outputstreams (C++20)

What happens when you write without synchronization to `std::cout`?

Non-synchronized access to `std::cout`

```cpp
1   // coutUnsynchronized.cpp
2
3   #include <chrono>
4   #include <iostream>
5   #include <thread>
6
7   class Worker{
8   public:
9     Worker(std::string n):name(n) {};
10      void operator() (){
11        for (int i = 1; i <= 3; ++i) {
12          // begin work
13          std::this_thread::sleep_for(std::chrono::milliseconds(200));
14          // end work
15          std::cout << name << ": " << "Work " << i << " done !!!" << '\n';
16        }
17      }
18  private:
19    std::string name;
20  };
21
22
23  int main() {
24
25    std::cout << '\n';
26
27    std::cout << "Boss: Let's start working.\n\n";
28
29    std::thread herb= std::thread(Worker("Herb"));
30    std::thread andrei= std::thread(Worker("  Andrei"));
31    std::thread scott= std::thread(Worker("    Scott"));
32    std::thread bjarne= std::thread(Worker("      Bjarne"));
33    std::thread bart= std::thread(Worker("        Bart"));
34    std::thread jenne= std::thread(Worker("          Jenne"));
35
36
37    herb.join();
38    andrei.join();
39    scott.join();
```

```
40    bjarne.join();
41    bart.join();
42    jenne.join();
43
44    std::cout << "\n" << "Boss: Let's go home." << '\n';
45
46    std::cout << '\n';
47
48  }
```

The boss has six workers (lines 29 - 34). Each worker has to take care of three work packages that take 1/5 second each (line 13). After the worker is done with his work package, he screams out loudly to the boss (line 15). Once the boss receives notifications from all workers, he sends them home (line 44).

What a mess for such a simple workflow! Each worker screams out his message ignoring his coworkers!

Non-synchronized writing to `std::cout`

I want to remind you. This interleaving of output operations is only a visual issue and not a data race.

How can we solve this issue? With C++11, the answer is straightforward: use a lock such as `lock_guard` to synchronize the access to `std::cout`:

Synchronized access to `std::cout`
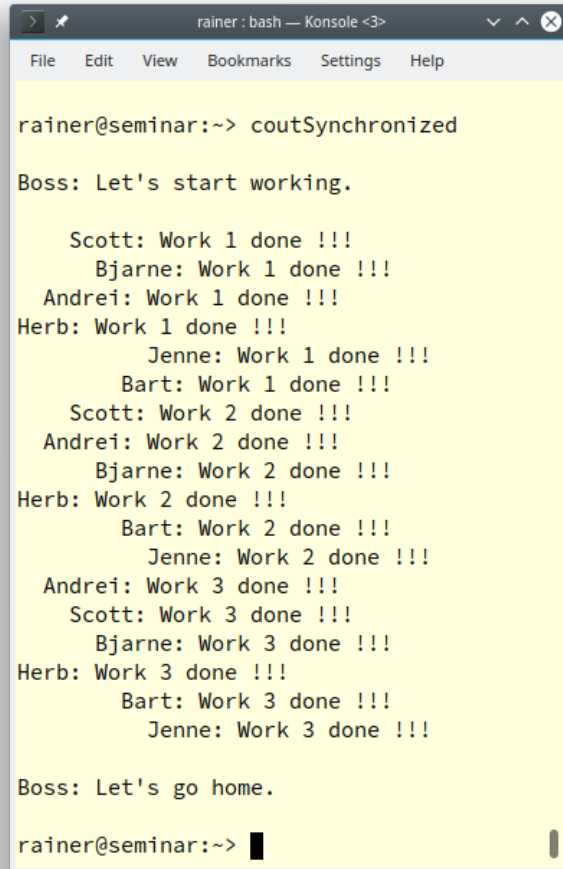
```cpp
1    // coutSynchronized.cpp
2
3    #include <chrono>
4    #include <iostream>
5    #include <mutex>
6    #include <thread>
7
8    std::mutex coutMutex;
9
10   class Worker{
11   public:
12     Worker(std::string n):name(n) {};
13
14       void operator() () {
15         for (int i = 1; i <= 3; ++i) {
16           // begin work
17           std::this_thread::sleep_for(std::chrono::milliseconds(200));
18           // end work
19           std::lock_guard<std::mutex> coutLock(coutMutex);
20           std::cout << name << ": " << "Work " << i << " done !!!\n";
21         }
22       }
23   private:
24     std::string name;
25   };
26
27
28   int main() {
29
30     std::cout << '\n';
31
32     std::cout << "Boss: Let's start working." << "\n\n";
33
34     std::thread herb= std::thread(Worker("Herb"));
35     std::thread andrei= std::thread(Worker("  Andrei"));
36     std::thread scott= std::thread(Worker("    Scott"));
37     std::thread bjarne= std::thread(Worker("      Bjarne"));
38     std::thread bart= std::thread(Worker("        Bart"));
39     std::thread jenne= std::thread(Worker("          Jenne"));
40
41     herb.join();
42     andrei.join();
43     scott.join();
44     bjarne.join();
```
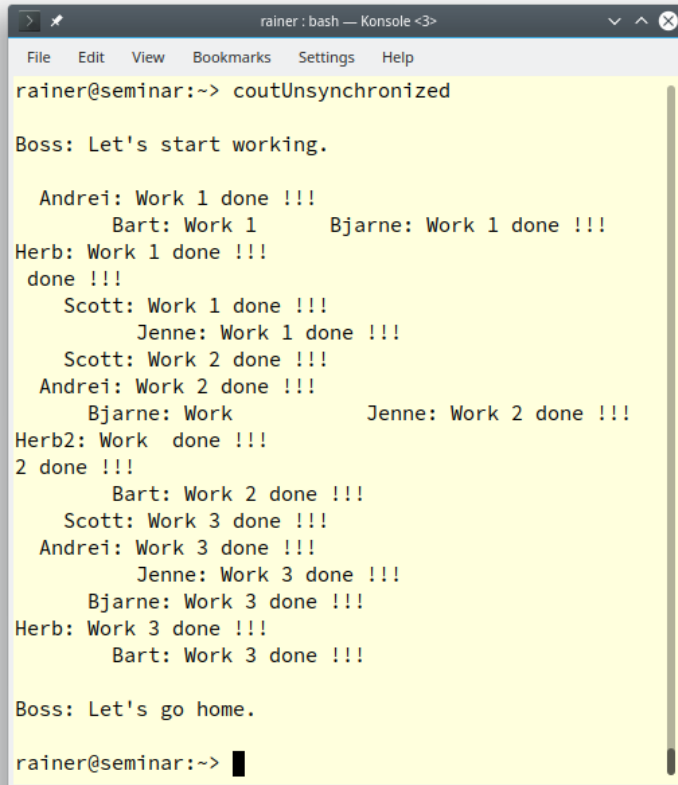
```
45    bart.join();
46    jenne.join();
47
48    std::cout << "\n" << "Boss: Let's go home." << '\n';
49
50    std::cout << '\n';
51
52  }
```

The `coutMutex` in line 8 protects the shared object `std::cout`. Putting the `coutMutex` into a `std::lock_-guard` guarantees that the `coutMutex` is locked in the constructor (line 19) and unlocked in the destructor (line 21) of the `std::lock_guard`. Thanks to the `coutMutex` guarded by the `coutLock` the mess becomes a harmony.

Synchronized access of `std::cout`

With C++20, writing synchronized to `std::cout` is a piece of cake. `std::basic_syncbuf` is a wrapper for a `std::basic_streambuf`[25]. It accumulates output in its buffer. The wrapper sets its content to the wrapped buffer when it is destructed. Consequently, the content appears as a contiguous sequence of characters, and no interleaving of characters can happen.

Thanks to `std::basic_osyncstream`, you can directly write synchronously to `std::cout`.

C++20 defines two specializations of `std::basic_osyncstream` for `char`, and `wchar_t`.

---

[25]https://en.cppreference.com/w/cpp/io/basic_streambuf

```
std::osyncstream            std::basic_osyncstream<char>
std::wosyncstream            std::basic_osyncstream<wchar_t>
```

You can create a named-synchronized output stream. Here is how the previous program coutUnsynchronized.cpp is refactored to write synchronized to std::cout.

Synchronized access of `std::cout` with `std::basic_osyncstream`

```cpp
1   // synchronizedOutput.cpp
2
3   #include <chrono>
4   #include <iostream>
5   #include <syncstream>
6   #include <thread>
7
8   class Worker{
9   public:
10    Worker(std::string n):name(n) {};
11      void operator() (){
12        for (int i = 1; i <= 3; ++i) {
13          // begin work
14          std::this_thread::sleep_for(std::chrono::milliseconds(200));
15          // end work
16          std::osyncstream syncStream(std::cout);
17          syncStream << name << ": " << "Work " << i << " done !!!" << '\n';
18        }
19      }
20   private:
21     std::string name;
22   };
23
24
25   int main() {
26
27     std::cout << '\n';
28
29     std::cout << "Boss: Let's start working.\n\n";
30
31     std::thread herb= std::thread(Worker("Herb"));
32     std::thread andrei= std::thread(Worker("  Andrei"));
33     std::thread scott= std::thread(Worker("    Scott"));
34     std::thread bjarne= std::thread(Worker("      Bjarne"));
35     std::thread bart= std::thread(Worker("        Bart"));
36     std::thread jenne= std::thread(Worker("          Jenne"));
37
38
```

```
39    herb.join();
40    andrei.join();
41    scott.join();
42    bjarne.join();
43    bart.join();
44    jenne.join();
45
46    std::cout << "\n" << "Boss: Let's go home." << '\n';
47
48    std::cout << '\n';
49
50  }
```

The only change to the previous program `coutUnsynchronized.cpp` is that `std::cout` is wrapped in a `std::osyncstream` (line 16). To use the `std::osyncstream`, I add the header `<syncstream>`. When the `std::osyncstream` goes out of scope in line 17, the characters are transferred, and `std::cout` is flushed. It is worth mentioning that the `std::cout` calls in the main program do not introduce a data race and have, therefore, not be synchronized.

Because I use the `syncStream` declared on line 17 only once, a temporary object may be more appropriate. The following code snippet presents the modified call operator.

```
void operator()() {
    for (int i = 1; i <= 3; ++i) {
        // begin work
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        // end work
        std::osyncstream(std::cout) << name << ": " << "Work " << i << " done !!!"
                                    << '\n';
    }
}
```

`std::basic_osyncstream syncStream` offers two interesting member functions.

- `syncStream.emit()` emits all buffered output and executes all pending flushes.
- `syncStream.get_wrapped()` returns a pointer to the wrapped buffer.

cppreference.com[26] shows how you can sequence the output of different output streams with the `get_wrapped` member function.

---

[26]https://en.cppreference.com/w/cpp/io/basic_osyncstream/get_wrapped

Sequence output

```cpp
// sequenceOutput.cpp

#include <syncstream>
#include <iostream>
int main() {

  std::osyncstream bout1(std::cout);
  bout1 << "Hello, ";
  {
      std::osyncstream(bout1.get_wrapped()) << "Goodbye, " << "Planet!" << '\n';
  } // emits the contents of the temporary buffer

  bout1 << "World!" << '\n';

} // emits the contents of bout1
```

```
Goodbye, Planet!
Hello, World!
```

Synchronized access of `std::cout`

Distilled Information

- C++ supports two kind of threads: the basic thread `std::thread` (C++11) and the improved thread `std::jthread` (C++20).

    - A `std::thread` gets a callable as work package and starts immediately. The creator of a thread is responsible for the created thread. Either the creator waits until the created thread `t` is done (`t.join()`) or the creator detaches (`t.detach()`) itself from the created thread. A thread is joinable if no operation `t.join()` or `t.detach()` was performed on it. A joinable thread calls `std::terminate` in its destructor, and the program terminates.

    - A `std::jthread` extends the interface of a `std::thread`. It automatically joins in its destructor and supports cooperative interruption.

- You have to coordinate access to a shared variable if more than one thread uses it simultaneously and the variable is mutable. A mutex guarantees that only one thread can access a shared variable at any given time. You should encapsulate a mutex in a lock to release the mutex automatically and, therefore, overcome the many issues of mutexes. C++ offers mutex and locks in many variations.

- If shared data is read-only, it's sufficient to initialize it in a thread-safe way. C++ offers various ways to achieve this, including using a constant expression, a static variable with block scope, or the function `std::call_once` combined with the flag `std::once_flag`.

- Declaring a variable as thread-local ensures that each thread gets its copy. The lifetime of thread-local data is bound to the lifetime of its thread.

- Condition variables enable threads to be synchronized via messages. One thread acts as the sender while the other acts as the receiver of the message. The receiver blocks waiting for the message from the sender.

- `std::jthread` and `std::condition_variable_any` support cooperative interruption by design. Cooperative interruption is based on the `std::stop_source`, `std::stop_-token`, and the `std::stop_callback`.

- Semaphores are a synchronization mechanism used to control concurrent access to a shared resource. The counter of a semaphore is initialized in the constructor. Acquiring the semaphore decreases the counter, and releasing the semaphore increases the counter. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

- Latches and barriers are coordination types that enable some threads to block until a counter becomes zero. The counter is initialized in the constructor.

- Tasks have a lot in common with threads. While you explicitly create a thread, a task is just a job you start. The C++ runtime automatically handles the task's lifetime, such as in the simple case of std::async. Tasks are like data channels between two communication endpoints. They enable thread-safe communication between threads. The promise as one endpoint puts data into the data channel, the future at the other endpoint picks the value up. The data can be a value, an exception, or simply a notification. In addition to std::async, C++ has the class templates std::promise and std::future that give you more control over the task.

- C++ enables synchronized output streams. A synchronized output stream accumulates output in its buffer and flushes it when it is destructed. Consequently, the content appears as a contiguous sequence of characters, and no interleaving of characters can happen

4. Parallel Algorithms of the Standard Template Library
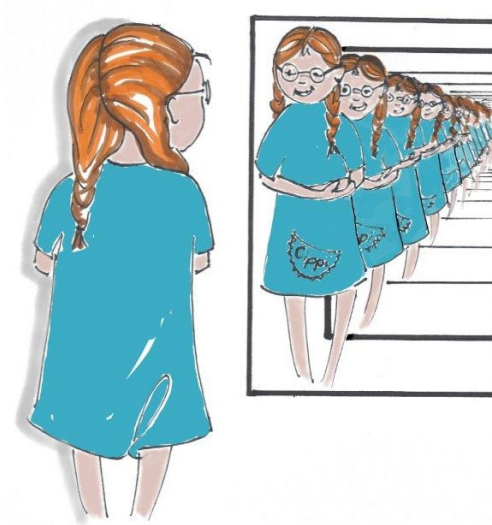
Cippi clones herself

The Standard Template Library has more than 100 algorithms for searching, counting, and manipulating ranges and their elements. With C++17, 69 of them get new overloads, and eight new ones are added. The overloaded, and new algorithms can be invoked with a so-called execution policy.

C++11	C++14	C++17	C++20	C++23
2011	2014	2017	2020	2023

- Memory model
- Atomics

- Threads
- Mutexes and locks
- Thread local data
- Condition variables
- Tasks

- Reader-writer locks

- Parallel STL

- Atomic smart pointers

- std::jthread
- Latches and barriers
- Semaphores

- Coroutines

- Executors
- `std::future` extensions
- Transactional memory
- Task blocks
- Data-parallel vector library

Parallel algorithms of C++17
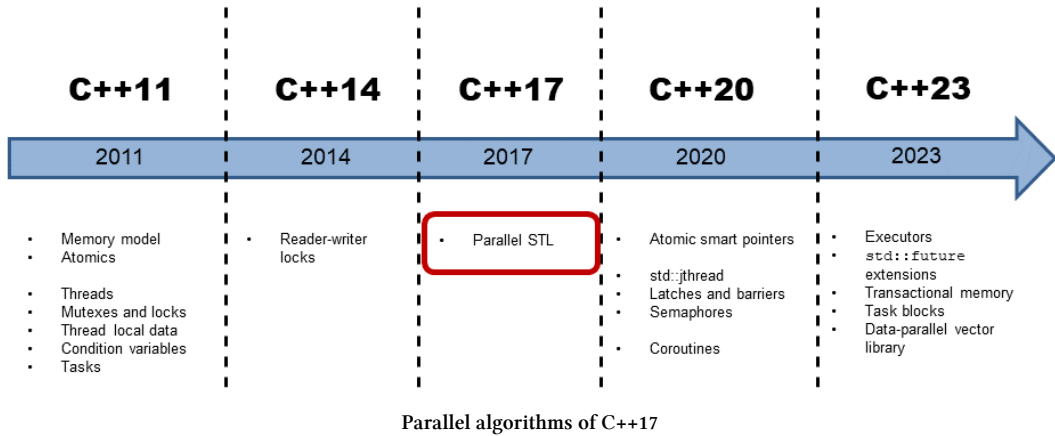
Using an execution policy, you can specify whether the algorithm should run sequentially, in parallel, or parallel with vectorization. For using the execution policy, you have to include the header `<execution>`.

4.1 Execution Policies

The standard defines three execution policies:

- **`std::execution::sequenced_policy`**
- **`std::execution::parallel_policy`**
- **`std::execution::parallel_unsequenced_policy`**
- **`std::execution::unsequenced_policy`** (C++20)

The corresponding policy tag specifies whether a program should run sequentially, in parallel, or parallel with vectorization.

- **`std::execution::seq`**: runs the program sequentially
- **`std::execution::par`**: runs the program in parallel on multiple threads
- **`std::execution::par_unseq`**: runs the program in parallel on multiple threads and allows the interleaving of individual loops; permits a vectorized version with SIMD[1] (**S**ingle **I**nstruction **M**ultiple **D**ata) extensions.
- **`std::execution::unseq`** (C++20): runs the program in parallel on multiple threads; permits a vectorized version with SIMD.

The usage of the execution policy `std::execution::par`, `std::execution::unseq`, or `std::execution::par_-unseq` allows the algorithm to run parallel or parallel and vectorized. The difference between `std::execution::unseq`, and `std::execution::par_unseq` is that the later allows the interleaving the individual threads.

This policy is a permission and not a requirement.

The following code snippet applies 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 vectorized execution
std::sort(std::execution::par_unseq, v.begin(), v.end());
```

---

[1] https://en.wikipedia.org/wiki/SIMD

```
14
15   // permitting parallel and vectorized execution, but no interleaving of individual threads
16   std::sort(std::execution::unseq, v.begin(), v.end());
```

The example shows that you can still use the classic variant of `std::sort` (line 4). Besides, in C++17, you can specify explicitly whether the sequential (line 7), parallel (line 10), or the parallel and vectorized (line 13) version should be used.

Thanks to `std::is_execution_policy` you can check, whether `T` is a standard or implementation-defined execution policy type: `std::is_execution_policy<T>::value`. The function checks whether T is a standard or implementation-defined execution policy type. The expression evaluates to `true`, if `T` is `std::execution::sequenced_policy`, `std::execution::parallel_policy`, `std::execution::unsequenced_-policy`, `std::execution::parallel_unsequenced_policy`, or an implementation-defined execution policy type. Otherwise, value is equal to `false`.

4.1.1 Parallel and Vectorized Execution

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

The following example shows a simple loop for filling a vector.

Filling a vector

```
1    const int SIZE= 8;
2
3    int vec[] = {1, 2, 3, 4, 5, 6, 7, 8};
4    int res[] = {0, 0, 0, 0, 0, 0, 0, 0};
5
6    int main(){
7        for (int i = 0; i < SIZE; ++i) {
8            res[i] = vec[i]+5;
9        }
10   }
```

Line 8 is the crucial line in this small example. Thanks to Compiler Explorer[2], we can have a closer look at the assembler instructions generated by clang 3.6.

4.1.1.1 Without Optimization

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

---

[2]https://godbolt.org/

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

Sequential execution

4.1.1.2 With Maximum Optimization

By using the highest optimization level, `-O3`, special registers such as `xmm0` are used that can hold 128 bits or 4 `ints`. This special register 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
```

Vectorized execution

An overload of an algorithm without an execution policy and an overload of an algorithm with a sequential execution policy `std::execution::seq` differ in one aspect: exceptions.

4.1.2 Exceptions

If an exception occurs during the usage of an algorithm with an execution policy, `std::terminate`[3] is called. `std::terminate` calls the installed `std::terminate_handler`[4]. The consequence is that per default `std::abort`[5] is called, which causes abnormal program termination. The handling of exceptions is the difference between an algorithm's invocation without an execution policy and an algorithm with a sequential `std::execution::seq` execution policy. The invocation of the algorithm without an execution policy propagates the exception, and, therefore, the exception can be handled. The program `exceptionExecutionPolicy.cpp` shows my point.

---

[3]https://en.cppreference.com/w/cpp/error/terminate
[4]https://en.cppreference.com/w/cpp/error/terminate_handler
[5]https://en.cppreference.com/w/cpp/utility/program/abort

Exceptions with execution policies
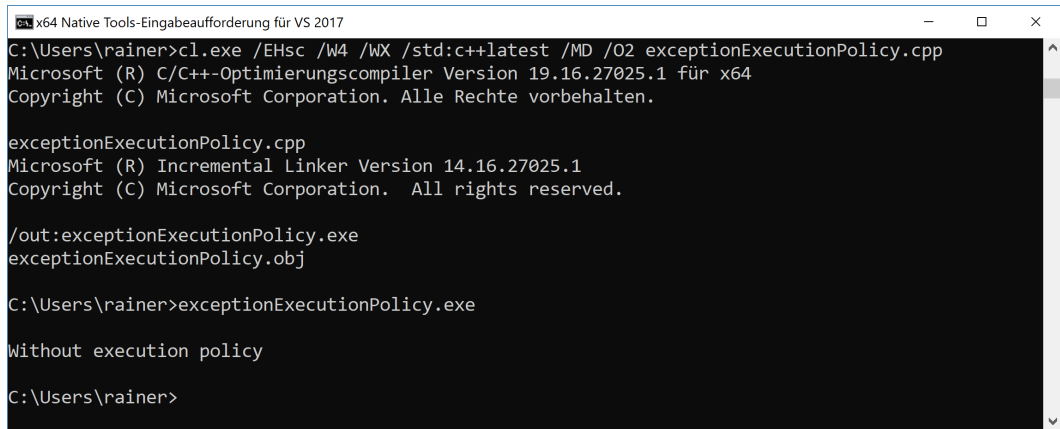
```cpp
1   // exceptionExecutionPolicy.cpp
2
3   #include <algorithm>
4   #include <execution>
5   #include <iostream>
6   #include <stdexcept>
7   #include <string>
8   #include <vector>
9
10  int main(){
11
12      std::cout << '\n';
13
14      std::vector<int> myVec{1, 2, 3, 4, 5};
15
16      try{
17          std::for_each(myVec.begin(), myVec.end(),
18              [](int){ throw std::runtime_error("Without execution policy"); }
19          );
20      }
21      catch(const std::runtime_error& e){
22          std::cout << e.what() << '\n';
23      }
24
25      try{
26          std::for_each(std::execution::seq, myVec.begin(), myVec.end(),
27              [](int){ throw std::runtime_error("With execution policy"); }
28          );
29      }
30      catch(const std::runtime_error& e){
31          std::cout << e.what() << '\n';
32      }
33      catch(...){
34          std::cout << "Catch-all exceptions" << '\n';
35      }
36
37  }
```

The exception handler in line 21 catches the exception std::runtime_error but doesn't work with the exception handler in line 30 or even in line 33 with a catch-all exception handler.

With a new MSVC compiler and the flag std:c++latest, the program gives the expected output.

Exception policy of the parallel algorithms

Only the first exception handler is executed.

4.1.3 Hazards of Data Races and Deadlocks

The parallel algorithm does not automatically protect you from data races and deadlocks.

Parallel execution with a data race

```cpp
std::vector<int> v = {1, 2, 3 };
int sum = 0;

std::for_each(std::execution::par, v.begin(), v.end(), [&sum](int i){
    sum += i + i;
});
```

The small code snippet has a data race on sum. sum builds the sum of all i + i and is concurrently modified. sum has to be protected.

Parallel execution

```cpp
std::vector<int> v = {1, 2, 3 };
int sum = 0;
std::mutex m;

std::for_each(std::execution::par, v.begin(), v.end(), [&sum](int i){
    std::lock_guard<std::mutex> lock(m);
    sum += i + i;
});
```

When I change the execution policy to std::execution::par_unseq, I have a race condition that usually results in a deadlock.

Parallel and vectorized execution with a deadlock

```cpp
std::vector<int> v = {1, 2, 3 };
int sum = 0;
std::mutex m;

std::for_each(std::execution::par_unseq, v.begin(), v.end(), [&sum](int i){
    std::lock_guard<std::mutex> lock(m);
    sum += i + i;
});
```

The lambda function may result in two consecutive calls of `m.lock` on the same thread. Two times trying to lock a non-recursive `std::mutex` is undefined behavior and gives most of the time a deadlock. You can avoid the deadlock by using an atomic.

Parallel and vectorized execution without a deadlock

```cpp
std::vector<int> v = {1, 2, 3 };
std::atomic<int> sum = 0;
std::mutex m;
std::for_each(std::execution::par_unseq, v.begin(), v.end(), [&sum](int i){
    sum += i + i;
});
```

Because `sum` is an atomic counter, relaxed semantic is also fine: `sum.fetch_add(i + i, std::memory_-order_relaxed);`.

69 of the STL algorithms can be parametrized with an execution policy. Additionally, C++17 has eight new algorithms.

4.2 Algorithms

Here are the 69 algorithms with parallelized versions.

The 69 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::fill	std::fill_n
std::find	std::find_end	std::find_first_of	std::find_if
std::find_if_not	std::generate	std::generate_n	std::includes
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::remove	std::remove_copy	std::remove_copy_if	std::remove_if
std::replace	std::replace_copy	std::replace_copy_if	std::replace_if
std::reverse	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::uninitialized_copy
std::uninitialized_copy_n	std::uninitialized_fill	std::uninitialized_fill_n	std::unique
std::unique_copy			

Besides, we get eight new algorithms.

4.3 The New Algorithms

The new algorithms are in the `std` namespace. The algorithms `std::for_each` and `std::for_each_n` require the header `<algorithm>`. The remaining six other algorithms require the header `<numeric>`.

Here is an overview of the new algorithms.

The new algorithms

Algorithm	Description
`std::for_each`	Applies a unary callable to the range.
`std::for_each_n`	Applies a unary callable to the first n elements of the range.

The new algorithms

Algorithm	Description
`std::exclusive_scan`	Applies from the left a binary callable up to the ith (exclusive) element of the range and writes the result to an output range. The left argument of the callable is the previous result. Stores intermediate results. If the binary callable is non-associative the result is non-deterministic. Similar to `std::partial_sum`[6]
`std::inclusive_scan`	Applies from the left a binary callable up to the ith (inclusive) element of the range and writes the result to an output range. The left argument of the callable is the previous result. Stores intermediate results. If the binary callable is non-associative the result is non-deterministic. Similar to `std::partial_sum`
`std::transform_exclusive_scan`	First applies a unary callable to the range and then applies `std::exclusive_scan`. If the binary callable is non-associative the result is non-deterministic.
`std::transform_inclusive_scan`	First applies a unary callable to the range and then applies `std::inclusive_scan`. If the binary callable is non-associative the result is non-deterministic.
`std::reduce`	Applies from the left a binary callable to the range. If the binary callable is non-associative or non-commutative the result is non-deterministic. Similar to `std::accumulate`[7]
`std::transform_reduce`	Applies first a unary callable to one or a binary callable to two ranges and then `std::reduce` to the resulting range. If the binary callables are non-associative or non-commutative the result is non-deterministic.

Admittedly this description is not easy to digest, but if you already know `std::accumulate` and `std::partial_sum`, the reduce and scan variations should be quite familiar. `reduce` is the parallel pendant to accumulate and scan the parallel pendant to partial_sum. The parallel execution is the reason that `std::reduce` needs an associative and commutative callable. The corresponding statement hold for the scan variations in contrary to the partial_sum variations.

First, I present an exhaustive example of the algorithms and then write about these functions' functional heritage. In my example, I ignore the new `std::for_each` algorithm. In contrast to the C++98 variant that returns a unary function, the additional C++17 variant returns nothing. While `std::accumulate` processes its elements from left to the right, `std::reduce` does it in an arbitrary order. Let me start with a small code snippet using `std::accumulate` and `std::reduce`. The callable is the lambda function `[](int a, int b){ return a * b; }`.

---

[6]http://en.cppreference.com/w/cpp/algorithm/partial_sum
[7]http://en.cppreference.com/w/cpp/algorithm/accumulate

```
std::vector<int> v{1, 2, 3, 4};

std::accumulate(v.begin(), v.end(), 1, [](int a, int b){ return a * b; });
std::reduce(std::execution::par, v.begin(), v.end(), 1 ,
            [](int a, int b){ return a * b; });
```

The two following graphs show the different fold strategies of std::accumulate and std::reduce.
std::accumulate starts at the left and successively applies the binary operator.
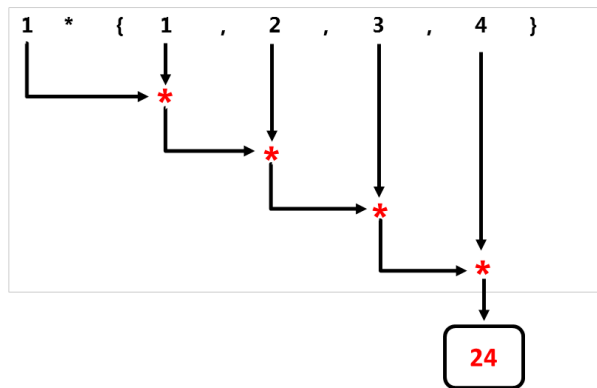
Fold strategy of std::accumulate

On the contrary, std::reduce applies the binary operator in a non-deterministic way.
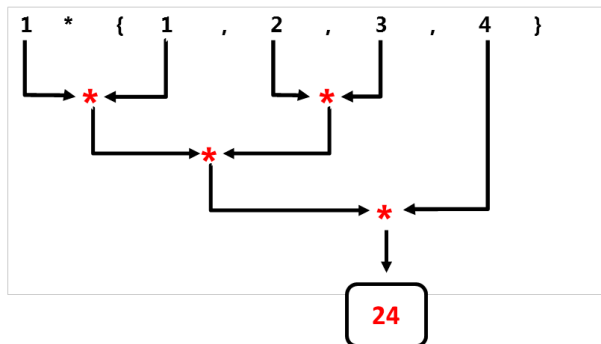
Fold strategy of std::reduce

The associativity allows the std::reduce algorithm to compute the reduction step on arbitrary adjacents pairs of elements. Thanks to commutativity, the intermediate results can be computed in an arbitrary order.

The new algorithms

```cpp
// newAlgorithm.cpp

#include <algorithm>
#include <execution>
#include <numeric>
#include <iostream>
#include <string>
#include <vector>


int main(){

  std::cout << '\n';

  // for_each_n

  std::vector<int> intVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  std::for_each_n(std::execution::par,
                  intVec.begin(), 5, [](int& arg){ arg *= arg; });

  std::cout << "for_each_n: ";
  for (auto v: intVec) std::cout << v << " ";
  std::cout << "\n\n";

  // exclusive_scan and inclusive_scan
  std::vector<int> resVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::exclusive_scan(std::execution::par,
                      resVec.begin(), resVec.end(), resVec.begin(), 1,
                      [](int fir, int sec){ return fir * sec; });

  std::cout << "exclusive_scan: ";
  for (auto v: resVec) std::cout << v << " ";
  std::cout << '\n';

  std::vector<int> resVec2{1, 2, 3, 4, 5, 6, 7, 8, 9};

  std::inclusive_scan(std::execution::par,
                      resVec2.begin(), resVec2.end(), resVec2.begin(),
                      [](int fir, int sec){ return fir * sec; }, 1);

  std::cout << "inclusive_scan: ";
  for (auto v: resVec2) std::cout << v << " ";
  std::cout << "\n\n";

```

```cpp
45    // transform_exclusive_scan and transform_inclusive_scan
46    std::vector<int> resVec3{1, 2, 3, 4, 5, 6, 7, 8, 9};
47    std::vector<int> resVec4(resVec3.size());
48    std::transform_exclusive_scan(std::execution::par,
49                                  resVec3.begin(), resVec3.end(),
50                                  resVec4.begin(), 0,
51                                  [](auto fir, int sec){ return fir + sec; },
52                                  [](auto arg){ return arg * arg; });
53
54    std::cout << "transform_exclusive_scan: ";
55    for (auto v: resVec4) std::cout << v << " ";
56    std::cout << '\n';
57
58    std::vector<std::string> strVec{"Only","for","testing","purpose"};
59    std::vector<int> resVec5(strVec.size());
60
61    std::transform_inclusive_scan(std::execution::par,
62                                  strVec.begin(), strVec.end(),
63                                  resVec5.begin(),
64                                  [](auto fir, auto sec){ return fir + sec; },
65                                  [](auto s){ return s.length(); }, static_cast<std::size_t\
66    >(0));
67
68
69    std::cout << "transform_inclusive_scan: ";
70    for (auto v: resVec5) std::cout << v << " ";
71    std::cout << "\n\n";
72
73    // reduce and transform_reduce
74    std::vector<std::string> strVec2{"Only","for","testing","purpose"};
75
76    std::string res = std::reduce(std::execution::par,
77                      strVec2.begin() + 1, strVec2.end(), strVec2[0],
78                      [](auto fir, auto sec){ return fir + ":" + sec; });
79
80    std::cout << "reduce: " << res << '\n';
81
82    auto res7 = std::transform_reduce(std::execution::par,
83                      strVec2.begin(), strVec2.end(), static_cast<std::size_t>(0),
84                      [](auto a, auto b){ return a + b; },
85                      [](std::string s){ return s.length(); });                            \
86
87
88    std::cout << "transform_reduce: " << res7 << '\n';
89
```

```
90     std::cout << '\n';
91
92  }
```

I apply the new algorithms to a `std::vector<int>` (line 17) and a `std::vector<std::string>` (line 58).

The `std::for_each_n` algorithm in line 18 maps the first `n` ints of the vector to their squares.

`std::exclusive_scan` (line 27) and `std::inclusive_scan` (line 37) are quite similar. Both apply a binary operation to their elements. The difference is that `std::exclusive_scan` excludes the last element in each iteration.

The `std::transform_exclusive_scan` in line 48 is quite challenging to read. Let me try to explain it. In the first step I apply the lambda function `[](int arg){ return arg * arg; }` to each element of the range `resVec3.begin()` to `resVec3.end()`. In the second step, I apply the binary operation `[](int fir, int sec){ return fir + sec; }` to the intermediate vector. This means, sum up all elements using 0 as the initial value. The result is placed in `resVec4.begin()`.

The `std::transform_inclusive_scan` function in line 61 is similar. This function maps each element to its length.

The `std::reduce` function should be pretty easy to read; it puts `":"` characters between every two elements of the input vector. The resulting string should not start with a `":"` character; therefore, the range starts at the second element (`strVec2.begin() + 1`) and uses the first element of the vector `strVec2[0]` as the initial element.
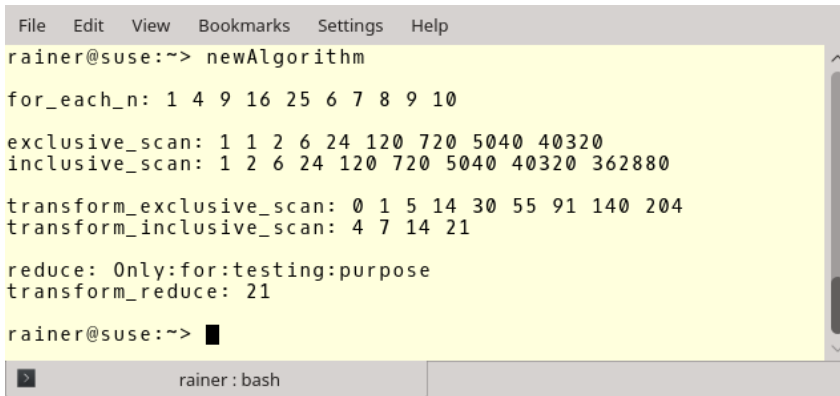
> **transform_reduce becomes map_reduce**
>
> I have more to say about the `std::transform_reduce` function in line 80. First of all, the C++ algorithm `transform` is in other languages often called `map`. Therefore, we could also call `std::transform_reduce` `std::map_reduce`. Now I assume you noticed it. `std::transform_reduce` is the well-known parallel MapReduce[8] algorithm implemented in C++. Accordingly, `std::transform_reduce` maps a unary callable (`[](std::string s){ return s.length(); }`) onto a range and reduces the pair to a output value: `[](std::size_t a, std::size_t b){ return a + b; }`.

Studying the output of the program should help you.

---

[8]https://en.wikipedia.org/wiki/MapReduce

The new algorithms

4.3.1 More overloads

All C++ variants reducing and scanning have more overloads. In the simplest form, you can invoke them without a binary callable and an initial element. If you do not use a binary callable, the addition is used as the binary operation. If you do not specify an initial element, the initial element depends on the used algorithm:

- `std::inclusive_scan` and `std::transform_inclusive_scan`: the first element.
- `std::reduce` and `std::transform_reduce`: `typename std::iterator_traits<InputIt>::value_-type{}`.

Let's look at these new algorithms from a functional perspective.

4.3.2 The functional Heritage

Long story short: all new functions have a pendant in the pure functional language Haskell.
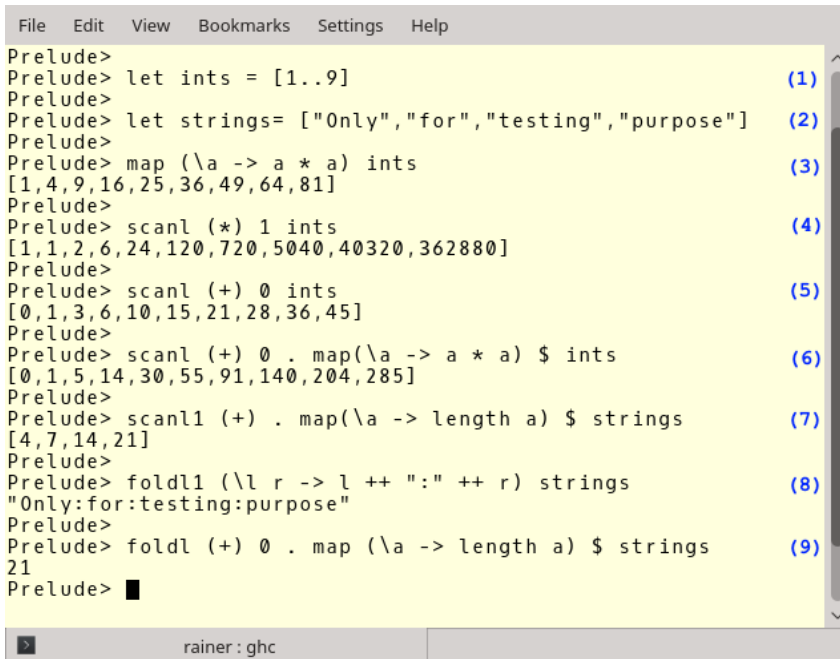
- `std::for_each_n` is called `map` in Haskell.
- `std::exclusive_scan` and `std::inclusive_scan` are called `scanl` and `scanl1` in Haskell.
- `std::transform_exclusive_scan` and `std::transform_inclusive_scan` is a composition of the Haskell functions `map` and `scanl` or `scanl1`.
- `std::reduce` is called `foldl` or `foldl1` in Haskell.
- `transform_reduce` is a composition of the Haskell functions `map` and `foldl` or `foldl1`.

Before I show you Haskell in action, let me say a few words about the different functions.

- `map` applies a function to a list.

- `foldl` and `foldl1` apply a binary operation to a list and reduce the list to a value. `foldl` needs, in contrast to `foldl1` an initial value. `foldl1` requires the list to be non-empty and uses the first element as the initial value.
- `scanl` and `scanl1` apply the same strategy such as `foldl` and `foldl1`, but they produce all intermediate results so that you get back a list.
- `foldl`, `foldl1`, `scanl`, and `scanl1` start their job from the left.

Let's have a look at the Haskell functions. Here is Haskell's interpreter shell.

```
File   Edit   View   Bookmarks   Settings   Help

Prelude>
Prelude> let ints = [1..9]                                             (1)
Prelude>
Prelude> let strings= ["Only","for","testing","purpose"]    (2)
Prelude>
Prelude> map (\a -> a * a) ints                                       (3)
[1,4,9,16,25,36,49,64,81]
Prelude>
Prelude> scanl (*) 1 ints                                             (4)
[1,1,2,6,24,120,720,5040,40320,362880]
Prelude>
Prelude> scanl (+) 0 ints                                             (5)
[0,1,3,6,10,15,21,28,36,45]
Prelude>
Prelude> scanl (+) 0 . map(\a -> a * a) $ ints                       (6)
[0,1,5,14,30,55,91,140,204,285]
Prelude>
Prelude> scanl1 (+) . map(\a -> length a) $ strings                 (7)
[4,7,14,21]
Prelude>
Prelude> foldl1 (\l r -> l ++ ":" ++ r) strings                    (8)
"Only:for:testing:purpose"
Prelude>
Prelude> foldl (+) 0 . map (\a -> length a) $ strings              (9)
21
Prelude> █

     ▶               rainer : ghc
```

New algorithms in Haskell

(1) and (2) define a list of integers and a list of strings. In (3), I apply the lambda function (`\a -> a * a`) to the list of integers. (4) and (5) are more sophisticated. The expression (4) multiplies (*) all pairs of integers starting with the one as the neutral element of multiplication. Expression (5) does the corresponding for addition. Expressions (6), (7), and (9) are for the imperative eye quite challenging. You have to read them from right to left. `scanl1 (+) . map(\a -> length)` (7) is a function composition. The dot (.) symbol composes the two functions. The first function maps each element to its length; the second function adds the list of lengths together. (9) is similar to (7). The difference is that `foldl` produces one value and requires an initial element, that is in this case 0. Now expression (8) should be readable; it successively joins two strings with the ":" character.

4.4 Compiler Support

As far as I know, there is no fully standard-compliant implementation of the parallel STL available. Thanks to the Microsoft Visual Compiler and the GCC Compiler, you can at least use the sequential and parallel execution policy but neither of the parallel and vectorized one: `std::execution::unseq` or `std::execution::par_unseq`. When you request a parallel and vectorized execution policy, the compiler maps it to the parallel execution policy (`std::execution::par`).

4.4.1 Microsoft Visual Compiler

MSVC 17.8 added support for about 30 algorithms.
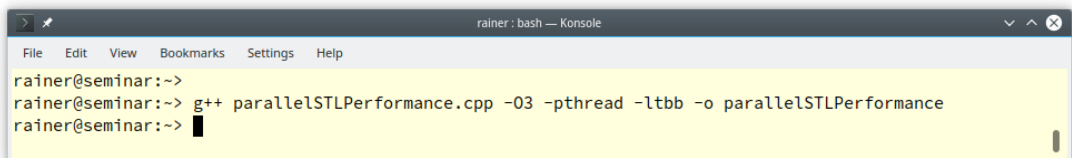
Parallel Algorithms with MSVC 17.8

std::adjacent_difference	std::adjacent_find	std::all_of
std::any_of	std::count	std::count_if
std::equal	std::exclusive_scan	std::find
std::find_end	std::find_first_of	std::find_if
std::for_each	std::for_each_n	std::inclusive_scan
std::mismatch	std::none_of	std::reduce
std::remove	std::remove_if	std::search
std::search_n	std::sort	std::stable_sort
std::transform	std::transform_exclusive_scan	std::transform_inclusive_scan
std::transform_reduce		

4.4.2 GCC Compiler

Thanks to Intels [Threading Building Block](https://en.wikipedia.org/wiki/Threading_Building_Blocks)[9] (TBB) you can use the parallel STL algorithms with the GCC 9. The TBB is a C++ template library developed by Intel for parallel programming on multi-core processors. To be precise, you need TBB 2018 version or higher.

Using the TBB is easy. You have to link against the TBB specifying the flag `-ltbb`.

Using the Threading Building Blocks

[9]https://en.wikipedia.org/wiki/Threading_Building_Blocks

4.4.3 Further Implementations of the Parallel STL

I used the HPX implementation to get my first intuition about the parllel STL algorithms. HPX (High-Performance ParallelX)[10] 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 its namespace.

For completeness, here are further (partial) implementations of the parallel STL:

- Intel[11]
- Thibaut Lutz[12]
- Nvidia (thrust)[13]
- Codeplay[14]

4.5 Performance

The program `parallelSTLPerformance.cpp` calculates the tangents with the sequential, parallel, and parallel and vectorized execution policy.

Performance of the various execution policies

```
1   // parallelSTLPerformance.cpp
2
3   #include <algorithm>
4   #include <cmath>
5   #include <chrono>
6   #include <execution>
7   #include <iostream>
8   #include <random>
9   #include <string>
10  #include <vector>
11
12  constexpr long long size = 500'000'000;
13
14  const double pi = std::acos(-1);
15
16  template <typename Func>
17  void getExecutionTime(const std::string& title, Func func){
18
19    const auto sta = std::chrono::steady_clock::now();
```

---

[10]http://stellar.cct.lsu.edu/projects/hpx/
[11]https://software.intel.com/en-us/get-started-with-pstl
[12]https://github.com/t-lutz/ParallelSTL
[13]https://thrust.github.io/doc/group__execution__policies.html
[14]https://github.com/KhronosGroup/SyclParallelSTL

```
20     func();
21     const std::chrono::duration<double> dur = std::chrono::steady_clock::now() - sta;
22     std::cout << title << ": " << dur.count() << " sec. " << '\n';
23
24  }
25
26  int main(){
27
28     std::cout << '\n';
29
30     std::vector<double> randValues;
31     randValues.reserve(size);
32
33     std::mt19937 engine;
34     std::uniform_real_distribution<> uniformDist(0, pi / 2);
35     for (long long i = 0 ; i < size ; ++i) randValues.push_back(uniformDist(engine));
36
37     std::vector<double> workVec(randValues);
38
39     getExecutionTime("std::execution::seq", [workVec]() mutable {
40       std::transform(std::execution::seq, workVec.begin(), workVec.end(),
41                         workVec.begin(),
42                   [](double arg){ return std::tan(arg); }
43                   );
44     });
45
46     getExecutionTime("std::execution::par", [workVec]() mutable {
47       std::transform(std::execution::par, workVec.begin(), workVec.end(),
48                         workVec.begin(),
49                   [](double arg){ return std::tan(arg); }
50                   );
51     });
52
53     getExecutionTime("std::execution::par_unseq", [workVec]() mutable {
54       std::transform(std::execution::par_unseq, workVec.begin(), workVec.end(),
55                         workVec.begin(),
56                   [](double arg){ return std::tan(arg); }
57                   );
58     });
59
60     std::cout << '\n';
61
62  }
```

The program `parallelSTLPerformance.cpp` calculates the tangents with the sequential (line 39), parallel (line 46), and parallel and vectorized (line 53) execution policy. First, the vector `randValues` is filled with 500 million numbers from the half-open interval [0, pi / 2 [. The function template `getExecutionTime` (lines 16 - 24) gets the title, and the lambda function executes the lambda function (line 20), and shows the execution time (line 22). There is one particular point about the three lambda functions (lines 39, 46, and 53) used in this program. They are declared as `mutable`. This is necessary because the lambda functions modify its argument `workVec`. Lambda functions are per default constant. If a lambda function wants to change its values, it has to be declared `mutable`.

Compiler Comparison

I explicitly want to emphasize. I don't want to compare the Microsoft Visual compiler and the GCC compiler. Both compilers run on computers with different capabilities. These performance numbers should only give you a gut feeling. This means if you want the numbers for your system, you have to repeat the test. To make it short. I'm really keen to know if the parallel execution of the STL algorithms pays and to what extent. My main focus is the relative performance of the sequential and parallel execution

I use maximum optimization on Windows and Linux. This means for Windows the flag `/O2` and on Linux the flag `-O3`.

4.5.1 Microsoft Visual Compiler

My Windows laptop has eight logical cores, but the parallel execution is more than ten times faster.

Sequential, parallel, and parallel and vectorized execution using the Microsoft Visual Compiler
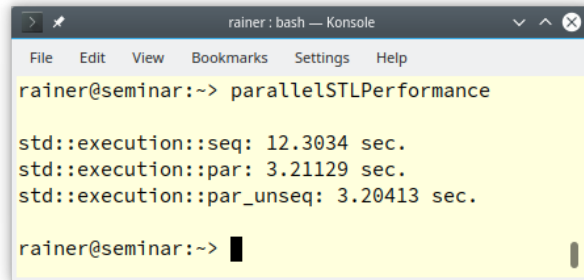
The numbers for the parallel and the parallel and vectorized execution are in the same ballpark. Here is the explanation for the Visual C++ Team Blog: Using C++17 Parallel Algorithms for Better Performance[15]: *Note that the Visual C++ implementation implements the parallel and parallel*

---

[15]https://blogs.msdn.microsoft.com/vcblog/2018/09/11/using-c17-parallel-algorithms-for-better-performance/

unsequenced policies the same way, so you should not expect better performance for using par_unseq on our implementation, but implementations may exist that can use that additional freedom someday.

4.5.2 GCC Compiler

My Linux computer has only four cores. Here are the numbers.

Sequential, parallel, and parallel and vectorized execution using the GCC Compiler

The numbers are as expected. I have four cores and the parallel execution is about four times faster than the sequential execution. The performance numbers of the parallel and vectorized version and the parallel version are in the same ballpark. My assumption is, therefore, that the GCC compiler uses the same strategy such as the Windows compiler. When I ask for the parallel and vectorized execution by using the execution policy `std::execution::par_unseq`, I get the parallel execution policy (`std::execution::par`). This behavior is according to the C++17 standard because the execution policies are only a hint for the compiler.

Distilled Information

- With C++17, most of the STL algorithms are available in a parallel implementation. This makes it possible to invoke an algorithm with a so-called execution policy. This policy specifies whether the algorithm runs sequentially, in parallel, or in parallel with additional vectorization.
- Additionally to the 69 algorithms available in overloaded versions for parallel or parallel and vectorized execution, we get new algorithms. These new ones are well suited for parallel reducing, scanning, or transforming ranges.

5. Coroutines (C++20)
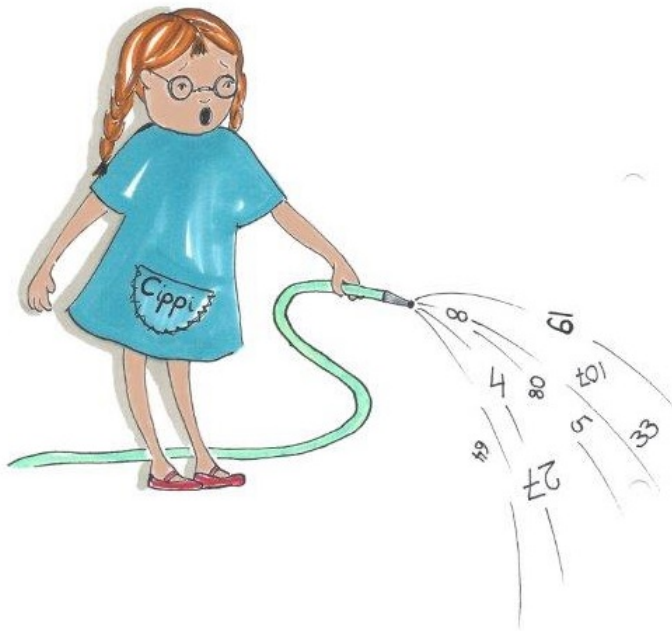
Cippi waters the flowers

Coroutines are functions that can suspend and resume their execution while keeping their state. The evolution of functions in C++ goes one step further.
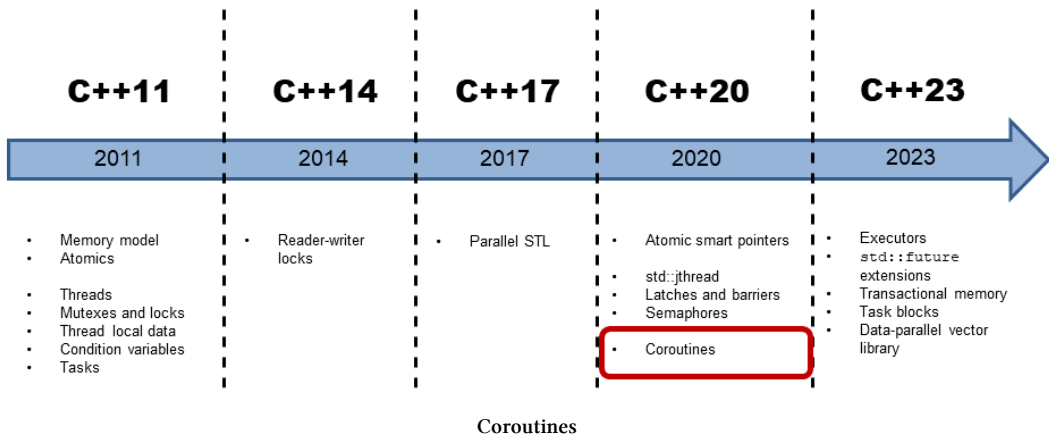
Coroutines

⚠ The Challenge of Understanding Coroutines

It was quite a challenge for me to understand coroutines. I strongly suggest that you should not read the sections in the chapter in sequence. Skip in your first iteration the sections "The Framework", and "The Workflow". Furthermore, read the case studies "Variations of Futures", "Modification and Generalization of a Generator", and "Various Job Workflows". Reading, studying, and playing with the provided examples should give you an initial intuition need for you to actually dive into details and the workflow of coroutines.

What I present in this section as a new idea in C++20 is quite old. The term coroutine was coined by Melvin Conway[1]. He used it in his publication on compiler construction in 1963. Donald Knuth[2] called procedures a special case of coroutines. Sometimes, it just takes a while to get your ideas accepted.

---

[1]https://en.wikipedia.org/wiki/Melvin_Conway
[2]https://en.wikipedia.org/wiki/Donald_Knuth
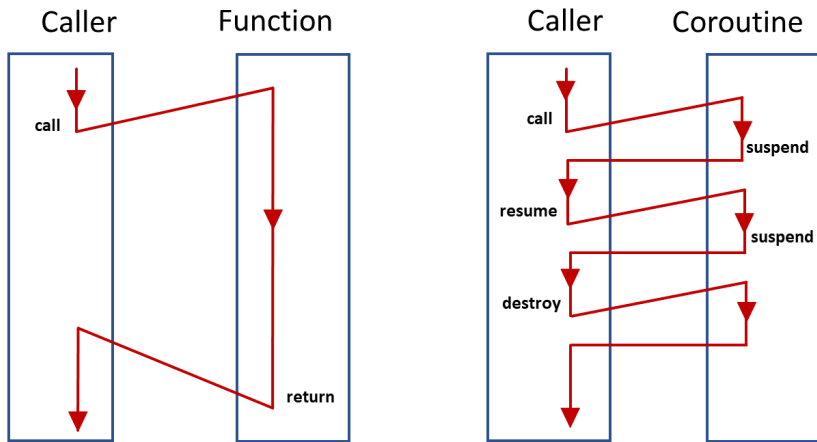
Functions versus Coroutines

While you can only call a function and return from it, you can call a coroutine, suspend and resume it, and destroy a suspended coroutine.

With the new keywords `co_await` and `co_yield`, C++20 extends the execution of C++ functions with two new concepts.

Thanks to `co_await` expression it is possible to suspend and resume the execution of the `expression`. If you use `co_await` expression in a function `func`, the call `auto getResult = func()` does not block if the result of the function is not available. Instead of resource-consuming blocking, you have resource-fri'\n'y waiting.

`co_yield` expression supports generator functions. The generator function returns a new value each time you call it. A generator function is a kind of data stream from which you can pick values. The data stream can be infinite. Therefore, we are at the center of lazy evaluation with C++.

5.1 A Generator Function

The following program is as simple as possible. The function `getNumbers` returns all integers from `begin` to `end`, incremented by `inc`. Value `begin` has to be smaller than `end`, and `inc` has to be positive.

A greedy generator function

```cpp
// greedyGenerator.cpp

#include <iostream>
#include <vector>

std::vector<int> getNumbers(int begin, int end, int inc = 1) {

    std::vector<int> numbers;
    for (int i = begin; i < end; i += inc) {
        numbers.push_back(i);
    }

    return numbers;

}

int main() {

    std::cout << '\n';

    const auto numbers= getNumbers(-10, 11);

    for (auto n: numbers) std::cout << n << " ";

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

    for (auto n: getNumbers(0, 101, 5)) std::cout << n << " ";

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

}
```

Of course, I am reinventing the wheel with getNumbers, because that job could be done with std::iota[3].

For completeness, here is the output.

───────────────────

[3]http://en.cppreference.com/w/cpp/algorithm/iota

A generator function

Two observations of the program `greedyGenerator.cpp` are essential. On the one hand, the vector `numbers` in line 8 always gets all values. This holds even if I'm only interested in the first 5 elements of a vector with 1000 elements. On the other hand, it's quite easy to transform the function `getNumbers` into a lazy generator. The following program is intentionally not complete. The definition of the `generator` is still missing.

A lazy generator function

```cpp
// lazyGenerator.cpp

#include <iostream>

generator<int> generatorForNumbers(int begin, int inc = 1) {

    for (int i = begin;; i += inc) {
        co_yield i;
    }

}

int main() {

    std::cout << '\n';

    const auto numbers = generatorForNumbers(-10);

    for (int i= 1; i <= 20; ++i) std::cout << numbers() << " ";

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

    for (auto n: generatorForNumbers(0, 5)) std::cout << n << " ";

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

}
```

While the function `getNumbers` in the file `greedyGenerator.cpp` returns a `std::vector<int>`, the

coroutine `generatorForNumbers` in `lazyGenerator.cpp` returns a generator. The generator `numbers` in line 17 or `generatorForNumbers(0, 5)` in line 23 returns a new number on request. The range-based for loop triggers the query. Precisely, the query of the coroutine returns the value `i` via `co_yield i` and immediately suspends its execution. If a new value is requested, the coroutine resumes its execution exactly at that place.

The expression `generatorForNumbers(0, 5)` in line 23 is a just-in-place use of a generator.

I want to stress one point explicitly. The coroutine `generatorForNumbers` creates an infinite data stream because the for loop in line 8 has no end condition. This is fine if I only ask for a finite number of values, such as in line 20. This does not hold for line 23, since there is no end condition. Therefore, the expression runs *forever*.

5.2 Characteristics

Coroutines have a few unique characteristics.

5.2.1 Typical Use Cases

Coroutines are the usual way to write event-driven applications[4], which can be simulations, games, servers, user interfaces, or even algorithms. Coroutines are also typically used for cooperative multitasking[5]. The key to cooperative multitasking is that each task takes as much time as it needs, but avoids sleeping or waiting, and instead allows some other task to run. Cooperative multitasking stands in contrast to pre-emptive multitasking, for which we have a scheduler that decides how long each task gets the CPU.

There are different kinds of coroutines.

5.2.2 Underlying Concepts

Coroutines in C++20 are asymmetric, first-class, and stackless.

The workflow of an **asymmetric** coroutine goes back to the caller. This does not hold for a symmetric coroutine. A symmetric coroutine can delegate its workflow to another coroutine.

First-class coroutines are similar to first-class functions, since coroutines behave like data. Behaving like data means that you can use them as arguments to or return values from functions, or store them in a variable.

A **stackless** coroutine can suspend and resume the top-level coroutine. The execution of the coroutine and the yielding from the coroutine comes back to the caller. The coroutine stores its state for resumption separate from the stack. Stackless coroutines are often called resumable functions.

---

[4]https://en.wikipedia.org/wiki/Event-driven_programming
[5]https://en.wikipedia.org/wiki/Computer_multitasking

5.2.3 Design Goals

Gor Nishanov describes in proposal N4402[6] the design goals of coroutines.

Coroutines should

- be highly scalable (to billions of concurrent coroutines)
- have highly efficient resume and suspend operations comparable in cost to the overhead of a function
- seamlessly interact with existing facilities with no overhead
- have open-ended coroutine machinery allowing library designers to develop coroutine libraries exposing various high-level semantics such as generators, goroutines[7], tasks and more
- usable in environments where exceptions are forbidden or not available

Due to the design goals of scalability and seamless interaction with existing facilities, the coroutines are stackless. In contrast, a stackful coroutine reserves a default stack of 1MB on Windows, and 2MB on Linux.

There are four ways for a function to become a coroutine.

5.2.4 Becoming a Coroutine

A function becomes a coroutine if it uses

- `co_return`, or
- `co_await`, or
- `co_yield`, or a
- `co_await` expression in a range-based for loop.

---

[6]https://isocpp.org/files/papers/N4402.pdf
[7]https://tour.golang.org/concurrency/1

⚠ Distinguish Between the Coroutine Factory and the Coroutine Object

The term coroutine is often used for two different aspects of coroutines: the function invoking `co_return`, `co_await`, or `co_yield`, and the coroutine object. Using one term for two different coroutine aspects may puzzle you (such as it did me). Let me clarify both terms.

A simple coroutine producing 2021

```cpp
MyFuture<int> createFuture() {
    co_return 2021;
}

int main() {

    auto fut = createFuture();
    std::cout << "fut.get(): " << fut.get() << '\n';

}
```

This straightforward example has a function `createFuture` and returns an object of type `MyFuture<int>`. Both are called coroutines. To be specific, the function `createFuture` is a coroutine factory that returns a coroutine object. The coroutine object is a resumable object that implements the framework to model a specific behavior. I present in the section `co_return` the implementation and the use of this straightforward coroutine.

5.2.4.1 Restrictions

Coroutines cannot have `return` statements or placeholder return types. This holds for unconstrained placeholders (`auto`), and constrained placeholders (concepts).

Additionally, functions having variadic arguments[8], `constexpr` functions, `consteval` functions, constructors, destructors, and the main function cannot be coroutines.

5.3 The Framework

The framework for implementing coroutines consists of more than 20 functions, some of which you must implement and some of which you may overwrite. Therefore, you can tailor the coroutine to your needs.

A coroutine is associated with three parts: the promise object, the coroutine handle, and the coroutine frame. The client gets the coroutine handle to interact with the promise object, which keeps its state in the coroutine frame.

---

[8]https://en.cppreference.com/w/cpp/language/variadic_arguments

5.3.1 Promise Object

The promise object is manipulated from inside the coroutine, and it delivers its result or exception via the promise object.

The promise object must support the following interface.

Promise object

Member Function	Description
Default constructor	A promise must be default constructible.
`initial_suspend()`	Determines if the coroutine suspends before it runs.
`final_suspend noexcept()`	Determines if the coroutine suspends before it ends.
`unhandled_exception()`	Called when an exception happens.
`get_return_object()`	Returns the coroutine object (resumable object).
`return_value(val)`	Is invoked by `co_return val`.
`return_void()`	Is invoked by `co_return`.
`yield_value(val)`	Is invoked by `co_yield val`.

The compiler automatically invokes these functions during its execution of the coroutine. The section workflow presents this workflow in detail.

The function `get_return_object` returns a resumable object that the client uses to interact with the coroutine. A promise needs at least one of the member functions `return_value`, `return_void`, or `yield_value`. You don't need to define the member functions `return_value` or `return_void` if your coroutine never ends.

The three functions `yield_value`, `initial_suspend`, and `final_suspend` return awaitables. An Awaitable is something that you can await on. The awaitable determines if the coroutine pauses or not.

5.3.2 Coroutine Handle

The coroutine handle is a non-owning handle to resume or destroy the coroutine frame from the outside. The coroutine handle is part of the resumable function.

The following code snippet shows a simple `Generator` having a coroutine handle `coro`.

A coroutine handle

```
1    template<typename T>
2    struct Generator {
3
4        struct promise_type;
5        using handle_type = std::coroutine_handle<promise_type>;
6
7        Generator(handle_type h): coro(h) {}
8        handle_type coro;
9
10       ~Generator() {
11           if ( coro ) coro.destroy();
12       }
13       T getValue() {
14           return coro.promise().current_value;
15       }
16       bool next() {
17           coro.resume();
18           return not coro.done();
19       }
20   ...
21   }
```

The constructor (line 7) gets the coroutine handle to the promise that has type `std::coroutine_-handle<promise_type>`[9]. The member functions `next` (line 16) and `getValue` (line 13) allow a client to resume the promise (`gen.next()`) or ask for its value (`gen.getValue()`) using the coroutine handle.

Invoking a coroutine

```
Generator<int> coroutineFactory(); // function that returns a coroutine object

auto gen = coroutineFactory();
gen.next();
auto result = gen.getValue();
```

Internally, both functions trigger the coroutine handle `coro` (line 8) to

- resume the coroutine: `coro.resume()` (line 17) or `coro()`;
- destroy the coroutine: `coro.destroy()` (line 11);
- check the state of the coroutine: `coro` (line 11).

---

[9]https://en.cppreference.com/w/cpp/coroutine/coroutine_handle

The coroutine is automatically destroyed when its function body ends. The call `coro` only returns `true` at its final suspension point.

> ⚠️ **The resumable object requires an inner type `promise_type`**
>
> A resumable object such as `Generator` must have an inner type `promise_type`. Alternatively, you can specialize `std::coroutine_traits`[10] on `Generator` and define a public member `promise_type` in it: `std::coroutine_traits<Generator>`.

5.3.3 Coroutine Frame

The coroutine frame is an internal, typically heap-allocated state. It consists of the already mentioned promise object, the coroutine's copied parameters, the representation of the suspension points, local variables whose lifetime ends before the current suspension point, and local variables whose lifetime exceed the current suspension point.

Two requirements are necessary to optimize out the allocation of the coroutine:

1. The lifetime of the coroutine has to be nested inside the lifetime of the caller.
2. The caller of the coroutine knows the size of the coroutine frame.

The crucial abstractions in the coroutine framework are Awaitables and Awaiters.

5.4 Awaitables and Awaiters

The three functions of a promis object `prom` `yield_value`, `initial_suspend`, and `final_suspend` return awaitables.

5.4.1 Awaitables

An Awaitable is something you can await on. The awaitable determines if the coroutine pauses or not.

Essentially, the compiler generates the following function calls using the promise `prom` and the `co_-await` operator.

---

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

Compiler-generated function calls

Call	Compiler generated call
Start coroutine execution	`co_await prom.initial_suspend()`
`co_yield value`	`co_await prom.yield_value(value)`
`co_return value`	`co_await prom.return_value(value)`
End coroutine execution	`co_await prom.final_suspend()`

The `co_await` operator needs an awaitable as argument. The awaitable is converted into an awaiter.

5.4.2 The Concept Awaiter

The concept Awaiter requires three functions.

The concept Awaiter

Function	Description
`await_ready`	Indicates if the result is ready. When it returns `false`, `await_suspend` is called.
`await_suspend`	Schedule the coroutine for resumption or destruction.
`await_resume`	Provides the result for the `co_await exp` expression.

The C++20 standard already defines two basic awaitables: `std::suspend_always`, and `std::suspend_never`.

5.4.3 `std::suspend_always` **and** `std::suspend_never`

As its name suggests, the Awaitable `suspend_always` always suspends. Therefore, the call `await_ready` returns `false`.

The Awaitable `std::suspend_always`

```
struct suspend_always {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(std::coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

The opposite holds for suspend_never. It never suspends and, hence, the call await_ready returns true.

The Awaitable `std::suspend_never`

```
struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(std::coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

The awaitables std::suspend_always and std::suspend_never are the basic building blocks for functions, such as initial_suspend and final_suspend. Both functions are automatically executed when the coroutine is exected: initial_suspend at the beginning and final_suspend at the end end of the coroutine.

5.4.4 `initial_suspend`

When the member function initial_suspend returns std::suspend_always, the coroutine suspends at its beginning. When returning std::suspend_never, the coroutine does not pause.

- A lazy coroutine that pauses immediately

A lazy coroutine

```
std::suspend_always initial_suspend() {
    return {};
}
```

- An eager coroutine that runs immediately

A eager coroutine

```
std::suspend_never initial_suspend() {
    return {};
}
```

5.4.5 `final_suspend`

When the member function `final_suspend` returns `std::suspend_always`, the coroutine suspends at its end. When returning `std::suspend_never`, the coroutine does not pause.

- A lazy coroutine that pauses at its end

A lazy coroutine that finally pauses

```
std::suspend_always final_suspend() noexcept() {
    return {};
}
```

- An eager coroutine that doesn't pause at its end

An eager coroutine that doesn't pause

```
std::suspend_never final_suspend() noexcept {
    return {};
}
```

So far, we have only Awaitables, but we need something to await for. Let me fill the gap and write about Awaiters.

5.4.6 Awaiter

There are essentially two ways to get an Awaiter.

- A `co_await` operator is defined.
- The Awaitable becomes the Awaiter.

Remember, when `co_await` expression is invoked, the expression is an Awaitable. Further, an expression is a call on the promise object (Awaitable): `prom.yield_value(value)`, `prom.initial_-suspend()`, or `prom.final_suspend()`.

Now, the compiler performs the following lookup rule to get an Awaiter:

1. It looks for the `co_await` operator on the promise object and returns an Awaiter:

   ```
   awaiter = awaitable.operator co_await();
   ```

2. It looks for a freestanding `co_await` operator and returns an Awaiter:

```
awaiter = operator co_await(awaitable);
```

3. If there is no `co_await` operator defined, the Awaitable becomes the Awaiter:

```
awaiter = awaitable;
```

`awaiter = awaitable`

When you study my coroutine implementations in this chapter, you may notice that I use most of the time that an Awaitable implicitly becomes an Awaiter. Only the example to thread synchronization uses the `co_await` operator to get the Awaiter.

After these static aspects of coroutines, I want to continue with their dynamic aspects.

5.5 The Workflows

The compiler transforms your coroutine and runs two workflows: the outer promise workflow and the inner awaiter workflow.

5.5.1 The Promise Workflow

When you use `co_yield`, `co_await`, or `co_return` in a function, the function becomes a coroutine, and the compiler transforms its body to something equivalent to the following lines.

The transformed coroutine

```
 1  {
 2      Promise prom;
 3      co_await prom.initial_suspend();
 4      try {
 5          ⟨function body having co_return, co_yield, or co_await⟩
 6      }
 7      catch (...) {
 8          prom.unhandled_exception();
 9      }
10  FinalSuspend:
11      co_await prom.final_suspend();
12  }
```

The compiler automatically runs the transformed code using the functions of the promise object. In short, I call this workflow the promise workflow. Here are the main steps of this workflow.

- Coroutine begins execution

- allocates the coroutine frame if necessary
- copies all function parameters to the coroutine frame
- creates the prom object `prom` (line 2)
- calls `prom.get_return_object()` to create the coroutine handle, and keeps it in a local variable. The result of the call will be returned to the caller when the coroutine first suspends.
- calls `prom.initial_suspend()` and `co_awaits` its result. The promise type typically returns `suspend_never` for eagerly-started coroutines or `suspend_always` for lazily-started coroutines. (line 3)
- the body of the coroutine is executed when `co_await prom.initial_suspend()` resumes
- Coroutine reaches a suspension point
  - the return object (`prom.get_return_object()`) is returned to the caller which resumed the coroutine
- Coroutine reaches `co_return`
  - calls `prom.return_void()` for `co_return` or `co_return expression`, where `expression` has type `void`
  - calls `prom.return_value(expression)` for `co_return expression`, where `expression` has non-`void` type.
  - destroys all stack-created variables
  - calls `prom.final_suspend()` and `co_awaits` its result
- Coroutine is destroyed (by terminating via `co_return` an uncaught exception, or via the coroutine handle)
  - calls the destruction of the promise object
  - calls the destructor of the function parameters
  - frees the memory used by the coroutine frame
  - transfers control back to the caller

When a coroutine ends with an uncaught exception, the following happens:

- catches the exception and calls `prom.unhandled_exception()` from the catch block
- calls `prom.final_suspend()` and `co_awaits` the result (line 11)

When you use `co_await expr` in a coroutine, or the compiler implicitly invokes `co_await prom.initial_-suspend()`, `co_await prom.final.suspend()`, or `co_await prom.yield_value(value)`, a second, inner awaitable workflow starts.

5.5.2 The Awaiter Workflow

Using `co_await expr` causes the compiler to transform the code based on the functions `await_ready`, `await_suspend`, and `await_resume`. Consequently, I call the execution of the transformed code the awaiter workflow.

The compiler generates approximately the following code using the `awaiter`. For simplicity, I ignore exception handling and describe the workflow with comments.

The generated Awaiter Workflow

```
1    awaiter.await_ready() returns false:
2
3        suspend coroutine
4
5        awaiter.await_suspend(coroutineHandle) returns:
6
7            void:
8                awaiter.await_suspend(coroutineHandle);
9                coroutine keeps suspended
10               return to caller
11
12           bool:
13               bool result = awaiter.await_suspend(coroutineHandle);
14               if result:
15                   coroutine keep suspended
16                   return to caller
17               else:
18                   go to resumptionPoint
19
20           another coroutine handle:
21               auto anotherCoroutineHandle = awaiter.await_suspend(coroutineHandle);
22               anotherCoroutineHandle.resume();
23               return to caller
24
25   resumptionPoint:
26
27   return awaiter.await_resume();
```

The workflow is only executed if `awaiter.await_ready()` returns `false` (line 1). In case it returns `true`, the coroutine is ready and returns with the result of the call `awaiter.await_resume()` (line 27).

Let me assume that `awaiter.await_ready()` returns `false`. First, the coroutine is suspended (line 3), and immediately the return value of `awaiter.await_suspend()` is evaluated. The return type can be `void` (line 7), a boolean (line 12), or another coroutine handle (line 20), such as `anotherCoroutineHandle`. Depending on the return type, the program flow returns or another coroutine is executed.

<div align="center">

Return value of `awaiter.await_suspend()`

</div>

Type	Description
`void`	The coroutine keeps suspended and returns to the caller.
`bool`	`bool == true`: The coroutine keeps suspended and returns to the caller. `bool == false`: The coroutine is resumed and does not return to the caller.
`anotherCoroutineHandle`	The other coroutine is resumed and returns to the caller.

Whats happens in case an exception is thrown? It makes a difference if the exception occurs in `await_-read`, `await_suspend`, or `await_resume`.

- `await_ready`: The coroutine is not suspended, nor are the calls `await_suspend` or `await_resume` evaluated.

- `await_suspend`: The exception is caught, the coroutine is resumed, and the exception rethrown. `await_resume` is not called.

- `await_resume`: `await_ready` and `await_suspend` are evaluated and all values are returned. Of course, the call `await_resume` does not return a result.

Let me put theory into practice.

5.6 `co_return`

A coroutine uses **co_return** as its return statement.

5.6.1 A Future

Admittedly, the coroutine in the following program `eagerFuture.cpp` is the simplest coroutine I can imagine that still does something meaningful: it automatically stores the result of its invocation.

An eager future

```cpp
1  // eagerFuture.cpp
2
3  #include <coroutine>
4  #include <iostream>
5  #include <memory>
6
7  template<typename T>
8  struct MyFuture {
9      std::shared_ptr<T> value;
10     MyFuture(std::shared_ptr<T> p): value(p) {}
```

```
11        ~MyFuture() { }
12        T get() {
13            return *value;
14        }
15
16        struct promise_type {
17            std::shared_ptr<T> ptr = std::make_shared<T>();
18            ~promise_type() { }
19            MyFuture<T> get_return_object() {
20                return ptr;
21            }
22            void return_value(T v) {
23                *ptr = v;
24            }
25            std::suspend_never initial_suspend() {
26                return {};
27            }
28            std::suspend_never final_suspend() noexcept {
29                return {};
30            }
31            void unhandled_exception() {
32                std::exit(1);
33            }
34        };
35    };
36
37    MyFuture<int> createFuture() {
38        co_return 2021;
39    }
40
41    int main() {
42
43        std::cout << '\n';
44
45        auto fut = createFuture();
46        std::cout << "fut.get(): " << fut.get() << '\n';
47
48        std::cout << '\n';
49
50    }
```

MyFuture behaves as a future[11], which runs immediately. The call of the coroutine createFuture (line 45) returns the future, and the call fut.get (line 46) picks up the result of the associated promise.

---

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

There is one subtle difference to a future, the return value of the coroutine `createFuture` is available after its invocation. Due to the lifetime issues, the return value is managed by a `std::shared_ptr` (lines 9 and 17). The coroutine always uses `std::suspend_never` (lines 25, and 28) and, therefore, neither suspends before it runs nor after. This means the coroutine is executed when the function `createFuture` is invoked. The member function `get_return_object` (line 19) creates and stores the handle to the coroutine object, and `return_value` (lines 22) stores the result of the coroutine, which was provided by `co_return 2021` (line 38). The client invokes `fut.get` (line 46) and uses the future as a handle to the promise. The member function `get` returns the result to the client (line 13).

```
fut.get(): 2021
```

An eager future

You may think that it is not worth the effort of implementing a coroutine that behaves just like a function. You are right! However, this simple coroutine is an ideal starting point for writing various implementations of futures. Read more about Variations of Futures in chapter case studies.

5.7 `co_yield`

Thanks to `co_yield` you can implement a generator generating an infinite data stream from which you can successively query values. The return type of the generator `generator<int> generatorForNumbers(int begin, int inc= 1)` is `generator<int>`, where generator internally holds a special promise `p` such that a call `co_yield i` is equivalent to a call `co_await p.yield_value(i)`. Statement `co_yield i` can be called an arbitrary number of times. Immediately after each call, the execution of the coroutine is suspended.

5.7.1 An Infinite Data Stream

The program `infiniteDataStream.cpp` produces an infinite data stream. The coroutine `getNext` uses `co_yield` to create a data stream that starts at `start` and gives on request the next value, incremented by `step`.

An infinite data stream

```cpp
1  // infiniteDataStream.cpp
2
3  #include <coroutine>
4  #include <memory>
5  #include <iostream>
6
7  template<typename T>
8  struct Generator {
9
```

```
10        struct promise_type;
11        using handle_type = std::coroutine_handle<promise_type>;
12
13        Generator(handle_type h): coro(h) {}
14        handle_type coro;
15
16        ~Generator() {
17            if ( coro ) coro.destroy();
18        }
19        Generator(const Generator&) = delete;
20        Generator& operator = (const Generator&) = delete;
21        Generator(Generator&& oth) noexcept : coro(oth.coro) {
22            oth.coro = nullptr;
23        }
24        Generator& operator = (Generator&& oth) noexcept {
25            coro = oth.coro;
26            oth.coro = nullptr;
27            return *this;
28        }
29        T getValue() {
30            return coro.promise().current_value;
31        }
32        bool next() {
33            coro.resume();
34            return not coro.done();
35        }
36        struct promise_type {
37            promise_type()  = default;
38
39            ~promise_type() = default;
40
41            auto initial_suspend() {
42                return std::suspend_always{};
43            }
44            auto final_suspend() noexcept {
45                return std::suspend_always{};
46            }
47            auto get_return_object() {
48                return Generator{handle_type::from_promise(*this)};
49            }
50            auto return_void() {
51                return std::suspend_never{};
52            }
53
54            auto yield_value(const T value) {
```

```
55                current_value = value;
56                return std::suspend_always{};
57            }
58            void unhandled_exception() {
59                std::exit(1);
60            }
61            T current_value;
62        };
63
64    };
65
66    Generator<int> getNext(int start = 0, int step = 1) {
67        auto value = start;
68        while (true) {
69            co_yield value;
70            value += step;
71        }
72    }
73
74    int main() {
75
76        std::cout << '\n';
77
78        std::cout << "getNext():";
79        auto gen = getNext();
80        for (int i = 0; i <= 10; ++i) {
81            gen.next();
82            std::cout << " " << gen.getValue();
83        }
84
85        std::cout << "\n\n";
86
87        std::cout << "getNext(100, -10):";
88        auto gen2 = getNext(100, -10);
89        for (int i = 0; i <= 20; ++i) {
90            gen2.next();
91            std::cout << " " << gen2.getValue();
92        }
93
94        std::cout << '\n';
95
96    }
```
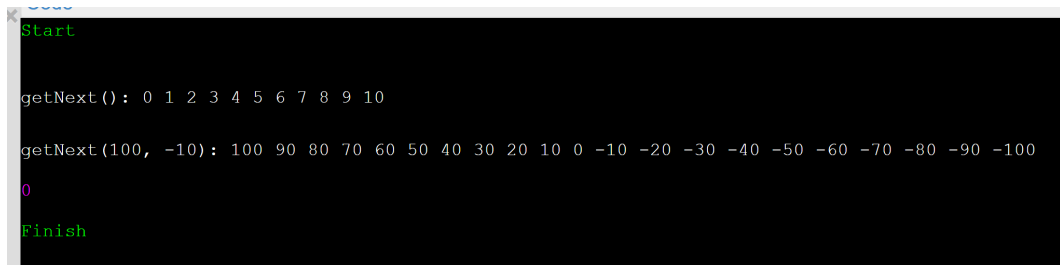
The main program creates two coroutines. The first one gen (line 79) returns the values from 0 to 10,

and the second one `gen2` (line 88) the values from 100 to -100. Before I dive into the workflow, thanks to the online compiler Wandbox[12], here is the output of the program.

An infinite data stream

The numbers in the program `infiniteDataStream.cpp` stand for the steps in the first iteration of the workflow.

1. creates the promise
2. calls `promise.get_return_object()` and keeps the result in a local variable
3. creates the generator
4. calls `promise.initial_suspend()`. The generator is lazy and, therefore, always suspends.
5. asks for the next value and returns if the generator is consumed
6. triggered by the `co_yield` call. The next value is available thereafter.
7. gets the next value

In additional iterations, only steps 5, 6, and 7 are performed.

Section Modification and Generalization of Threads in chapter case studies discusses further improvements and modifications of the generator `infiniteDataStream.cpp`.

5.8 `co_await`

`co_await` eventually causes the execution of the coroutine to be suspended or resumed. The expression `exp` in `co_await exp` has to be a so-called awaitable expression, i.e. which must implement a specific interface, consisting of the three functions `await_ready`, `await_suspend`, and `await_resume`.

A typical use case for `co_await` is a server that waits for events.

---

[12]https://wandbox.org/

A blocking server

```
1  Acceptor acceptor{443};
2  while (true) {
3      Socket socket = acceptor.accept();          // blocking
4      auto request = socket.read();               // blocking
5      auto response = handleRequest(request);
6      socket.write(response);                     // blocking
7  }
```

The server is quite simple because it sequentially answers each request in the same thread. The server listens on port 443 (line 1), accepts the connection (line 3), reads the incoming data from the client (line 4), and writes its answer to the client (line 6). The calls in lines 3, 4, and 6 are blocking.

Thanks to `co_await`, the blocking calls can now be suspended and resumed.

A waiting server

```
1  Acceptor acceptor{443};
2  while (true) {
3      Socket socket = co_await acceptor.accept();
4      auto request = co_await socket.read();
5      auto response = handleRequest(request);
6      co_await socket.write(response);
7  }
```

Before I present the challenging example of thread synchronization with coroutines, I want to start with something straightforward: starting a job on request.

5.8.1 Starting a Job on Request

The coroutine in the following example is as simple as it can be. It awaits on the predefined Awaitable `std::suspend_never()`.

Starting a job on request

```
1  // startJob.cpp
2
3  #include <coroutine>
4  #include <iostream>
5
6  struct Job {
7      struct promise_type;
8      using handle_type = std::coroutine_handle<promise_type>;
9      handle_type coro;
```

```
10          Job(handle_type h): coro(h){}
11          ~Job() {
12              if ( coro ) coro.destroy();
13          }
14          void start() {
15              coro.resume();
16          }
17
18
19          struct promise_type {
20              auto get_return_object() {
21                  return Job{handle_type::from_promise(*this)};
22              }
23              std::suspend_always initial_suspend() {
24                  std::cout << "    Preparing job" << '\n';
25                  return {};
26              }
27              std::suspend_always final_suspend() noexcept {
28                  std::cout << "    Performing job" << '\n';
29                  return {};
30              }
31              void return_void() {}
32              void unhandled_exception() {}
33
34          };
35      };
36
37  Job prepareJob() {
38      co_await std::suspend_never();
39  }
40
41  int main() {
42
43      std::cout << "Before job" << '\n';
44
45      auto job = prepareJob();
46      job.start();
47
48      std::cout << "After job" << '\n';
49
50  }
```

You may think that the coroutine `prepareJob` (line 37) is meaningless because the Awaitable never suspends. No! The function `prepareJob` is at least a coroutine factory using `co_await` (line 38) and

returning a coroutine object. The function call `prepareJob()` in line 45 creates the coroutine object of type `Job`. When you study the data type `Job`, you recognize that the coroutine object is immediately suspended, because the member function of the promise returns the Awaitable `std::suspend_always` (line 23). This is exactly the reason why the function call `job.start` (line 46) is necessary to resume the coroutine (line 15). The member function `final_suspend` also returns `std::suspend_always` (line 27).

```
Before job
    Preparing job
    Performing job
After job
```

Starting a Job on Request

In the case studies' section various job flows, I use the program `startJob` as a starting point for further experiments.

5.8.2 Thread Synchronization

It's typical for threads to synchronize themselves. One thread prepares a work package another thread awaits. Condition variables[13], promises and futures[14], and also an atomic boolean[15] can be used to create a sender-receiver workflow. Thanks to coroutines, thread synchronization is quite easy, without the inherent risks of condition variables, such as spurious wakeups and lost wakeups.

Thread Synchronization

```cpp
1   // senderReceiver.cpp
2
3   #include <coroutine>
4   #include <chrono>
5   #include <iostream>
6   #include <functional>
7   #include <string>
8   #include <stdexcept>
9   #include <atomic>
10  #include <thread>
11
12  class Event {
13   public:
14
15      Event() = default;
16
```

[13]https://en.cppreference.com/w/cpp/thread/condition_variable
[14]https://en.cppreference.com/w/cpp/thread
[15]https://en.cppreference.com/w/cpp/atomic/atomic

```cpp
17        Event(const Event&) = delete;
18        Event(Event&&) = delete;
19        Event& operator=(const Event&) = delete;
20        Event& operator=(Event&&) = delete;
21
22        class Awaiter;
23        Awaiter operator co_await() const noexcept;
24
25        void notify() noexcept;
26
27    private:
28
29        friend class Awaiter;
30
31        mutable std::atomic<void*> suspendedWaiter{nullptr};
32        mutable std::atomic<bool> notified{false};
33
34    };
35
36    class Event::Awaiter {
37    public:
38        Awaiter(const Event& eve): event(eve) {}
39
40        bool await_ready() const;
41        bool await_suspend(std::coroutine_handle<> corHandle) noexcept;
42        void await_resume() noexcept {}
43
44    private:
45        friend class Event;
46
47        const Event& event;
48        std::coroutine_handle<> coroutineHandle;
49    };
50
51    bool Event::Awaiter::await_ready() const {
52
53        // allow at most one waiter
54        if (event.suspendedWaiter.load() != nullptr){
55            throw std::runtime_error("More than one waiter is not valid");
56        }
57
58        // event.notified == false; suspends the coroutine
59        // event.notified == true; the coroutine is executed like a normal function
60        return event.notified;
61    }
```

```
62
63    bool Event::Awaiter::await_suspend(std::coroutine_handle<> corHandle) noexcept {
64            coroutineHandle = corHandle;
65
66            const Event& ev = event;
67            ev.suspendedWaiter.store(this);
68
69            if (ev.notified) {
70                    void* thisPtr = this;
71
72                    if (ev.suspendedWaiter.compare_exchange_strong(thisPtr, nullptr)) {
73                            return false;
74                    }
75            }
76
77            return true;
78    }
79
80    void Event::notify() noexcept {
81            notified = true;
82
83            void* waiter = suspendedWaiter.load();
84
85            if (waiter != nullptr && suspendedWaiter.compare_exchange_strong(waiter, nullptr)) {
86                    static_cast<Awaiter*>(waiter)->coroutineHandle.resume();
87            }
88    }
89
90    Event::Awaiter Event::operator co_await() const noexcept {
91        return Awaiter{ *this };
92    }
93
94    struct Task {
95        struct promise_type {
96            Task get_return_object() { return {}; }
97            std::suspend_never initial_suspend() { return {}; }
98            std::suspend_never final_suspend() noexcept { return {}; }
99            void return_void() {}
100           void unhandled_exception() {}
101       };
102   };
103
104   Task receiver(Event& event) {
105       auto start = std::chrono::high_resolution_clock::now();
106       co_await event;
```

```
107        std::cout << "Got the notification! " << '\n';
108        auto end = std::chrono::high_resolution_clock::now();
109        std::chrono::duration<double> elapsed = end - start;
110        std::cout << "Waited " << elapsed.count() << " seconds." << '\n';
111    }
112
113    using namespace std::chrono_literals;
114
115    int main() {
116
117        std::cout << '\n';
118
119        std::cout << "Notification before waiting" << '\n';
120        Event event1{};
121        auto senderThread1 = std::thread([&event1]{ event1.notify(); });  // Notification
122        auto receiverThread1 = std::thread(receiver, std::ref(event1));
123
124        receiverThread1.join();
125        senderThread1.join();
126
127        std::cout << '\n';
128
129        std::cout << "Notification after 2 seconds waiting" << '\n';
130        Event event2{};
131        auto receiverThread2 = std::thread(receiver, std::ref(event2));
132        auto senderThread2 = std::thread([&event2]{
133            std::this_thread::sleep_for(2s);
134            event2.notify();                                              // Notification
135        });
136
137        receiverThread2.join();
138        senderThread2.join();
139
140        std::cout << '\n';
141
142    }
```

From the user's perspective, thread synchronization with coroutines is straightforward. Let's have a look at the program `senderReceiver.cpp`. The threads `senderThread1` (line 121) and `senderThread2` (line 132) each uses an event to send its notification,respectively, in lines 121 and 134. The function `receiver` in lines 104 - 111 is the coroutine, which is executed in threads `receiverThread1` (line 122) and `receiverThread2` (line 132). I measured the time between the beginning and the end of the coroutine and displayed it. This number shows how long the coroutine waits. The following screenshot shows the output of the program.

Thread synchronization

If you compare the class `Generator` in the infinite data stream with the class `Event` in this example, there is a subtle difference. In the first case, the `Generator` is the awaitable and the awaiter; in the second case, the `Event` uses the `operator co_await` to return the awaiter. This separation of concerns into the Awaitable and the awaiter improves the structure of the code.

The output displays that the execution of the second coroutine takes about two seconds. The reason is that the `event1` sends its notification (line 121) before the coroutine is suspended, but the `event2` sends its notification after a time duration of 2 seconds (line 134).

Now, I put the implementer's hat on. The workflow of the coroutine is quite challenging to grasp. The class `Event` has two interesting members: `suspendedWaiter` and `notified`. Variable `suspendedWaiter` in line 31 holds the waiter for the signal, and `notified` in line 32 has the state of the notification.

In my explanation of both workflows, I assume in the first case (first workflow) that the event notification happens before the coroutine awaits the events. For the second case (second workflow), I assume it is the other way around.

Let's first look at `event1` and the first workflow. Here, `event1` sends its notification before `receiverThread1` is started. The invocation `event1` (line 121) triggers the method `notify` (lines 80 to 88). First the notification flag is set and then, the call `void* waiter = suspendedWaiter.load()` loads the potential waiter. In this case, the `waiter` is a `nullptr` because it was not set before. This means the following `resume` call on the `waiter` in line 86 is not executed. The subsequentially performed function `await_ready` (lines 51 - 61) checks first if there is more than one `waiter`. In this case, I throw a `std::runtime` exception. The crucial part of this method is the return value. `event.notification` was already set to `true` in the `notify` method. `true` means, in this case, that the coroutine is not suspended and executes such as a normal function.

In the second workflow, the `co_await event2` call happens before `event2` sends its notification. `co_-await event2` triggers the call `await_ready` (line 51). The big difference with the first workflow is that

event.notified is false. This false value causes the suspension of the coroutine. Technically, method await_suspend (lines 63 - 78) is executed. await_suspend gets the coroutine handle corHandle and stores it for later invocation in the variable coroutineHandle (line 64). Of course, later invocation means resumption. Second, the waiter is stored in the variable suspendedWaiter. When later event2.notify triggers its notification, method notify (line 80) is executed. The difference with the first workflow is that the condition waiter != nullptr evaluates to true. The result is that the waiter uses the coroutineHandle to resume the coroutine.

Distilled Information

- Coroutines are generalized functions that can pause and resume their execution while keeping their state.
- With C++20, we don't get concrete coroutines, but a framework for implementing coroutines. This framework consists of more than 20 functions that you partially have to implement and partially could overwrite.
- With the new keywords co_await and co_yield, C++20 extends the execution of C++ functions with two new concepts.
- Thanks to co_await expression it is possible to suspend and resume the execution of the expression. If you use co_await expression in a function func, the call auto getResult = func() does not block if the function's result is not available. Instead of resource-consuming blocking, you have resource-fri'\n'y waiting.
- co_yield empowers you to write infinite data streams.

6. Case Studies

Cippi applies here knowledge

After providing the theory on the memory model, the the multithreading interface, and the brand-new C++20 standard, I now apply the theory in practice. Additionally, the case studies calculating the sum of a vector, thread-safe initialization of a singleton, and [fast synchronization of threads] (#chapterXXXFastSSSSynchronizationSSSofSSSThreads) give you performance numbers.

🔑 The Reference PCs

You should take the performance numbers with a **grain of salt**. I'm not interested in the exact number for each variation of the algorithms on Linux and Windows. I'm more interested in getting a gut feeling of which algorithms may work and which algorithms may not work. I'm not comparing the absolute numbers of my Linux desktop with the numbers on my Windows laptop, but I'm interested to know if some algorithms work better on Linux or Windows.

6.1 Calculating the Sum of a Vector

What is the fastest way to add the elements of a std::vector? To get the answer, I fill a std::vector with one hundred million arbitrary but uniformly distributed[1] numbers between 1 and 10. The task is to calculate the sum of the numbers in various ways. I use the performance of a single-threaded addition as the reference execution time. I discuss atomics, locks, thread-local data and tasks.

Let's start with the single-threaded scenario.

6.1.1 Single Threaded addition of a Vector

The straightforward strategy is it to add the numbers in a range-based for loop.

6.1.1.1 Range-based for Loop

The summation takes place in line 27.

Summation of a vector in a range-based for loop

```cpp
// calculateWithLoop.cpp

#include <chrono>
#include <iostream>
#include <random>
#include <vector>

constexpr long long size = 100000000;

int main(){

  std::cout << '\n';

  std::vector<int> randValues;
  randValues.reserve(size);

  // random values
  std::random_device seed;
  std::mt19937 engine(seed());
  std::uniform_int_distribution<> uniformDist(1, 10);
  for (long long i = 0 ; i < size ; ++i)
       randValues.push_back(uniformDist(engine));

  const auto sta = std::chrono::steady_clock::now();
```

[1]https://en.wikipedia.org/wiki/Discrete_uniform_distribution
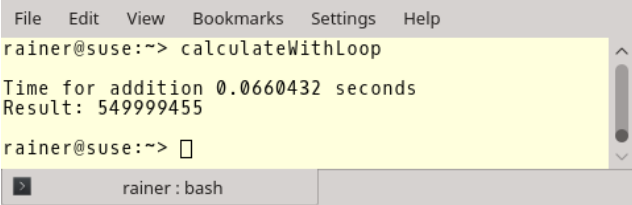
```
25
26     unsigned long long sum = {};
27     for (auto n: randValues) sum += n;
28
29     const std::chrono::duration<double> dur =
30           std::chrono::steady_clock::now() - sta;
31
32     std::cout << "Time for addition " << dur.count()
33             << " seconds" << '\n';
34     std::cout << "Result: " << sum << '\n';
35
36     std::cout << '\n';
37
38 }
```

How fast are my computers?
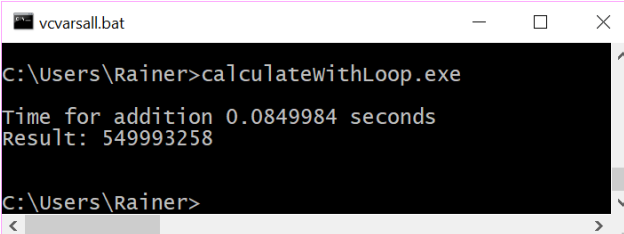
Explicit summation on Linux

Explicit summation on Windows

You should not use loops explicitly. Most of the time you can use an algorithm from the Standard Template Library.

6.1.1.2 Summation with `std::accumulate`

`std::accumulate` is the right way to calculate the sum of a vector. For the sake of simplicity, I show the application of `std::accumulate`. The entire source file can be found in the resources for this book.

Summation of a vector with `std::accumulate`

```cpp
// calculateWithStd.cpp

...

const unsigned long long sum = std::accumulate(randValues.begin(),
                                               randValues.end(), 0);

...
```
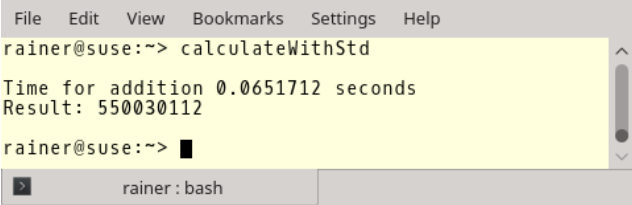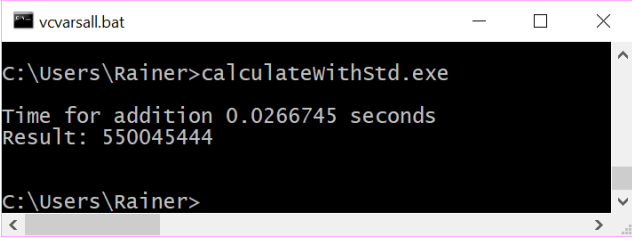
On Linux, the performance of `std::accumulate` is roughly the same as the performance of the range-based for-loop. However, using `std::accumulate` on Windows makes a big difference, and performs much better than an explicit for-loop.

Summation with `std::acccumulate` on Linux

Summation with `std::acccumulate` on Windows

Now we have our reference timings. Let me run two additional single-threaded scenarios. One with a lock and the other with an atomic. Why? We get the performance numbers indicating how expensive the protection by a lock or an atomic is when there is no contention.

6.1.1.3 Protection with a Lock

If I protect access to the summation variable with a lock, I get the answers to two questions.

1. How expensive is the synchronization of a lock without contention?
2. How fast can a lock be in the optimal case?

I only show the application of `std:lock_guard`. The entire source file is part of the resources for this book.

Summation of a vector by using a lock for the summation variable
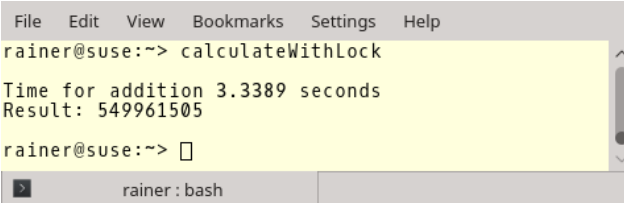
```cpp
// calculateWithLock.cpp

...

std::mutex myMutex;

for (auto i: randValues){
  std::lock_guard<std::mutex> myLockGuard(myMutex);
  sum += i;
}

...
```
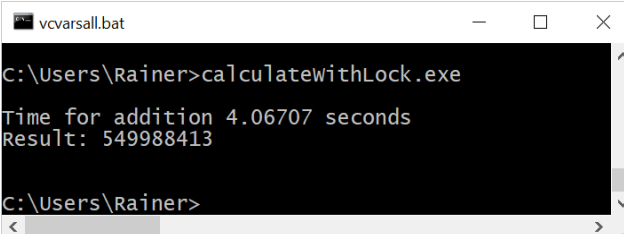
The execution times are as expected: the access to the protected variable sum is slower.

Single-threaded summation with std::acccumulate on Linux using a lock

Single-threaded summation with std::acccumulate on Windows using a lock

Using a std::lock_guard without contention is about 50 - 150 times slower than using std::accumulate. Let's finally get to atomics.

6.1.1.4 Protection with Atomics

Accordingly, I have the same questions for atomics that I had for locks.

1. How expensive is the synchronization of an atomic?
2. How fast can an atomic be if there is no contention?

I have an additional question; what is the performance difference of an atomic compared to a lock?

Summation of a vector by using an atomic as summation variable

```cpp
1   // calculateWithAtomic.cpp
2
3   #include <atomic>
4   #include <chrono>
5   #include <iostream>
6   #include <numeric>
7   #include <random>
8   #include <vector>
9
10  constexpr long long size = 100000000;
11
12  int main(){
13
14      std::cout << '\n';
15
16      std::vector<int> randValues;
17      randValues.reserve(size);
18
19      // random values
20      std::random_device seed;
21      std::mt19937 engine(seed());
22      std::uniform_int_distribution<> uniformDist(1, 10);
23      for (long long i = 0 ; i < size ; ++i)
24          randValues.push_back(uniformDist(engine));
25
26      std::atomic<unsigned long long> sum = {};
27      std::cout << std::boolalpha << "sum.is_lock_free(): "
28                << sum.is_lock_free() << '\n';
29      std::cout << '\n';
30
31      auto sta = std::chrono::steady_clock::now();
32
33      for (auto i: randValues) sum += i;
34
35      std::chrono::duration<double> dur = std::chrono::steady_clock::now() - sta;
36
37
38      std::cout << "Time for addition " << dur.count()
39                << " seconds" << '\n';
40      std::cout << "Result: " << sum << '\n';
41
42      std::cout << '\n';
43
44      sum = 0;
```
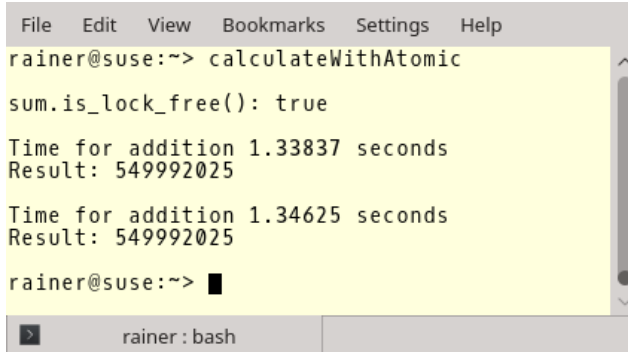
```
45    sta = std::chrono::steady_clock::now();
46
47    for (auto i: randValues) sum.fetch_add(i);
48
49    dur = std::chrono::steady_clock::now() - sta;
50    std::cout << "Time for addition " << dur.count()
51              << " seconds" << '\n';
52    std::cout << "Result: " << sum << '\n';
53
54    std::cout << '\n';
55
56  }
```

First, I check in line 28 if the atomic `sum` has a lock. That is crucial because otherwise, there would be no difference between using locks and atomics. On all mainstream platforms I know, atomics are lock-free. Second, I calculate the sum in two ways. I use in line 33 the `+=` operator, in line 47 the member function `fetch_add`. In the single-threaded case, both variants have comparable performance; however, for `fetch_add` I can explicitly specify the memory-ordering. More about that point in the next subsection.

Here are the results.

Summation with an atomic on Linux

Summation with an atomic on Windows

6.1.1.5 All Single-Threaded Numbers

I want to stress three points.

1. Atomics are 12 - 50 times slower on Linux and Windows than `std::accumulate` without synchronization.

2. Atomics are 2 - 3 times faster on Linux and Windows than locks.

3. `std::accumulate` seems to be highly optimized on Windows.

Before we look at the multithreaded scenarios, here is a table summarizing the results for single-threaded execution. The unit is seconds.

Performance of all single threaded summations

Operating System (Compiler)	Range-based for loop	std::accumulate	Locks	Atomics
Linux (GCC)	0.07	0.07	3.34	1.34
				1.33
Windows (cl.exe)	0.08	0.03	4.07	1.50
				1.61

6.1.2 Multithreaded Summation with a Shared Variable

You may have already guessed it. Using a shared variable for the summation with four threads is not optimal because the synchronization overhead outweighs the performance benefit. Let me show you the numbers.

The questions I want to answer are still the same.

1. What is the difference in performance between the summation using a lock and an atomic?

2. What is the difference in performance between single-threaded and multithreaded execution of `std::accumulate`?

6.1.2.1 Using a `std::lock_guard`

The simplest way to make the thread-safe summation is to use a `std::lock_guard`.

Multithreaded summation of a vector using a std::lock_guard

```cpp
// synchronizationWithLock.cpp

#include <chrono>
#include <iostream>
#include <mutex>
#include <random>
#include <thread>
#include <utility>
#include <vector>

constexpr long long size = 100000000;

constexpr long long fir =  25000000;
constexpr long long sec =  50000000;
constexpr long long thi =  75000000;
constexpr long long fou = 100000000;

std::mutex myMutex;

void sumUp(unsigned long long& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        std::lock_guard<std::mutex> myLock(myMutex);
        sum += val[it];
    }
}

int main(){

    std::cout << '\n';

    std::vector<int> randValues;
    randValues.reserve(size);

    std::mt19937 engine;
    std::uniform_int_distribution<> uniformDist(1,10);
    for (long long i = 0 ; i < size ; ++i)
```
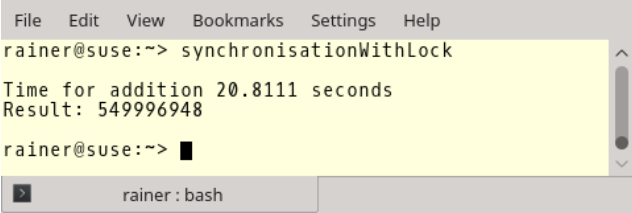
```
38          randValues.push_back(uniformDist(engine));
39
40      unsigned long long sum = 0;
41      const auto sta = std::chrono::steady_clock::now();
42
43      std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
44      std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
45      std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
46      std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);
47
48      t1.join();
49      t2.join();
50      t3.join();
51      t4.join();
52
53      std::chrono::duration<double> dur= std::chrono::steady_clock::now() - sta;
54      std::cout << "Time for addition " << dur.count()
55                << " seconds" << '\n';
56      std::cout << "Result: " << sum << '\n';
57
58      std::cout << '\n';
59
60  }
```

The program is easy to explain. The function sumUp (lines 20 - 26) is the work package each thread executes. sumUp gets the summation variable sum and the std::vector val by reference. beg and end specify the range of the summation. The std::lock_guard (line 23) is used to protect the shared sum. Each thread (lines 43 - 46) performs a quarter of the summation.
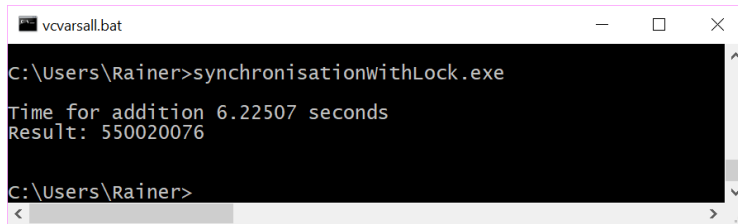
Here are the performance numbers of the program.

Summation with a shared variable on Linux

Summation with a shared variable on Windows

The program's bottleneck is the shared variable sum because a std::lock_guard heavily synchronizes it. One obvious solution comes immediately to mind: replace the heavyweight lock with a lightweight atomic.

Reduced Source Files

For the sake of simplicity, I show the function sumUp for the remainder of this subsection because all other parts of the program hardly change. For the complete examples, please see the resources for this book.

6.1.2.2 Using an atomic variable

Now, the summation variable sum is an atomic. That means I don't need the std::lock_guard anymore. Here is the modified sumUp function.

Summation of a vector by using an atomic

```cpp
// synchronizationWithAtomic.cpp

...

void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        sum += val[it];
    }
}

...
```
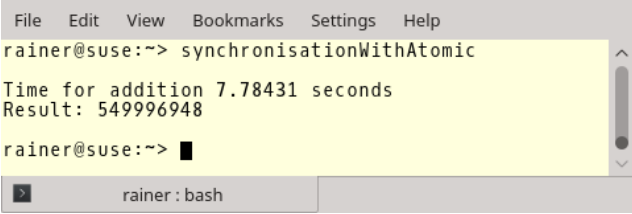
The performance numbers are pretty weird on my Windows laptop. The synchronization with std::lock_guard is more than twice as fast as the atomic version.
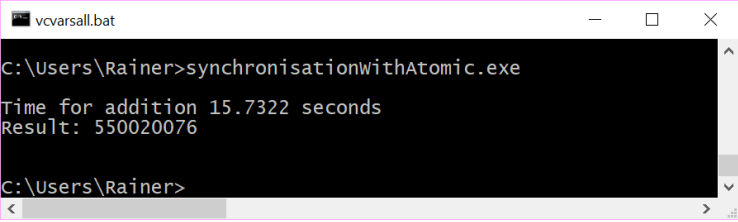
Summation with an atomic on Linux

Summation with an atomic on Windows

In addition to using the += operator on an atomic, you can use the `fetch_add` member function. Let's try it out.

6.1.2.3 Using the member function `fetch_add`

Once more. The modification of the source code is minimal. I have only changed the summation expression to `sum.fetch_add(val[it])`.

Summation of a vector by using the member function fetch_add

```
// synchronizationWithFetchAdd.cpp


...


void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
          unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        sum.fetch_add(val[it]);
    }
}


...
```
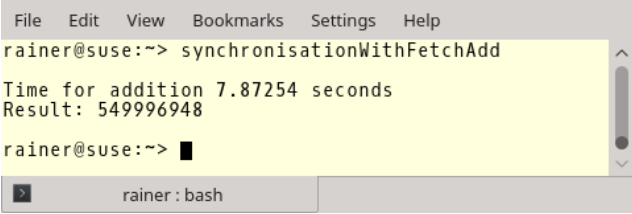
Now we have a similar performance as the previous example; there is little difference between the `operator +=` and `fetch_add`.
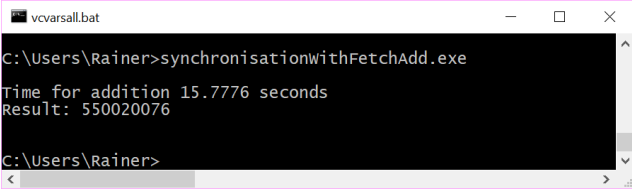
Summation with an atomic on Linux

Summation with an atomic on Windows

Although there is no performance difference between the += operation and the `fetch_add` member function on an atomic, `fetch_add` has an advantage; it allows me to weaken the memory-ordering explicitly and to apply relaxed semantic.

6.1.2.4 Using the member function `fetch_add` with relaxed semantic
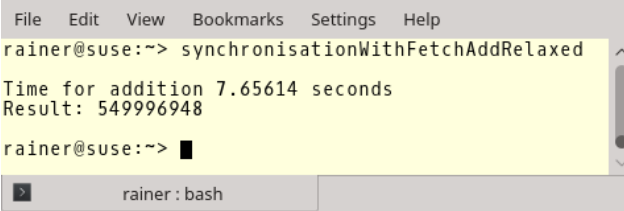
```
 1  // synchronizationWithFetchAddRelaxed.cpp
 2
 3  ...
 4
 5  void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
 6           unsigned long long beg, unsigned long long end){
 7      for (auto it = beg; it < end; ++it){
 8          sum.fetch_add(val[it], std::memory_order_relaxed);
 9      }
10  }
11
12  ...
```

The default behavior for atomics is sequential consistency. This statement is true for the addition and assignment of an atomic and, of course, for the `fetch_add` member function, but we can optimize even more. I adjust the memory-ordering in the summation expression to the relaxed semantic: `sum.fetch_-add(val[it], std::memory_order_relaxed)`. The relaxed semantic is the weakest memory-ordering; therefore, the endpoint of my optimization.
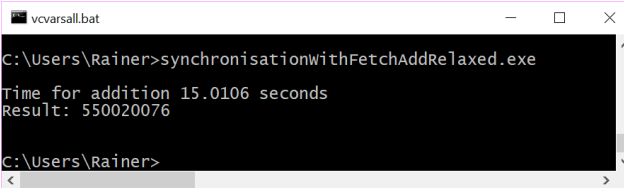
The relaxed semantic is acceptable in this use-case because we have two guarantees: each addition with `fetch_add` takes place atomically, and the threads synchronize with the join calls.

Because of the weakest memory model, we have the best performance.

Summation with a relaxed atomic on Linux

Summation with a relaxed atomic on Windows

6.1.2.5 All Multithreading Numbers with a Shared Variable

The units of the performance numbers are seconds.

Performance of all multi threaded summations

Operating System (Compiler)	std::lock_guard	atomic +=	fetch_add	fetch_add (relaxed)
Linux (GCC)	20.81	7.78	7.87	7.66
Windows (cl.exe)	6.22	15.73	15.78	15.01

The result of the performance numbers is not promising. Using a shared atomic variable with relaxed semantic and calculating the sum with four threads' help is about 100 times slower than using a single thread with the algorithm `std::accumulate`.

Let's combine the two previous strategies for adding the numbers. I use four threads and minimize the synchronization between the threads.

6.1.3 Thread-Local Summation

There are different ways to minimize synchronization. I can use local variables, thread-local data, and tasks.

6.1.3.1 Using a Local Variable

Since each thread can use a local summation variable, it can do its job without synchronization. The synchronization is only necessary to sum up the local variables. The summation of the local variables

is the critical section that must be protected. I can do this in various ways. A short remark: since only four additions take place, it doesn't matter from a performance perspective which synchronization I use. I use a `std::lock_guard`, an atomic with sequential consistency and relaxed semantic for the summation.

6.1.3.1.1 `std::lock_guard`

Summation with minimal synchronization using a `std::lock_guard`

```cpp
// localVariable.cpp

#include <mutex>
#include <chrono>
#include <iostream>
#include <random>
#include <thread>
#include <utility>
#include <vector>

constexpr long long size = 100000000;

constexpr long long fir =  25000000;
constexpr long long sec =  50000000;
constexpr long long thi =  75000000;
constexpr long long fou = 100000000;

std::mutex myMutex;

void sumUp(unsigned long long& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    unsigned long long tmpSum{};
    for (auto i = beg; i < end; ++i){
        tmpSum += val[i];
    }
    std::lock_guard<std::mutex> lockGuard(myMutex);
    sum += tmpSum;
}

int main(){

    std::cout << '\n';

    std::vector<int> randValues;
    randValues.reserve(size);

```
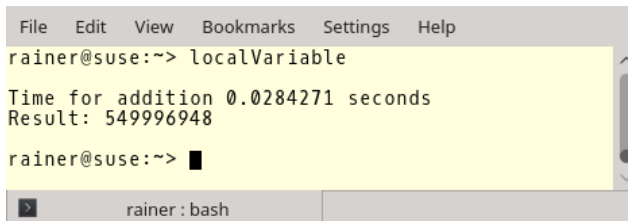
```
37      std::mt19937 engine;
38      std::uniform_int_distribution<> uniformDist(1, 10);
39      for (long long i = 0; i < size; ++i)
40          randValues.push_back(uniformDist(engine));
41
42      unsigned long long sum{};
43      const auto sta = std::chrono::system_clock::now();
44
45      std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
46      std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
47      std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
48      std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);
49
50      t1.join();
51      t2.join();
52      t3.join();
53      t4.join();
54
55      const std::chrono::duration<double> dur=
56          std::chrono::system_clock::now() - sta;
57
58
59      std::cout << "Time for addition " << dur.count()
60              << " seconds" << '\n';
61      std::cout << "Result: " << sum << '\n';
62
63      std::cout << '\n';
64
65  }
```

Lines 26 and 27 are interesting. These are the lines where the local summation result tmpSum is added to the global summation variable sum.
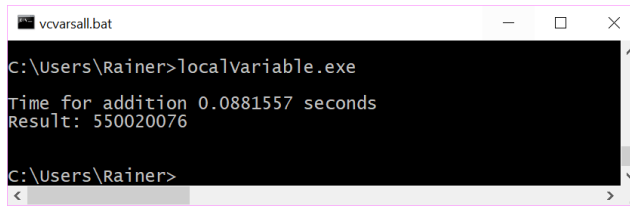

Summation with a local variable on Linux

Summation with a local variable on Windows

In the following two variations using a local variable, only the function sumUp changes; therefore, I display the function. For the entire program, please refer to the source files.

6.1.3.1.2 Using an Atomic Variable with Sequential Consistency

Let's replace the non-atomic global summation variable sum with an atomic.

Summation of a vector with minimal synchronization and an atomic

```cpp
1   // localVariableAtomic.cpp
2
3   ...
4
5   void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
6             unsigned long long beg, unsigned long long end){
7       unsigned int long long tmpSum{};
8       for (auto i = beg; i < end; ++i){
9           tmpSum += val[i];
10      }
11      sum+= tmpSum;
12  }
13
14  ...
```

Here are the performance numbers.

Summation with a local variable on Linux

Summation with a local variable on Windows

6.1.3.1.3 Using an Atomic Variable with Relaxed Semantic

We can do better. I use relaxed semantic now instead of the default memory-ordering. That's well defined because the only guarantee we need is that all summations occur and are atomic.
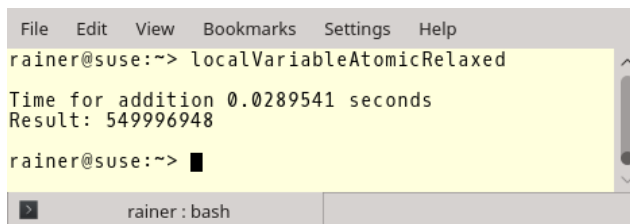
Summation of a vector with minimal synchronization and an atomic using relaxed semantic
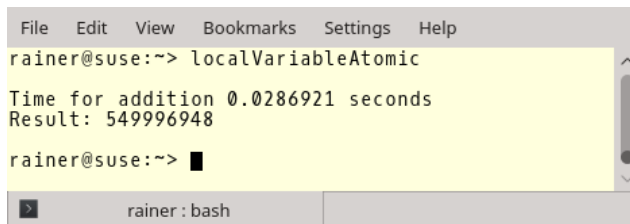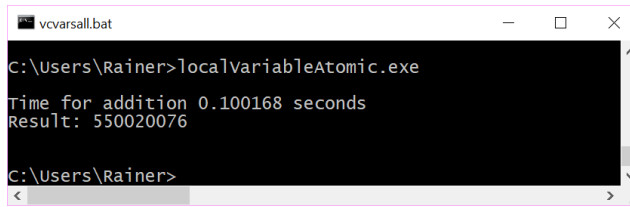
```
1   // localVariableAtomicRelaxed.cpp
2
3   void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
4             unsigned long long beg, unsigned long long end){
5       unsigned int long long tmpSum{};
6       for (auto i = beg; i < end; ++i){
7           tmpSum += val[i];
8       }
9       sum.fetch_add(tmpSum, std::memory_order_relaxed);
10  }
11
12  ...
```

As expected, it doesn't make any difference whether I use a std::lock_guard or an atomic with sequential consistency or relaxed semantic.

Summation with a local variable on Linux

Summation with a local variable on Windows

Thread-local data is a particular kind of local data. It is lifetime is bound to the scope of the thread and not to the scope of the function, such as for the variable tmpSum in this example.

6.1.3.2 Using Thread-Local Data

Thread-local data belongs to the thread in which it was created; it is only created when needed. Thread-local data is an ideal fit for the local summation variable tmpSum.

Summation of a vector with minimal synchronization using thread-local data

```cpp
1   // threadLocalSummation.cpp
2
3   #include <atomic>
4   #include <chrono>
5   #include <iostream>
6   #include <random>
7   #include <thread>
8   #include <utility>
9   #include <vector>
10
11  constexpr long long size = 100000000;
12
13  constexpr long long fir =  25000000;
14  constexpr long long sec =  50000000;
15  constexpr long long thi =  75000000;
16  constexpr long long fou = 100000000;
17
18  thread_local unsigned long long tmpSum = 0;
19
20  void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
21            unsigned long long beg, unsigned long long end){
22      for (auto i = beg; i < end; ++i){
23          tmpSum += val[i];
24      }
25      sum.fetch_add(tmpSum, std::memory_order_relaxed);
26  }
27
```

```
28    int main(){
29
30      std::cout << '\n';
31
32      std::vector<int> randValues;
33      randValues.reserve(size);
34
35      std::mt19937 engine;
36      std::uniform_int_distribution<> uniformDist(1, 10);
37      for (long long i = 0; i < size; ++i)
38          randValues.push_back(uniformDist(engine));
39
40      std::atomic<unsigned long long> sum{};
41      const auto sta = std::chrono::system_clock::now();
42
43      std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
44      std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
45      std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
46      std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);
47
48      t1.join();
49      t2.join();
50      t3.join();
51      t4.join();
52
53      const std::chrono::duration<double> dur=
54              std::chrono::system_clock::now() - sta;
55
56      std::cout << "Time for addition " << dur.count()
57                  << " seconds" << '\n';
58      std::cout << "Result: " << sum << '\n';
59
60      std::cout << '\n';
61
62    }
```

I declare in line 18 the thread-local variable tmpSum and use it for the addition in lines 23 and 25.

Here are the performance numbers using thread-local data.

Summation with a thread-local variable on Linux

Summation with a thread-local variable on Windows

In the last scenario, I use tasks.

6.1.3.3 Using Tasks

Using tasks, we can do the whole job without explicit synchronization. Each partial summation is performed in a separate thread, and the final summation takes place in the main thread.

Here is the program:

Summation of a vector with minimal synchronization using tasks

```cpp
// tasksSummation.cpp

#include <chrono>
#include <future>
#include <iostream>
#include <random>
#include <thread>
#include <utility>
#include <vector>

constexpr long long size = 100000000;

constexpr long long fir =  25000000;
constexpr long long sec =  50000000;
constexpr long long thi =  75000000;
constexpr long long fou = 100000000;

```

```
18    void sumUp(std::promise<unsigned long long>&& prom, const std::vector<int>& val,
19              unsigned long long beg, unsigned long long end){
20        unsigned long long sum={};
21        for (auto i = beg; i < end; ++i){
22            sum += val[i];
23        }
24        prom.set_value(sum);
25    }
26
27    int main(){
28
29      std::cout << '\n';
30
31      std::vector<int> randValues;
32      randValues.reserve(size);
33
34      std::mt19937 engine;
35      std::uniform_int_distribution<> uniformDist(1,10);
36      for (long long i = 0; i < size; ++i)
37          randValues.push_back(uniformDist(engine));
38
39      std::promise<unsigned long long> prom1;
40      std::promise<unsigned long long> prom2;
41      std::promise<unsigned long long> prom3;
42      std::promise<unsigned long long> prom4;
43
44      auto fut1= prom1.get_future();
45      auto fut2= prom2.get_future();
46      auto fut3= prom3.get_future();
47      auto fut4= prom4.get_future();
48
49      const auto sta = std::chrono::system_clock::now();
50
51      std::thread t1(sumUp, std::move(prom1), std::ref(randValues), 0, fir);
52      std::thread t2(sumUp, std::move(prom2), std::ref(randValues), fir, sec);
53      std::thread t3(sumUp, std::move(prom3), std::ref(randValues), sec, thi);
54      std::thread t4(sumUp, std::move(prom4), std::ref(randValues), thi, fou);
55
56      auto sum= fut1.get() + fut2.get() + fut3.get() + fut4.get();
57
58      std::chrono::duration<double> dur= std::chrono::system_clock::now() - sta;
59      std::cout << "Time for addition " << dur.count()
60              << " seconds" << '\n';
61      std::cout << "Result: " << sum << '\n';
62
```
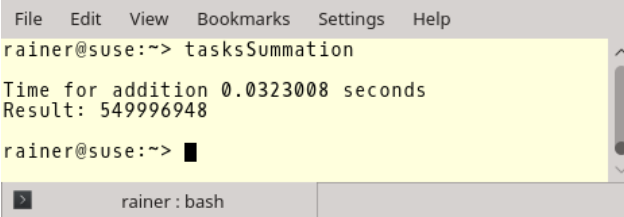
```
63    t1.join();
64    t2.join();
65    t3.join();
66    t4.join();
67
68    std::cout << '\n';
69
70  }
```

I define in lines 39 - 47 the four promises and the associated futures. In lines 51 - 54, each promise is moved to its thread. A promise can only be moved but not copied. The threads execute the function sumUp (lines 18 - 25). sumUp takes as its first argument a promise by rvalue reference. The futures ask in line 56 for the summation result by using the blocking get call.
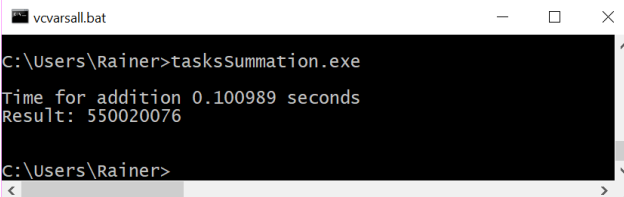
Summation with a local variable on Linux

Summation with a local variable on Windows

To conclude this section, there is an overview of all performance numbers.

6.1.3.4 All Numbers for the Thread-Local Summation

It does not make a big difference whether I use local variables or tasks to calculate the partial sum or if I use various synchronization primitives such as atomics. Only the thread-local data seems to make the program a little slower. This observation holds for Linux and Windows. Don't be surprised by the higher performance of Linux relative to Windows. The program was optimized for four cores, and my Windows laptop has only two. The numbers are in seconds.

Performance of all thread-local summations

Operating System (Compiler)	std::lock_-guard	Atomic using sequential consistency	Atomic using relaxed semantic	Thread-local data	Tasks
Linux (GCC)	0.03	0.03	0.03	0.04	0.03
Windows (cl.exe)	0.10	0.10	0.10	0.20	0.10

I want to draw our focus to a fascinating point. The thread-local multithreaded summation of the vector is about twice as fast as the single-threaded summation. I would expect a fourfold performance improvement in the optimal case because there is hardly a synchronization necessary between the threads. What is the reason?

# 6.1.4 Summation of a Vector: The Conclusion

6.1.4.1 Single Threaded

The range-based for loop and the STL algorithm `std::accumulate` are in the same performance range. In the optimized version, the compiler uses for the summation case the optimized version vectorized SIMD[2] instruction (SSE or AVX). Therefore, the loop counter is increased by 4 (SSE) or 8 (AVX).

6.1.4.2 Multithreading with a Shared Variable

The usage of a shared variable for the summation variable makes one point clear: synchronization is costly and should be avoided as much as possible. Although I used an atomic variable and even broke the sequential consistency, the four threads are 100 times slower than one thread. From a performance perspective, minimizing expensive synchronization has to be our first goal.

6.1.4.3 Thread-local Summation

The thread-local summation is only two times faster than the single-threaded range-based for loop or `std::accumulate`. That holds even though each of the four threads can work independently. That surprised me because I was expecting a nearly fourfold improvement. What surprised me, even more was that my four cores are not fully utilized.
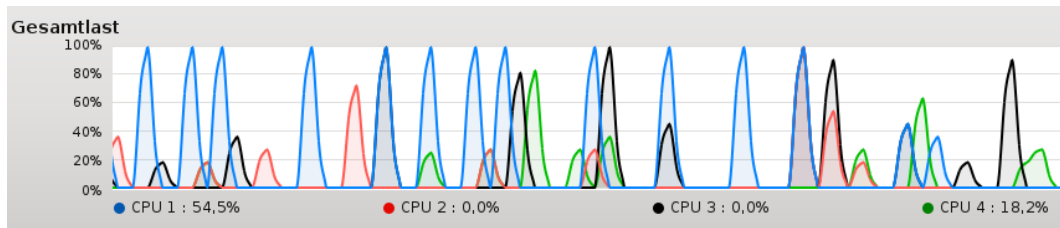
Utilisation of the four cores

The reason is simple; the cores can't get the data fast enough from memory. The execution is memory bound[3]. Or to say it the other way around, the memory slows down the cores. The following pictures show the bottleneck memory.

---

[2]https://en.wikipedia.org/wiki/SIMD
[3]https://en.wikipedia.org/wiki/Memory_bound_function

The bottleneck memory
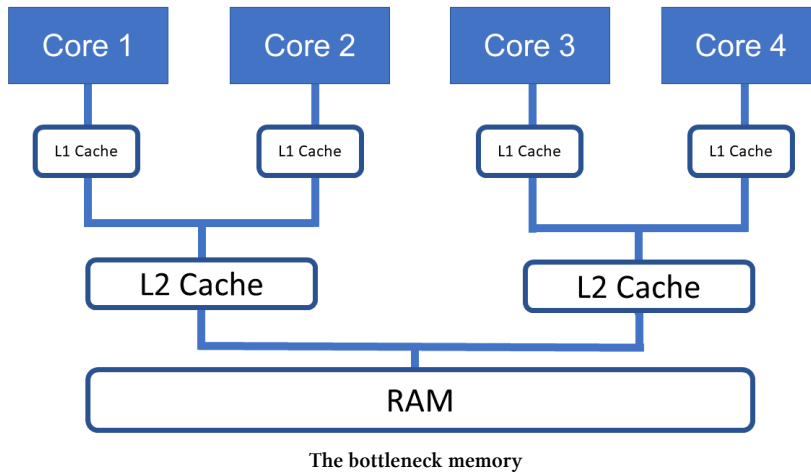
The Roofline model[4] is an intuitive performance model to provide performance estimates of applications running on multi-core or many-core architectures. The model depends on the peak performance, peak bandwidth, and arithmetic intensity of the architecture.

---

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

6.2 The Dining Philosophers Problem by Andre Adrian

Andre Adrian is the guest author of this chapter. He solved the classical dining philosopher's problem in various ways using modern C++. Andres's case study exemplifies in a very instructive way the theory explained in this book. I just made a few minimal editorial adjustments to match its layout to the layout of my book.
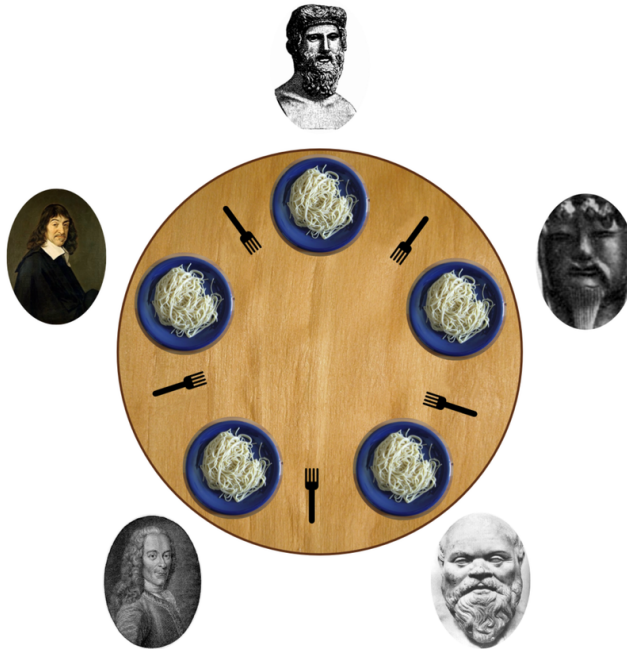
The dining philosophers problem

The dining philosophers problem is a classic synchronization problem formulated by Edsger Dijkstra[5] in the article Hierarchical Ordering of Sequential Processes[6]: Five philosophers, numbered from 0 through 4 are living in a house where the table laid for them, each philosopher having his own place at the table. Their only problem - besides those of philosophy - is that the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks. There are two forks next to each plate, so that presents no difficulty: as a consequence, however, no two neighbours may be eating simultaneously.

We use the following problem description: 4 philosophers live a simple life. Every philosopher performs the same routine: he thinks for some random duration, gets his first fork, gets his second fork, eats for some random duration, puts down the forks, and starts thinking again. To make the problem interesting the 4 philosophers have only 4 forks. Philosopher number 1 has to take forks number 1

---

[5]https://en.wikipedia.org/wiki/Edsger_W._Dijkstra
[6]https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD310.html

and 2 for eating. Philosopher 2 needs forks 2 and 3, and so on up to philosopher 4 who needs forks 4 and 1 for eating. After eating, the philosopher puts the forks back on the table.

6.2.1 Multiple Resource Use

As we go from problem description to programming, we translate philosophers to threads and forks to resources. In our first program (dp_1.cpp) we create 4 "philosopher" threads and 4 "fork" resource integers.

Multiple Resource Use

```cpp
// dp_1.cpp
#include <iostream>
#include <thread>
#include <chrono>

int myrand(int min, int max) {
  return rand()%(max-min)+min;
}

void lock(int& m) {
  m=1;
}

void unlock(int& m) {
  m=0;
}

void phil(int ph, int& ma, int& mb) {
  while(true) {
    int duration=myrand(1000, 2000);
    std::cout<<ph<<" thinks "<<duration<<"ms\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));

    lock(ma);
    std::cout<<"\t\t"<<ph<<" got ma\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    lock(mb);
    std::cout<<"\t\t"<<ph<<" got mb\n";

    duration=myrand(1000, 2000);
    std::cout<<"\t\t\t\t"<<ph<<" eats "<<duration<<"ms\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));

```
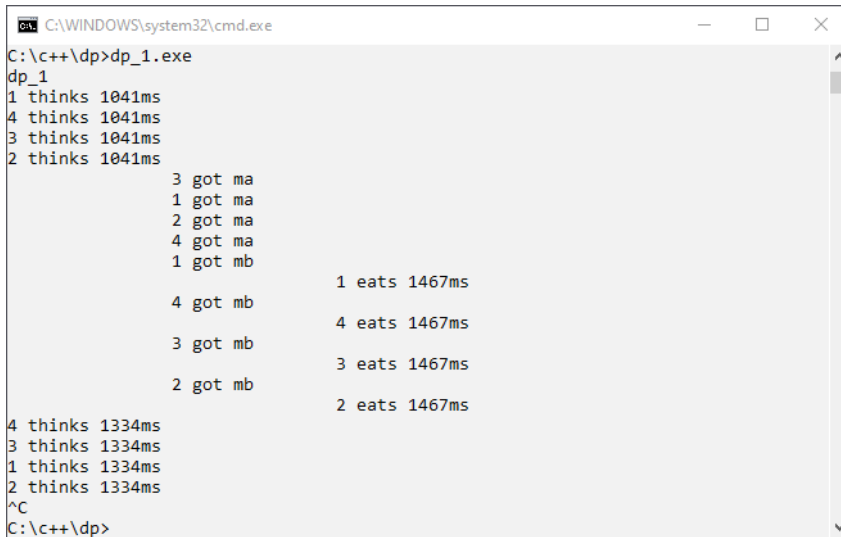
```
35        unlock(mb);
36        unlock(ma);
37    }
38  }
39
40  int main() {
41      std::cout<<"dp_1\n";
42      srand(time(nullptr));
43
44      int m1{0}, m2{0}, m3{0}, m4{0};
45
46      std::thread t1([&] {phil(1, m1, m2);});
47      std::thread t2([&] {phil(2, m2, m3);});
48      std::thread t3([&] {phil(3, m3, m4);});
49      std::thread t4([&] {phil(4, m4, m1);});
50
51      t1.join();
52      t2.join();
53      t3.join();
54      t4.join();
55  }
```

The `main()` function establishes random numbers in line 42. We set the random number generator seed value to the number of seconds since 1st January 1970. We define our fork resources in line 44. Then we start four threads beginning in line 46. To avoid premature thread termination we join the threads beginning in line 51. The thread function `phil()` has a forever loop. The `while(true)` statement is always `true`, therefore the thread will never terminate. The problem description says "he thinks for some random duration". First, we calculate a random duration with the function `myrand()` (lines 20 and 6). The function `myrand()` produces a pseudo-random return value in the range of [min, max]. For program trace, we log the philosopher's number, his current state of he thinks and the duration to the console. The `sleep_for()` statement lets the scheduler put the thread for the duration into the state waiting. In a "real" program application source code uses up time instead of `sleep_for()`. After the philosopher's thread thinking time is over, he "gets his first fork" (line 24). We use a function `lock()` to perform the "gets fork" thing. At the moment the function `lock()` is very simple because we don't know better. We just set the fork resource to the value 1 (line 10). After the philosopher thread obtained his first fork, he proudly announces the new state with a `got ma` console output. Now the thread "gets his second fork" (line 28). The corresponding console output is `got mb`. The next state is eats. Again we determine the duration, produce a console output, and occupy the thread with a `sleep_for()` (line 31). After the state eats the philosopher puts down his forks (lines 35 and 14). The `unlock()` function is again really simple and sets the resource back to 0.

Please compile the program without compiler optimization. We will see the reason later. The console output of our program looks promising:

```
C:\WINDOWS\system32\cmd.exe                                          —    □    ×

C:\c++\dp>dp_1.exe
dp_1
1 thinks 1041ms
4 thinks 1041ms
3 thinks 1041ms
2 thinks 1041ms
                  3 got ma
                  1 got ma
                  2 got ma
                  4 got ma
                  1 got mb
                                    1 eats 1467ms
                  4 got mb
                                    4 eats 1467ms
                  3 got mb
                                    3 eats 1467ms
                  2 got mb
                                    2 eats 1467ms
4 thinks 1334ms
3 thinks 1334ms
1 thinks 1334ms
2 thinks 1334ms
^C
C:\c++\dp>
```

<div align="center">Multiple resource use</div>

Have we already solved the dining philosophers problem? Well, the program output is not detailed enough to answer this question.

6.2.2 Multiple Resource Use with Logging

We should add some more logging. At the moment the function `lock()` does not check if the fork is available before the resource is used. The improved version of `lock()` in program `dp_2.cpp` is:

Multiple resource use with logging

```cpp
// dp_2.cpp

...

void lock(int& m) {
  if (m) {
    std::cout<<"\t\t\t\t\tERROR lock\n";
  }
  m=1;
}

...
```

Program version 2 produces the following output:

Multiple resource use with logging

We see `ERROR lock` console output. This output tells us that two philosophers use the same resource at the same time. What can we do?

6.2.3 Erroneous Busy Waiting without Resource Hierarchy

We can change the if statement in `lock()` into a while statement. This while statement produces a spinlock. A spinlock is a fancy word for busy waiting. While the fork resource is in use, the thread is busy waiting for a change from the state in use to the state available. At this very moment, we set the fork resource again to state in use. In program `dp_3.cpp` we have:

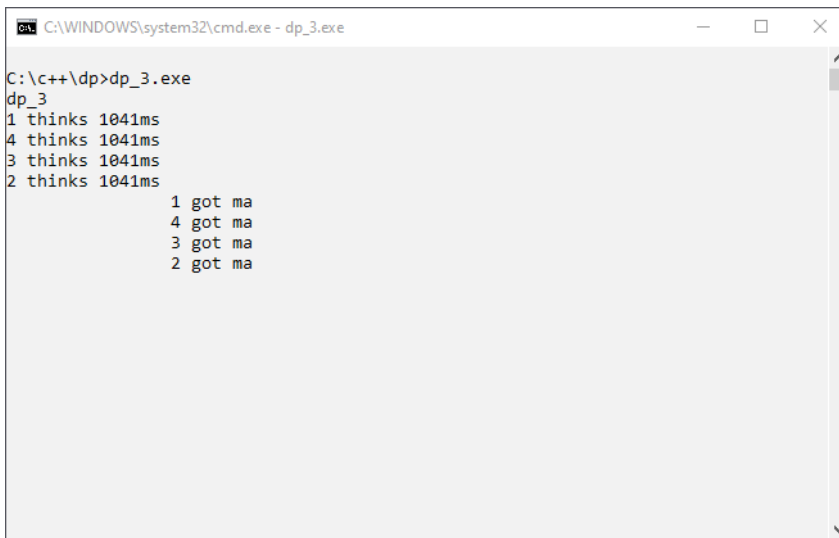**Erroneous Busy Waiting without Resource Hierarchy**

```
// dp_3.cpp

...

void lock(int& m) {
  while (m)
    ; // busy waiting
  m=1;
}

...
```

Please believe that this little change is still not a CORRECT solution for the dining philosophers'

problem. We have no longer the wrong resource usage. But we have another problem. See program version 3 output:

Erroneous busy waiting without resource hierarchy

Every philosopher thread takes his first fork resource and then can not take the second fork. What can we do? Andrew S. Tanenbaum[7] wrote in his book "Operating Systems. Design and Implementation, 3rd edition": "Another way to avoid the circular wait is to provide a global numbering of all the resources. Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order."

6.2.4 Erroneous Busy Waiting with Resource Hierarchy

This solution is known as resource hierarchy or partial ordering. For the dining philosophers problem, partial ordering is easy. The first fork taken has to be the fork with the lower number. For philosophers 1 to 3 the resources are taken in the correct order. Only philosopher thread 4 needs a change for correct partial ordering. First get fork resource 1, then get fork resource 4. See the main program in file dp_4.cpp:

---

[7]https://en.wikipedia.org/wiki/Andrew_S._Tanenbaum

Erroneous Busy Waiting with Resource Hierarchy

```cpp
// dp_4.cpp

...

int main() {
  std::cout<<"dp_4\n";
  srand(time(nullptr));

  int m1{0}, m2{0}, m3{0}, m4{0};

  std::thread t1([&] {phil(1, m1, m2);});
  std::thread t2([&] {phil(2, m2, m3);});
  std::thread t3([&] {phil(3, m3, m4);});
  std::thread t4([&] {phil(4, m1, m4);});

  t1.join();
  t2.join();
  t3.join();
  t4.join();
}

...
```

Program version 4 output looks fine:

Erroneous Busy Waiting with Resource Hierarchy (without optimization)

Now there is no longer wrong resource usage nor do we have a deadlock. We get brave and use compiler optimization. We want to have a good program that executes fast! This is program version 4 output with compiler optimization:

Erroneous Busy Waiting with Resource Hierarchy (with optimization)

It is always the same philosopher thread that eats. Is it possible that the setting of compiler optimization can change the behavior of a program? Yes, it is possible. The philosopher threads read from memory the value of fork resource. The compiler optimization optimizes some of these memory reads away. Everything has a price!

6.2.5 Still Erroneous Busy Waiting with Resource Hierarchy

The programming language C++ has the atomic template to define an atomic type. If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined. In file dp_5.cpp we use `atomic<int>` for the fork resources (lines 11, 17, 21, and 47). We include the header `<atomic>` in line 5:

Erroneous Busy Waiting with Resource Hierarchy

```cpp
1   // dp_5.cpp
2   #include <iostream>
3   #include <thread>
4   #include <chrono>
5   #include <atomic>
6
7   int myrand(int min, int max) {
8      return rand()%(max-min)+min;
9   }
10
```

```cpp
11   void lock(std::atomic<int>& m) {
12     while (m)
13       ; // busy waiting
14     m=1;
15   }
16
17   void unlock(std::atomic<int>& m) {
18     m=0;
19   }
20
21   void phil(int ph, std::atomic<int>& ma, std::atomic<int>& mb) {
22     while(true) {
23       int duration=myrand(1000, 2000);
24       std::cout<<ph<<" thinks "<<duration<<"ms\n";
25       std::this_thread::sleep_for(std::chrono::milliseconds(duration));
26
27       lock(ma);
28       std::cout<<"\t\t"<<ph<<" got ma\n";
29       std::this_thread::sleep_for(std::chrono::milliseconds(1000));
30
31       lock(mb);
32       std::cout<<"\t\t"<<ph<<" got mb\n";
33
34       duration=myrand(1000, 2000);
35       std::cout<<"\t\t\t\t"<<ph<<" eats "<<duration<<"ms\n";
36       std::this_thread::sleep_for(std::chrono::milliseconds(duration));
37
38       unlock(mb);
39       unlock(ma);
40     }
41   }
42
43   int main() {
44     std::cout<<"dp_5\n";
45     srand(time(nullptr));
46
47     std::atomic<int> m1{0}, m2{0}, m3{0}, m4{0};
48
49     std::thread t1([&] {phil(1, m1, m2);});
50     std::thread t2([&] {phil(2, m2, m3);});
51     std::thread t3([&] {phil(3, m3, m4);});
52     std::thread t4([&] {phil(4, m1, m4);});
53
54     t1.join();
55     t2.join();
```

```
56     t3.join();
57     t4.join();
58   }
```

The program version 5 output is:

Still Erroneous Busy Waiting with Resource Hierarchy

This output looks great. Now we have reached the limits of our testing methodology. There is still a *tiny chance for misbehavior.* The two operations "is a resource available" and "mark resource as in use" in the `lock()` function is atomic, but they are still two operations. Between these two operations, the scheduler can place a thread switch. And this thread switch at this most inconvenient time can produce very hard-to-find bugs in the program.

6.2.6 Correct Busy Waiting with Resource Hierarchy

Thankfully all current computers have an atomic operation "test the resource and if the test is positive mark resource as in use". In the programming language C++, the `atomic_flag` type makes this special "test and set" operation available for us. File `dp_6.cpp` has the first correct solution for the dining philosophers problem:

Correct Busy Waiting with Resource Hierarchy

```cpp
// dp_6.cpp
#include <iostream>
#include <thread>
#include <chrono>
#include <atomic>

int myrand(int min, int max) {
  return rand()%(max-min)+min;
}

void lock(std::atomic_flag& m) {
  while (m.test_and_set())
    ; // busy waiting
}

void unlock(std::atomic_flag& m) {
  m.clear();
}

void phil(int ph, std::atomic_flag& ma, std::atomic_flag& mb) {
  while(true) {
    int duration=myrand(1000, 2000);
    std::cout<<ph<<" thinks "<<duration<<"ms\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));

    lock(ma);
    std::cout<<"\t\t"<<ph<<" got ma\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    lock(mb);
    std::cout<<"\t\t"<<ph<<" got mb\n";

    duration=myrand(1000, 2000);
    std::cout<<"\t\t\t\t"<<ph<<" eats "<<duration<<"ms\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));

    unlock(mb);
    unlock(ma);
  }
}

int main() {
  std::cout<<"dp_6\n";
  srand(time(nullptr));
```

```
std::atomic_flag m1, m2, m3, m4;
unlock(m1);
unlock(m2);
unlock(m3);
unlock(m4);

std::thread t1([&] {phil(1, m1, m2);});
std::thread t2([&] {phil(2, m2, m3);});
std::thread t3([&] {phil(3, m3, m4);});
std::thread t4([&] {phil(4, m1, m4);});

t1.join();
t2.join();
t3.join();
t4.join();
}
```

The program version 6 output is similar to the last output. The dining philosophers' problem is good-natured. One resource is only shared between two threads. The atomic_flag spinlock is needed if several threads want to get the same resource.

6.2.7 Good low CPU load Busy Waiting with Resource Hierarchy

The spinlock disadvantage is the busy waiting. The while loop in lock() is a waste of CPU resources. A remedy to this problem is to put a sleep_for() function in the body of this while loop. The sleep_for() function performs waiting in the scheduler. This waiting is much better than waiting in the application. As always there is a price. The sleep_for() slows down the program's progress. File dp_7.cpp is the second correct solution:

Correct Busy Waiting with Resource Hierarchy

```
// dp_7.cpp

...

void lock(std::atomic_flag& m) {
  while (m.test_and_set())
    std::this_thread::sleep_for(std::chrono::milliseconds(8));
}

...
```

Note: a `std::this_thread::yield()` instead of the `sleep_for()` does not reduce CPU load on the author's computer. The impact of `yield()` is implementation-dependent.

6.2.8 `std::mutex` with Resource Hierarchy

To completely avoid busy waiting we need more help from the scheduler. If every thread tells the scheduler the resource state, the scheduler can put a "wait for a resource" thread into the "waiting" state. After the scheduler gets a "resource is available" information, the waiting thread state changes to ready. The thread to scheduler information exchange is expensive. Because of this C++ offers both, spinlock and `mutex`. Spinlock is waiting in the thread and mutex is waiting in the scheduler. File `dp_8.cpp` shows the mutex solution. Please note the `#include <mutex>`:

`std::mutex` with Resource Hierarchy

```cpp
// dp_8.cpp
#include <iostream>
#include <thread>
#include <chrono>
#include <mutex>

int myrand(int min, int max) {
  return rand()%(max-min)+min;
}

void phil(int ph, std::mutex& ma, std::mutex& mb) {
  while(true) {
    int duration=myrand(1000, 2000);
    std::cout<<ph<<" thinks "<<duration<<"ms\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));

    ma.lock();
    std::cout<<"\t\t"<<ph<<" got ma\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    mb.lock();
    std::cout<<"\t\t"<<ph<<" got mb\n";

    duration=myrand(1000, 2000);
    std::cout<<"\t\t\t\t"<<ph<<" eats "<<duration<<"ms\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));
    mb.unlock(); // (9)
    ma.unlock();
  }
}

int main() {
  std::cout<<"dp_8\n";
  srand(time(nullptr));
```

```
  std::mutex m1, m2, m3, m4;

  std::thread t1([&] {phil(1, m1, m2);});
  std::thread t2([&] {phil(2, m2, m3);});
  std::thread t3([&] {phil(3, m3, m4);});
  std::thread t4([&] {phil(4, m1, m4);});

  t1.join();
  t2.join();
  t3.join();
  t4.join();
}
```

Program version 8 is correct and uses minimal CPU resources. C++ offers a wrapper to mutex to make life easier for programmers.

6.2.9 `std::lock_guard` with Resource Hierarchy

Using the `lock_guard` template, we put only the mutex into the lock. The mutex member function `lock` is automatically called in the locks constructor and `unlock` in its destructor at the end of the scope. `unlock` is also called if an exception is thrown.

The convenient version is `dp_9.cpp`:

`std::lock_guard` with Resource Hierarchy

```
// dp_9.cpp

...

void phil(int ph, std::mutex& ma, std::mutex& mb) {
  while(true) {
    int duration=myrand(1000, 2000);
    std::cout<<ph<<" thinks "<<duration<<"ms\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));

    std::lock_guard<std::mutex> ga(ma);
    std::cout<<"\t\t"<<ph<<" got ma\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    std::lock_guard<std::mutex> gb(mb);
    std::cout<<"\t\t"<<ph<<" got mb\n";

    duration=myrand(1000, 2000);
    std::cout<<"\t\t\t\t"<<ph<<" eats "<<duration<<"ms\n";
```

```
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));
  }
}
```

...

We get better and better. Program versions 8 and 9 are correct and are light on the CPU load. But look careful on the program output:

std::lock_guard with Resource Hierarchyy

The program output is slightly garbled. Maybe you have seen this output distortion before. There is nothing wrong with the spinlock program versions 6 and 7 or the mutex program versions 8 and 9.

6.2.10 `std::lock_guard` and Synchronized Output with Resource Hierarchy

The console output itself is a resource. That is the reason for garbled output in multi-thread programs. The solution is to put a `lock_guard` around every console output in `dp_10.cpp`:

`std::lock_guard` with Resource Hierarchy

```cpp
// dp_10.cpp

...

std::mutex mo;

void phil(int ph, std::mutex& ma, std::mutex& mb) {
  while(true) {
    int duration=myrand(1000, 2000);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<ph<<" thinks "<<duration<<"ms\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));

    std::lock_guard<std::mutex> ga(ma);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<"\t\t"<<ph<<" got ma\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    std::lock_guard<std::mutex> gb(mb);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<"\t\t"<<ph<<" got mb\n";
    }

    duration=myrand(1000, 2000);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<"\t\t\t\t"<<ph<<" eats "<<duration<<"ms\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));
  }
}

...
```

The global mutex `mo` controls the console output resource. Every `cout` statement is in its block and the `lock_guard()` template ensures that console output is no longer garbled.

6.2.11 `std::lock_guard` and Synchronized Output with Resource Hierarchy and a count

As a little bonus, I added dp_11.cpp. This program version counts the number of philosophers threads that are eating at the same time. Because we have 4 forks, there should be times where 2 philosopher threads eat concurrently. Please note that you need again `#include <atomic>` in program dp_11.cpp:

`std::lock_guard` and Synchronized Output with Resource Hierarchy and a count

```cpp
// dp_11.cpp

...

std::mutex mo;
std::atomic<int> cnt = 0;

void phil(int ph, std::mutex& ma, std::mutex& mb) {
  while(true) {
    int duration=myrand(1000, 2000);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<ph<<" thinks "<<duration<<"ms\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));

    std::lock_guard<std::mutex> ga(ma);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<"\t\t"<<ph<<" got ma\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    std::lock_guard<std::mutex> gb(mb);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<"\t\t"<<ph<<" got mb\n";
    }

    duration=myrand(1000, 2000);
    ++cnt;
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<"\t\t\t\t"<<ph<<" eats "<<duration<<"ms "<<cnt<<"\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));
    --cnt;
```

```
    }
}
```

...

The program version 11 output is:

std::lock_guard and Synchronized Output with Resource Hierarchy and a count

6.2.12 A `std::unique_lock` using deferred locking

C++ offers alternative solutions next to resource hierarchy. At the moment we have two separate operations to acquire the two resources. If there is only one operation to acquire the two resources, there is no longer the danger of deadlock. The first "all or nothing" solution uses `unique_lock()` with `defer_lock`. The real resource acquire in program `dp_12.cpp` happens in the `lock()` statement:

A `std::unique_lock` using deferred locking

```cpp
// dp_12.cpp

...

nt myrand(int min, int max) {
  static std::mt19937 rnd(std::time(nullptr));
  return std::uniform_int_distribution<>(min,max)(rnd);
}
```

```cpp
std::mutex mo;

void phil(int ph, std::mutex& ma, std::mutex& mb) {
  while(true) {
    int duration=myrand(1000, 2000);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<ph<<" thinks "<<duration<<"ms\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));

    std::unique_lock<std::mutex> ga(ma, std::defer_lock);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<"\t\t"<<ph<<" got ma\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    std::unique_lock<std::mutex> gb(mb,std::defer_lock);
    std::lock(ga, gb);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<"\t\t"<<ph<<" got mb\n";
    }

    duration=myrand(1000, 2000);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<"\t\t\t\t"<<ph<<" eats "<<duration<<"ms\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));
  }
}
```

...

So far we have generated the random numbers using the rand() function. This function is not reentrant. Not reentrant means not threadable. This error is fixed with a modified myrand() function. The static function object rnd is a Mersenne Twister[8] random number generator. With static we avoid a global function object. Scaling to a value between min and max is now done with uniform_int_distribution<>. Using the library is better than writing your own code. Who would have thought that simple things like cout output and random number are so difficult in programs with threads?

---

[8]https://en.wikipedia.org/wiki/Mersenne_Twister

6.2.13 A `std::scoped_lock` with Resource Hierarchy

The second "all or nothing" solution is even more straightforward. The C++17 function `std::scoped_-lock()` allows acquiring multiple resources. This powerful function in program `dp_13.cpp` gives us the shortest dining philosophers solution:

A `std::scoped_lock` with Resource Hierarchy

```cpp
// dp_13.cpp

...

int myrand(int min, int max) {
  static std::mt19937 rnd(std::time(nullptr));
  return std::uniform_int_distribution<>(min,max)(rnd);
}

std::mutex mo;

void phil(int ph, std::mutex& ma, std::mutex& mb) {
  while(true) {
    int duration=myrand(1000, 2000);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<ph<<" thinks "<<duration<<"ms\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));

    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    std::scoped_lock scop(ma, mb);

    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<"\t\t"<<ph<<" got ma, mb\n";
    }

    duration=myrand(1000, 2000);
    {
      std::lock_guard<std::mutex> g(mo);
      std::cout<<"\t\t\t\t"<<ph<<" eats "<<duration<<"ms\n";
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(duration));
  }
}

...
```

There are more solutions. The original Dijkstra solution in the article Hierarchical Ordering of Sequential Processes[9] used one mutex, one semaphore per philosopher, and one state variable per philosopher. Andrew S. Tanenbaum provided a C language solution in his book "Operating Systems. Design and Implementation, 3rd edition; chapter 2.3.1".

6.2.14 The Original Dining Philosophers Problem using Semaphores

Program dp_14.cpp is the Tanenbaum solution rewritten in C++20:

The Original Dining Philosophers Problem using Semaphores

```cpp
// dp_14.cpp
#include <iostream>
#include <chrono>
#include <thread>
#include <mutex>
#include <semaphore>
#include <random>

int myrand(int min, int max) {
  static std::mt19937 rnd(std::time(nullptr));
  return std::uniform_int_distribution<>(min,max)(rnd);
}

enum {
  N=5,                 // number of philosophers
  THINKING=0,          // philosopher is thinking
  HUNGRY=1,            // philosopher is trying to get forks
  EATING=2,            // philosopher is eating
};

#define LEFT (i+N-1)%N  // number of i's left neighbor
#define RIGHT (i+1)%N   // number of i's right neighbor

int state[N];          // array to keep track of everyone's state
std::mutex mutex_;      // mutual exclusion for critical regions
std::binary_semaphore s[N]{0, 0, 0, 0, 0};
                       // one semaphore per philosopher

void test(int i)       // i: philosopher number, from 0 to N-1
{
  if (state[i] == HUNGRY
   && state[LEFT] != EATING && state[RIGHT] != EATING) {
    state[i] = EATING;
```

[9]https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD310.html

```cpp
    s[i].release();
  }
}

void take_forks(int i)  // i: philosopher number, from 0 to N-1
{
  mutex_.lock();        // enter critical region
  state[i] = HUNGRY;    // record fact that philosopher i is hungry
  test(i);              // try to acquire 2 forks
  mutex_.unlock();      // exit critical region
  s[i].acquire();       // block if forks were not acquired
}

void put_forks(int i)  // i: philosopher number, from 0 to N-1
{
  mutex_.lock();        // enter critical region
  state[i] = THINKING;  // philosopher has finished eating
  test(LEFT);           // see if left neighbor can now eat
  test(RIGHT);          // see if right neighbor can now eat
  mutex_.unlock();      // exit critical region
}

std::mutex mo;

void think(int i) {
  int duration = myrand(1000, 2000);
  {
                std::lock_guard<std::mutex> g(mo);
                std::cout<<i<<" thinks "<<duration<<"ms\n";
        }
  std::this_thread::sleep_for(std::chrono::milliseconds(duration));
}

void eat(int i) {
  int duration = myrand(1000, 2000);
  {
                std::lock_guard<std::mutex> g(mo);
                std::cout<<"\t\t\t\t"<<i<<" eats "<<duration<<"ms\n";
        }
  std::this_thread::sleep_for(std::chrono::milliseconds(duration));
}

void philosopher(int i) // i: philosopher number, from 0 to N-1
{
  while (true) {        // repeat forever
```

```cpp
    think(i);              // philosopher is thinking
    take_forks(i);         // acquire two forks or block
    eat(i);                // yum-yum, spaghetti
    put_forks(i);          // put both forks back on table
  }
}

int main() {
  std::cout<<"dp_14\n";

  std::thread t0([&] {philosopher(0);});
  std::thread t1([&] {philosopher(1);});
  std::thread t2([&] {philosopher(2);});
  std::thread t3([&] {philosopher(3);});
  std::thread t4([&] {philosopher(4);});
  t0.join();
  t1.join();
  t2.join();
  t3.join();
  t4.join();
}
```

By the way, the semaphore is the oldest thread synchronization primitive. Dijkstra defined the `P()` and `V()` operation in 1965: "It is the P-operation, which represents the potential delay, viz. when a process initiates a P-operation on a semaphore, that at that moment is = 0, in that case, this P-operation cannot be completed until another process has performed a V-operation on the same semaphore and has given it the value '1'." Today `P()` is called `release()` and `V()` is called `acquire()`. (Cooperating sequential processes[10])

6.2.15 A C++20 Compatible Semaphore

You need a C++20 compiler like LLVM (clang++) version 13.0.0 or newer to compile `dp_14.cpp`. Or you change the `#include <semaphore>` into `#include "semaphore.h"` and provide the following header file. Then a C++11 compiler is sufficient.

---

[10]https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html

The Original Dining Philosophers Problem using Semaphores

```cpp
// semaphore.h
#include <mutex>
#include <condition_variable>
#include <limits>

namespace std {
  template <std::ptrdiff_t least_max_value
   = std::numeric_limits<std::ptrdiff_t>::max()>
  class counting_semaphore {
  public:
    counting_semaphore(std::ptrdiff_t desired) : counter(desired) {}

    counting_semaphore(const counting_semaphore&) = delete;
    counting_semaphore& operator=(const counting_semaphore&) = delete;

    inline void release(ptrdiff_t update = 1) {
      std::unique_lock<std::mutex> lock(mutex_);
      counter += update;
      cv.notify_one();
    }

    inline void acquire() {
      std::unique_lock<std::mutex> lock(mutex_);
      while (0 == counter) cv.wait(lock);
      --counter;
    }

  private:
    ptrdiff_t counter;
    std::mutex mutex_;
    std::condition_variable cv;
  };

  using binary_semaphore = counting_semaphore<1>;
}
```

The C++ semaphore consists of a counter, a mutex, and a condition_variable. After 14 program versions, we leave this topic. The programs versions 1 to 6 have problems. I presented them to show bad multi-thread programming. Don't copy that!

6.3 Thread-Safe Initialization of a Singleton

Before I start with this case study, let me emphasize: I am not advocating using the singleton pattern. Therefore, allow me to begin this chapter about the thread-safe initialization of a singleton with a warning.

⚠ Thoughts about Singletons

I only use the singleton pattern in my case studies because it is a classic example of a variable that has to be initialized in a thread-safe way. The singleton pattern has a few severe disadvantages. Let me give you a few ideas:

- A singleton is a global variable in disguise. Therefore, it makes it quite challenging to test your function because it depends on the global state.
- Typically, you use a singleton in a function by invoking the static member function `MySingleton::getInstance()`. This means the interface of a function does not tell you that you use a singleton inside. You hide the dependency on the singleton.
- If you have to static object x and y in separate source files and the construction of these objects depend on each other, you are in the static initialization order fiasco[11] because there is no guarantee which static is initialized first. Moreover, singletons are static objects.
- The singleton pattern manages the lazy creation of an object but not its destruction. If you don't destruct something you don't need anymore, this is called a memory leak.
- Imagine you want to subclass the singleton. Should it be possible? What does that mean for the implementation?
- A thread-safe and fast singleton implementation is quite challenging.

For a more elaborate discussion about the pros and cons of the singleton pattern, please refer to the referenced articles in the Wikipedia page for the singleton pattern[12].

I want to start my discussion of the thread-safe initialization of the singleton with a short detour.

6.3.1 Double-Checked Locking Pattern

The double-checked locking[13] pattern is the classical way to initialize a singleton in a thread-safe way. What sounds like established best practice or as a pattern, is more a kind of anti-pattern[14]. It assumes guarantees in the traditional implementation, which aren't given by the Java, C# or C++ memory model anymore. The wrong assumption is that creating a singleton is an atomic operation; therefore, a solution that seems to be thread-safe is not thread-safe.

---

[11]https://isocpp.org/wiki/faq/ctors
[12]https://en.wikipedia.org/wiki/Singleton_pattern
[13]https://www.dre.vanderbilt.edu/~schmidt/PDF/DC-Locking.pdf
[14]https://en.wikipedia.org/wiki/Anti-pattern

What is the double-checked locking pattern? The first idea to implement a thread-safe singleton is to protect the initialization of the singleton with a lock.

Thread-safe initialization with a lock

```
1   std::mutex myMutex;
2
3   class MySingleton{
4   public:
5     static MySingleton& getInstance(){
6       std::lock_guard<mutex> myLock(myMutex);
7       if(!instance) instance = new MySingleton();
8       return *instance;
9     }
10  private:
11    MySingleton() = default;
12    ~MySingleton() = default;
13    MySingleton(const MySingleton&) = delete;
14    MySingleton& operator= (const MySingleton&) = delete;
15    static MySingleton* instance;
16  };
17
18  MySingleton* MySingleton::instance = nullptr;
```

Any issues? Yes and no. Yes because there is a considerable performance penalty. No because the implementation is thread-safe. A heavyweight lock protects each access to the singleton in line 7. This also applies to the read access, which after the initial construction of MySingleton is not necessary. Here comes the double-checked locking pattern to our rescue. Let's have a look at the getInstance function.

The double-checked locking pattern

```
1   static MySingleton& getInstance(){
2     if (!instance){                              // check
3       lock_guard<mutex> myLock(myMutex);         // lock
4       if(!instance) instance = new MySingleton(); // check
5     }
6     return *instance;
7   }
```

Instead of the heavyweight lock, I use a lightweight pointer comparison in line 2. If I get a null pointer, I apply the heavyweight lock on the singleton (line 3). Because there is the possibility that another thread initializes the singleton between the pointer comparison in line 2 and the lock call in line 3, I have to perform an additional pointer comparison in line 4. So the name is obvious; two times a check and one time a lock.

Smart? Yes. Thread-safe? No.

What is the issue? The call `instance= new MySingleton()` in line 4 consists of at least three steps.

1. Allocate memory for `MySingleton`
2. Initialize the `MySingleton` object
3. Let `instance` refer to the fully initialized `MySingleton` object

The issue is that the C++ runtime provides no guarantee that the steps are performed in that sequence. For example, the processor may reorder the steps to the sequence 1,3, and 2. So in the first step, the memory is allocated, and in the second step, `instance` refers to a non-initialized singleton. If just at that moment another thread `t2` tries to access the singleton and makes the pointer comparison, the comparison succeeds. The consequence is that thread `t2` refers to a non-initialized singleton, and the program behavior is undefined.

6.3.2 Performance Measurement

I want to measure how expensive it is to access a singleton object. For reference timing, I use a singleton which I access 40 million times sequentially. Of course, the first access initializes the singleton object. In contrast, the accesses from four threads is done concurrently. I'm only interested in the performance numbers. Therefore I sum up the execution time of the four threads. I measure the performance using a static variable with block scope (Meyers Singleton), a lock `std::lock_guard`, the function `std::call_once` in combination with the `std::once_flag`, and atomics with sequential consistency and acquire release semantic.

The program runs on two PCs. My Linux PC with the GCC compiler has four cores, while my Windows PC with the cl.exe compiler has two. I compile the program with maximum optimization. Please refer to the beginning of this chapter for the details about my setup.

I want to answer two questions:

1. What are the performance numbers of the various singleton implementations?
2. Is there a significant difference between Linux (GCC) and Windows (cl.exe)?

Finally, I collect all numbers in a table.

Before I present the performance numbers of the various multithreading implementations, here is the sequential program. The `getInstance` member function is not thread-safe with the C++03 standard.

Single-threaded singleton implementation

```cpp
1    // singletonSingleThreaded.cpp
2
3    #include <chrono>
4    #include <iostream>
5
6    constexpr auto tenMill = 10000000;
7
8    class MySingleton{
9    public:
10     static MySingleton& getInstance(){
11       static MySingleton instance;
12       volatile int dummy{};
13       return instance;
14     }
15   private:
16     MySingleton() = default;
17     ~MySingleton() = default;
18     MySingleton(const MySingleton&) = delete;
19     MySingleton& operator=(const MySingleton&) = delete;
20
21   };
22
23   int main(){
24
25     constexpr auto fourtyMill = 4 * tenMill;
26
27     const auto begin= std::chrono::system_clock::now();
28
29     for ( size_t i = 0; i <= fourtyMill; ++i){
30       MySingleton::getInstance();
31     }
32
33     const auto end = std::chrono::system_clock::now() - begin;
34
35     std::cout << std::chrono::duration<double>(end).count() << '\n';
36
37   }
```

As the reference implementation, I use the so-called Meyers Singleton, named after Scott Meyers[15]. The elegance of this implementation is that the singleton object in line 11 is a static variable with a

---

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

block scope; therefore, `instance` is initialized only once. This initialization happens when the static member function `getInstance` (lines 10 - 14) is executed the first time.

The `volatile` Variable `dummy`

When I compiled the program with maximum optimization, the compiler removed the call `MySingleton::getInstance()` in line 30 because the call has no effect; therefore, I got very fast execution but wrong performance numbers. Using the `volatile` variable `dummy` (line 12), the compiler is not allowed to optimize away the `MySingleton::getInstance()` call in line 30.

Here are the reference numbers for the single-threaded use-case.

Meyers Singelton on Linux (single theaded)

Meyers Singleton on Windows (single theaded)

The beauty of the Meyers Singleton is that it becomes thread-safe with C++11.

6.3.3 Thread-Safe Meyers Singleton

The C++11 standard guarantees that static variables with block scope are initialized in a thread-safe way. The Meyers Singleton uses a static variable with block scope, so we are done. The only work that is left to do is to rewrite the previously used classical Meyers Singleton for the multithreading use-case.

Meyers-Singleton in the multithreading use-case
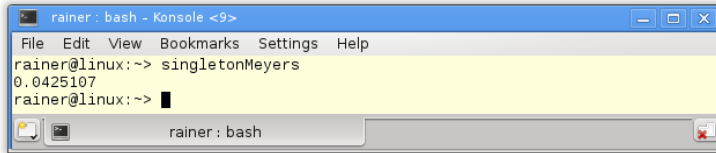
```cpp
1   // singletonMeyers.cpp
2
3   #include <chrono>
4   #include <iostream>
5   #include <future>
6
7   constexpr auto tenMill = 10000000;
8
9   class MySingleton{
10  public:
11    static MySingleton& getInstance(){
12      static MySingleton instance;
13      volatile int dummy{};
14      return instance;
15    }
16  private:
17    MySingleton() = default;
18    ~MySingleton() = default;
19    MySingleton(const MySingleton&) = delete;
20    MySingleton& operator=(const MySingleton&) = delete;
21
22  };
23
24  std::chrono::duration<double> getTime(){
25
26    auto begin = std::chrono::system_clock::now();
27    for (size_t i = 0; i <= tenMill; ++i){
28        MySingleton::getInstance();
29    }
30    return std::chrono::system_clock::now() - begin;
31
32  };
33
34  int main(){
35
36      auto fut1= std::async(std::launch::async, getTime);
37      auto fut2= std::async(std::launch::async, getTime);
38      auto fut3= std::async(std::launch::async, getTime);
39      auto fut4= std::async(std::launch::async, getTime);
40
41      const auto total= fut1.get() + fut2.get() + fut3.get() + fut4.get();
42
43      std::cout << total.count() << '\n';
```

```
44
45   }
```

I use the singleton object in the function `getTime` (lines 24 - 32). The function is executed by the four promises in lines 36 - 39. The results of the associated futures are summed up in line 41. That's all. Only the execution time is missing.
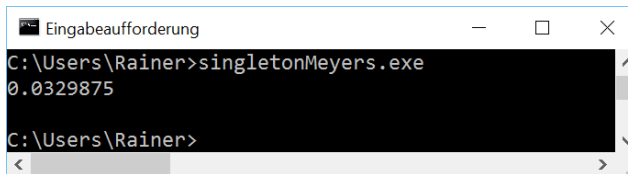
Meyers Singleton on Linux (multi threaded)

Meyers Singleton on Windows (multi threaded)

I reduce the examples to the singleton implementation

The function `getTime` for calculating the execution time and the `main` function are almost identical. Therefore, I skip them in the remaining examples of this subsection. For the entire program, please refer to the source code for this book.

Let's go for the most obvious one and use a lock.

6.3.4 `std::lock_guard`

The mutex wrapped in a `std::lock_guard` guarantees that the singleton is initialized in a thread-safe way.

A thread-safe singleton using a lock

```cpp
// singletonLock.cpp

#include <chrono>
#include <iostream>
#include <future>
#include <mutex>

constexpr auto tenMill = 10000000;

std::mutex myMutex;

class MySingleton{
public:
  static MySingleton& getInstance(){
    std::lock_guard<std::mutex> myLock(myMutex);
    if (!instance){
        instance= new MySingleton();
    }
    volatile int dummy{};
    return *instance;
  }
private:
  MySingleton() = default;
  ~MySingleton() = default;
  MySingleton(const MySingleton&) = delete;
  MySingleton& operator=(const MySingleton&) = delete;

  static MySingleton* instance;
};


MySingleton* MySingleton::instance = nullptr;

...
```

You may have already guessed that this approach is pretty slow.

Multi-threaded singleton on Linux using std::lock_guard

Multi-threaded singleton on Windows using std::lock_guard

The next version of the thread-safe singleton pattern is also based on the multithreading library: it uses `std::call_once` combined with the `std::once_flag`.

6.3.5 `std::call_once` **with** `std::once_flag`

You can use the function std::call_once together with the std::once_flag to register callables so that exactly one callable is executed in a thread-safe way.

A thread-safe singleton using std::call_once together with the `std::once_flag`

```cpp
// singletonCallOnce.cpp

#include <chrono>
#include <iostream>
#include <future>
#include <mutex>
#include <thread>

constexpr auto tenMill = 10000000;

class MySingleton{
public:
  static MySingleton& getInstance(){
    std::call_once(initInstanceFlag, &MySingleton::initSingleton);
    volatile int dummy{};
    return *instance;
  }
```

```
18  private:
19    MySingleton() = default;
20    ~MySingleton() = default;
21    MySingleton(const MySingleton&) = delete;
22    MySingleton& operator=(const MySingleton&) = delete;
23
24    static MySingleton* instance;
25    static std::once_flag initInstanceFlag;
26
27    static void initSingleton(){
28      instance= new MySingleton;
29    }
30  };
31
32  MySingleton* MySingleton::instance = nullptr;
33  std::once_flag MySingleton::initInstanceFlag;
34
35  ...
```

Here are the performance numbers.
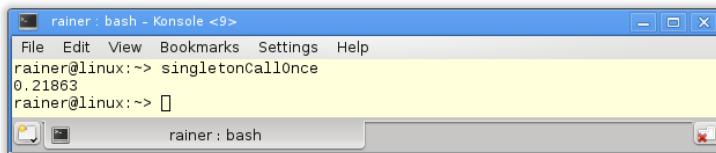
Multi-threaded singleton on Linux using `std::call_once` and `std::once_flag`
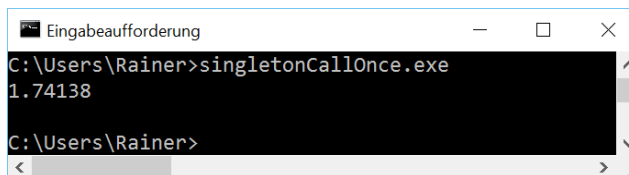
Multi-threaded singleton on Windows using `std::call_once` and `std::once_flag`

Let's continue our thread-safe singleton implementation using atomics.

6.3.6 Atomics

With atomic variables, my implementation becomes a lot more challenging. I can even specify the memory-ordering for my atomic operations. The following two implementations of the thread-safe singletons are based on the previously mentioned double-checked locking pattern.

6.3.6.1 Sequential consistency

In my first implementation, I use atomic operations without explicitly specifying the memory-ordering; therefore, sequential consistency applies.

A thread-safe singleton using atomics with sequential consistency

```cpp
1   // singletonSequentialConsistency.cpp
2
3   #include <atomic>
4   #include <iostream>
5   #include <future>
6   #include <mutex>
7   #include <thread>
8
9   constexpr auto tenMill = 10000000;
10
11  class MySingleton{
12  public:
13    static MySingleton* getInstance(){
14      MySingleton* sin = instance.load();
15      if (!sin){
16        std::lock_guard<std::mutex> myLock(myMutex);
17        sin = instance.load(std::memory_order_relaxed);
18        if(!sin){
19          sin= new MySingleton();
20          instance.store(sin);
21        }
22      }
23      volatile int dummy{};
24      return sin;
25    }
26  private:
27    MySingleton() = default;
28    ~MySingleton() = default;
29    MySingleton(const MySingleton&) = delete;
30    MySingleton& operator=(const MySingleton&) = delete;
31
32    static std::atomic<MySingleton*> instance;
33    static std::mutex myMutex;
34  };
35
36
37  std::atomic<MySingleton*> MySingleton::instance;
38  std::mutex MySingleton::myMutex;
39
```

40   ...

In contrast to the double-checked locking pattern, I now guarantee that the expression `sin= new MySingleton()` in line 19 happens before the store expression `instance.store(sin)` in line 20. This is due to the sequential consistency as default memory-ordering for atomic operations. Have a look at line 17: `sin = instance.load(std::memory_order_relaxed)`. This additional load is necessary because, between the first load in line 14 and the usage of the lock-in line 16, another thread may kick in and change the value of `instance`.

Multi-threaded singleton on Linux using atomics

Multi-threaded singleton on Windows using atomics

We can optimize the program even more.

6.3.6.2 Acquire-release semantic

Let's have a closer look at the previous thread-safe implementation of the singleton pattern using atomics. The loading (or reading) of the singleton in line 14 is an acquire operation, the storing (or writing) in line 20 a release operation. Both operations take place on the same atomic. Therefore sequential consistency is overkill. The C++11 standard guarantees that a release operation synchronizes with an acquire operation on the same atomic and establishes an ordering constraint. This means that all previous read and write operations cannot be moved after a release operation. All subsequent read and write operations cannot be moved before an acquire operation.

These are the minimum guarantees required to implement a thread-safe singleton.

A thread-safe singleton using atomics with acquire-release semantic

```cpp
1   // singletonAcquireRelease.cpp
2
3   #include <atomic>
4   #include <iostream>
5   #include <future>
6   #include <mutex>
7   #include <thread>
8
9   constexpr auto tenMill = 10000000;
10
11  class MySingleton{
12  public:
13    static MySingleton* getInstance(){
14      MySingleton* sin = instance.load(std::memory_order_acquire);
15      if (!sin){
16        std::lock_guard<std::mutex> myLock(myMutex);
17        sin = instance.load(std::memory_order_relaxed);
18        if(!sin){
19          sin = new MySingleton();
20          instance.store(sin, std::memory_order_release);
21        }
22      }
23      volatile int dummy{};
24      return sin;
25    }
26  private:
27    MySingleton() = default;
28    ~MySingleton() = default;
29    MySingleton(const MySingleton&) = delete;
30    MySingleton& operator=(const MySingleton&) = delete;
31
32    static std::atomic<MySingleton*> instance;
33    static std::mutex myMutex;
34  };
35
36
37  std::atomic<MySingleton*> MySingleton::instance;
38  std::mutex MySingleton::myMutex;
39
40  ...
```
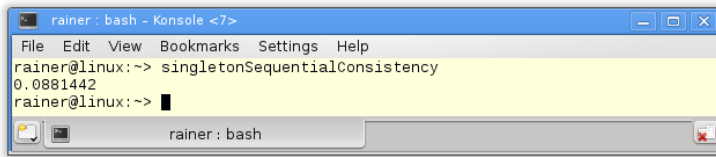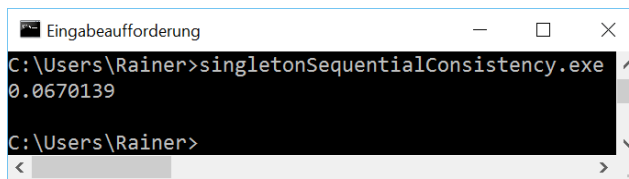
The acquire-release semantic has a similar performance as the sequential consistency.

Multi-threaded singleton on Linux using atomics

Multi-threaded singleton on Windows using atomics

This is not surprising because on the x86 architecture, both memory-orderings are very similar. We would probably more significant difference in the performance numbers on the ARMv7[16] or PowerPC[17] architecture. You can read the details Jeff Preshings blog Preshing on Programming[18].

At the end here is an overview of all performance numbers.

6.3.7 Performance Numbers of the various Thread-Safe Singleton Implementations

The numbers give a clear indication. The Meyers Singleton is the fastest one. It is not only the fastest one, but it is also the easiest one to get. The Meyers Singleton is about two times faster than the atomic versions. As expected the synchronization with the lock is the most heavyweight and, therefore, the slowest. `std::call_once` in particular on Windows is a lot slower than on Linux.

Performance of all singleton implementations

Operating System (Compiler)	Single Threaded	Meyers Singleton	std::lock_- guard	std::call_- once	Sequential Consis- tency	Acquire- Release Semantic
Linux (GCC)	0.03	0.04	12.47	0.22	0.09	0.07
Windows (cl.exe)	0.02	0.03	15.48	1.74	0.07	0.07

I want to stress one point about the numbers explicitly. These are the summed up numbers for all four

---

[16]https://en.wikipedia.org/wiki/ARM_architecture
[17]https://en.wikipedia.org/wiki/PowerPC
[18]http://preshing.com/

threads. That means that we get optimal concurrency with the Meyers Singleton because the Meyers Singleton is nearly as fast as the single-threaded implementation.

6.4 Ongoing Optimization with CppMem

I start with a small program and improve it successively. I verify each step of my process with CppMem. CppMem[19] is an interactive tool for exploring the behavior of small code snippets using the C++ memory model.

First, here is the small program.

The reference program for the ongoing optimization
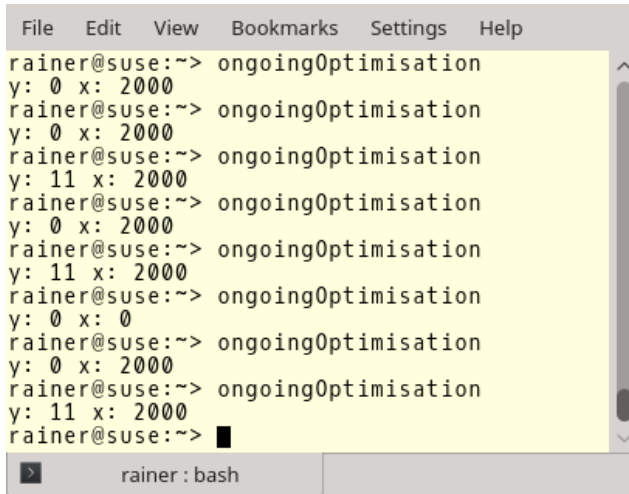
```cpp
// ongoingOptimization.cpp

#include <iostream>
#include <thread>

int x = 0;
int y = 0;

void writing(){
  x = 2000;
  y = 11;
}

void reading(){
  std::cout << "y: " << y << " ";
  std::cout << "x: " << x << '\n';
}

int main(){
  std::thread thread1(writing);
  std::thread thread2(reading);
  thread1.join();
  thread2.join();
}
```

The program is quite simple. It consists of the two threads thread1 and thread2. thread1 writes the values x and y. thread2 reads the values x and y in the **opposite sequence**. This idea sounds straightforward, but even in this simple program, we get three different results:

---

[19]http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/

The reference program

I have two questions in mind for my process of ongoing optimization.

1. Is the program well-defined? In particular: is there a data-race?
2. Which values for x and y are possible?

The first question is often very challenging to answer. In the first step, I think about the answer to the first question, and in the second step, I verify my reasoning with CppMem. Once I have answered the first question, the second answer can easily be determined from the first answer. I also present the possible values for x and y in a table.

However, I haven't explained what I mean by ongoing optimization. It's pretty simple; I successively optimize the program by weakening the C++ memory-ordering. These are my optimization steps:

- Non-atomic variables
- Locks
- Atomics with sequential consistency
- Atomics with acquire-release semantic
- Atomics with relaxed semantic
- Volatile variables

Before I start my process of ongoing optimization, you should have a basic understanding of CppMem. The chapter CppMem gives you a simplified introduction.

6.4.1 CppMem: Non-Atomic Variables

Using the run button shows it immediately there is a data race. To be more precise, it has two data races. Neither the access to the variable x nor the variable y is protected. As a result, the program has undefined behavior. In C++ jargon, this means that the program has so-called catch fire semantic; therefore, all results are possible. Your PC can even catch fire.

So, we are not able to conclude the values of x and y.

Guarantees for int variables

Most mainstream architectures guarantee that access to an int variable is atomic as long as the int variable is aligned naturally. Naturally aligned means that on a 32-bit or 64-bit architecture, the 32-bit int variable must have an address divisible by 4. There is a reason why I mention this so explicitly. With C++11, you can adjust the alignment of your data types.

I have to emphasize that I'm not advising you to use an int like an atomic int. I only want to point out that the compiler guarantees more in this case than the C++11 standard. If you rely on the compiler guarantee, your program is not compliant with the C++ standard and, therefore, may break on other hardware platforms or in the future.

These were my thoughts to int. Now we should have a look at what CppMem reports about the undefined behavior of the program.

CppMem allows me to reduce the program to its bare minimum.

CppMem: unsynchronized access

```
1   int main() {
2     int x = 0;
3     int y = 0;
4     {{{ {
5           x = 2000;
6           y = 11;
7       }
8     ||| {
9           y;
10          x;
11      }
12    }}}
13  }
```

You can define a thread in CppMem with the curly braces (lines 4 and 12) and the pipe symbol (line 8). The additional curly braces in lines 4 and 7 or lines 8 and 11 define the work package of the thread. Because I'm not interested in the output of the variables x and y, I only read them in lines 9 and 10.

That was the theory for CppMem, now to the practice.

6.4.1.1 The Analysis

When I execute the program, CppMem complains (1) (in red) that one of the four possible interleavings of threads is not race free. Only the first execution is consistent. Now I can use CppMem to switch between the four executions (2) and analyze the annotated graph (3).

A data race with non-atomics

You get the most out of CppMem by analyzing the various graphs.

6.4.1.1.1 First Execution

What conclusions can we derive from the following graph?
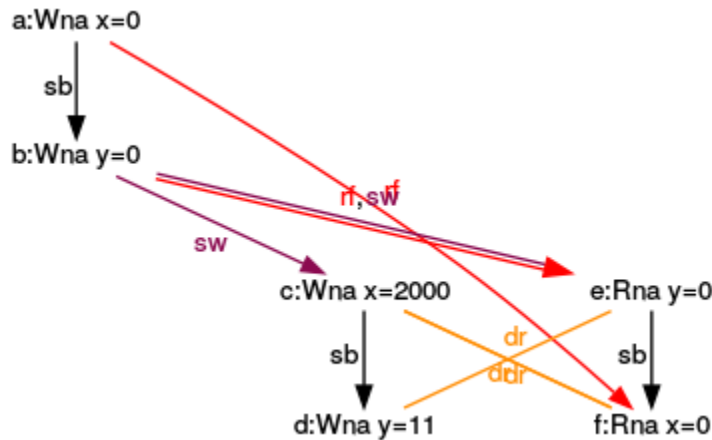
First execution

The nodes of the graph represent the expressions of the program, the edges the relation between the expressions. In my explanation, I refer to the names (a) to (f). What can I conclude from the annotations in this graph?

- **a:Wna x = 0**: Is the first expression (a), which is a non-atomic write of x.

- **sb (sequenced-before)**: The writing of the first expression (a) is sequenced before the writing of the second expression (b). These relations also hold between the expressions (c) and (d), and (e) and (f).

- **rf (read from)**: The expression (e) reads the value of y from the expression (b). Accordingly, (f) reads from (a).

- **sw (synchronizes-with)**: The expression (a) synchronises with (f). This relation holds because the expressions (f) takes place in a separate thread. The creation of a thread is a synchronization point. Everything that happens before the thread creation is visible in the thread. For symmetry reasons, the same argument holds between (b) and (e).

- **dr (data race)**: Here is the data race between the reading and writing of the variables x and y. The program has undefined behavior.

Why is the execution consistent?

The execution is consistent because the values x and y are initialized from the values in the main thread (a) and (b). The initialization of the values x and y from the expressions (c) and (d) is not consistent with the memory model.

The next three executions are not consistent.

6.4.1.1.2 Second Execution

Second execution
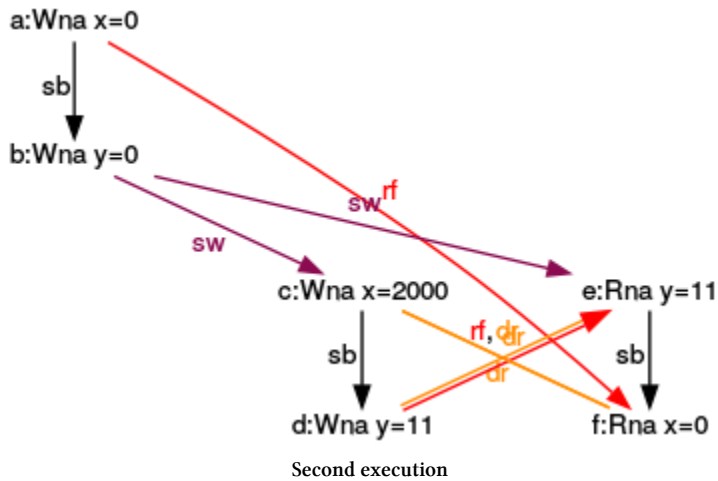
The expression (e) reads in this non-consistent execution the value of y from the expression (d). The writing of (d) happens concurrently with the reading of (e).

6.4.1.1.3 Third Execution

Third execution
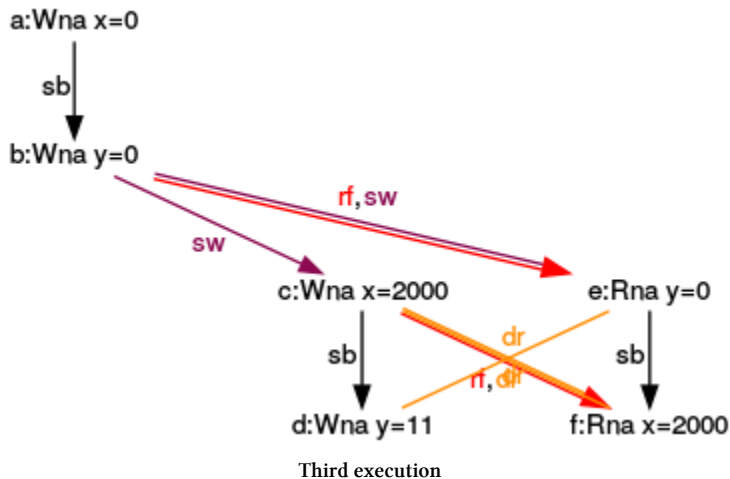
This execution is symmetric to the previous execution. The expression (f) reads from expression (c) concurrently.

6.4.1.1.4 Fourth Execution

Fourth exection
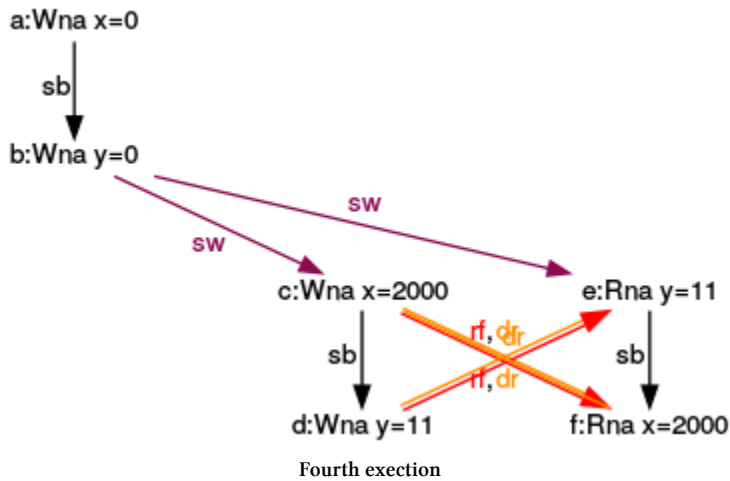
Now everything goes wrong. The expressions (e) and (f) read from the expressions (d) and (c) concurrently.

6.4.1.1.5 A Short Conclusion

Although I just used the default configuration of CppMem, I got a lot of valuable information and insight. In particular, the graphs from CppMem showed:

- All four combinations of x and y are possible: (0,0), (11,0), (0,2000), and (11,2000).
- The program has at least one data race and, therefore, has undefined behavior.
- Only one of the four possible executions is consistent.

🔑 Using volatile

From the memory model perspective, using the qualifier `volatile` for x and y makes no difference to using non-synchronized access to x and y.

CppMem: unsynchronized access with volatile

```
1   int main() {
2     volatile int x = 0;
3     volatile int y = 0;
4     {{{ {
5          x = 2000;
6           y = 11;
7        }
8     ||| {
9          y;
10          x;
11       }
12    }}}
13  }
```

CppMem generates identical graphs as in the previous example. The reason is quite simple, in C++ `volatile` has no multithreading semantic.

The access to x and y in this example was not synchronized, and we got a data race; therefore, undefined behavior. The most obvious way for synchronization is to use locks.

6.4.2 CppMem: Locks

Both threads `thread1` and `thread2` use the same mutex, wrapped in a `std::lock_guard`.

ongoing optimization with locks

```
1   // ongoingOptimizationLock.cpp
2
3   #include <iostream>
4   #include <mutex>
5   #include <thread>
6
7   int x = 0;
8   int y = 0;
9
10  std::mutex mut;
11
12  void writing(){
13    std::lock_guard<std::mutex> guard(mut);
14    x = 2000;
```

```
15    y = 11;
16  }
17
18  void reading(){
19    std::lock_guard<std::mutex> guard(mut);
20    std::cout << "y: " << y << " ";
21    std::cout << "x: " << x << '\n';
22  }
23
24  int main(){
25    std::thread thread1(writing);
26    std::thread thread2(reading);
27    thread1.join();
28    thread2.join();
29  };
```

The program is well-defined. Depending on the execution order (thread1 vs thread2), either both values are either at first read and then overwritten, or are at first overwritten and then read. The following values for x and y are possible.

Possible values for locks

y	x	Values possible?
0	0	Yes
11	0	
0	2000	
11	2000	Yes

Using `std::lock_guard` in CppMem

I could not find a way to use std::lock_guard in CppMem. If you know how to achieve it, please let me know.

Locks are easy to use, but the synchronization is often too heavyweight. I now switch to a more lightweight strategy and use atomics.

6.4.3 CppMem: Atomics with Sequential Consistency

If you don't specify the memory-ordering, sequential consistency is applied. Sequential consistency guarantees two properties. Each thread executes its instructions in source code order, and all threads follow the same global order.

Here is the optimized version of the program using atomics.

ongoing optimization with atomics (sequential consistency)

```cpp
1   // ongoingOptimizationSequentialConsistency.cpp
2
3   #include <atomic>
4   #include <iostream>
5   #include <thread>
6
7   std::atomic<int> x{0};
8   std::atomic<int> y{0};
9
10  void writing(){
11    x.store(2000);
12    y.store(11);
13  }
14
15  void reading(){
16    std::cout << y.load() << " ";
17    std::cout << x.load() << '\n';
18  }
19
20  int main(){
21    std::thread thread1(writing);
22    std::thread thread2(reading);
23    thread1.join();
24    thread2.join();
25  };
```

Let's analyze the program. The program is data race free because x and y are atomics. Therefore, only one question is left to answer. What values are possible for x and y? This question is easy to answer. Thanks to the sequential consistency, all threads have to follow the same global order.

It holds true:

- `x.store(2000);` **happens-before** `y.store(11);`
- `std::cout << y.load() << " ";` **happens-before** `std::cout << x.load() << '\n';`

Hence: the value of `x.load()` cannot be 0 if `y.load()` has the value 11, because `x.store(2000)` happens before `y.store(11)`.

All other values for x and y are possible. Here are three possible interleavings resulting in the three different values for x and y.

1. `thread1` is completely executed before `thread2`.
2. `thread2` is completely executed before `thread1`.
3. `thread1` executes its first instruction `x.store(2000)` before `thread2` is completely executed.

Now all values for x and y.

Possible values for the atomics(sequential consistency)

y	x	Values possible?	
:	-:		#
0	0	Yes	
11	0		
0	2000	Yes	
11	2000	Yes	

Let me verify my reasoning with CppMem.

6.4.3.1 CppMem

Here is the corresponding program in CppMem.

CppMem: atomics (sequential consistency

```
1   int main(){
2     atomic_int x = 0;
3     atomic_int y = 0;
4     {{{ {
5         x.store(2000);
6         y.store(11);
7       }
8     ||| {
9         y.load();
10        x.load();
11      }
12   }}}
13  }
```

First, a little bit of syntax. CppMem uses in lines 2 and 3 the `typedef atomic_int` for `std::atomic<int>`.

When I execute the program, I'm overwhelmed by the number of execution candidates.

Ongoing Optimization (sequential consistency)

There are 384 **(1)** possible execution candidates, only 6 of them are consistent. No candidate has a data race. I'm only interested in the six consistent executions and ignore the other 378 non-consistent executions. Non-consistent means, for example, that they do not respect the modification order of the memory model.

I use the interface **(2)** to get the six annotated graphs.

We already know that all values for x and y are possible except for y = 11 and x = 0. These results are possible because of sequential consistency. Now I'm curious, which interleaving of threads produces which values for x and y?

6.4.3.1.1 Execution for (y = 0, x = 0)
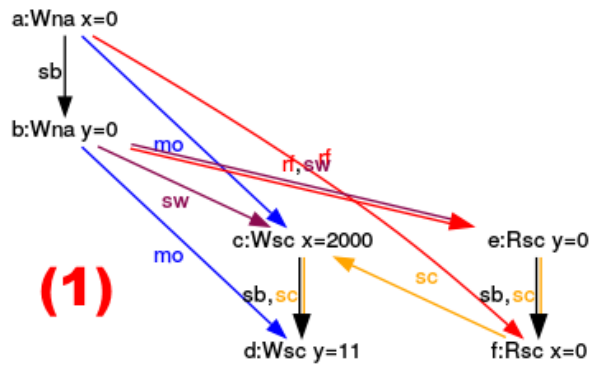
Execution for (y = 0, x = 0)

6.4.3.1.2 Executions for (y = 0, x = 2000)

Execution for (y = 0, x = 2000)

(3) Execution for (y = 0, x = 2000)

(4) Execution for (y = 0, x = 2000)
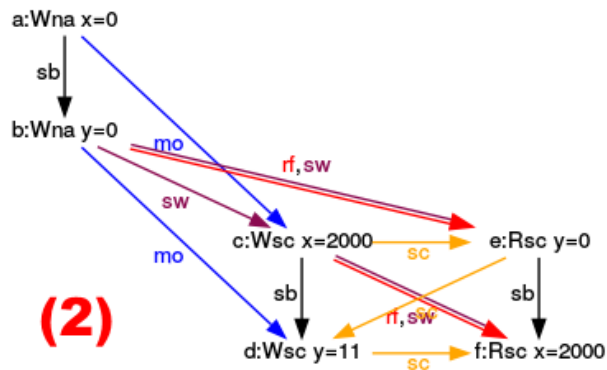
(5) Execution for (y = 0, x = 2000)
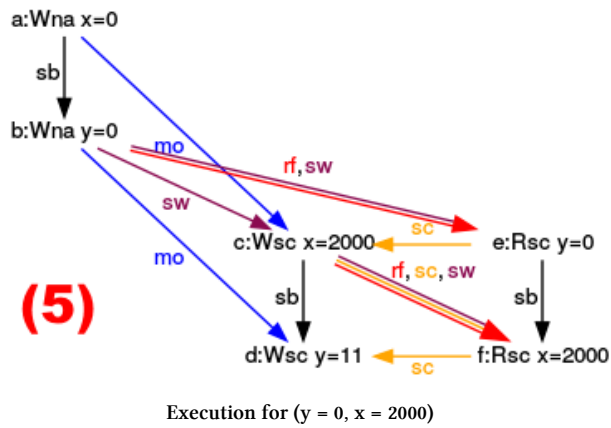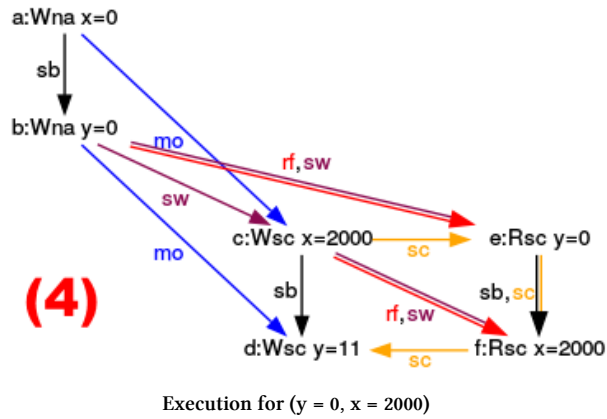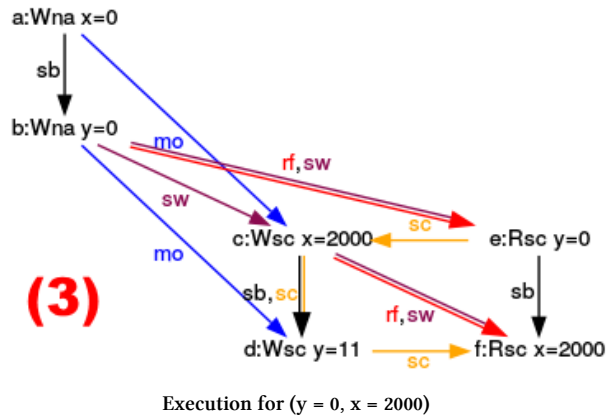
6.4.3.1.3 Execution for (y = 11, x = 2000)
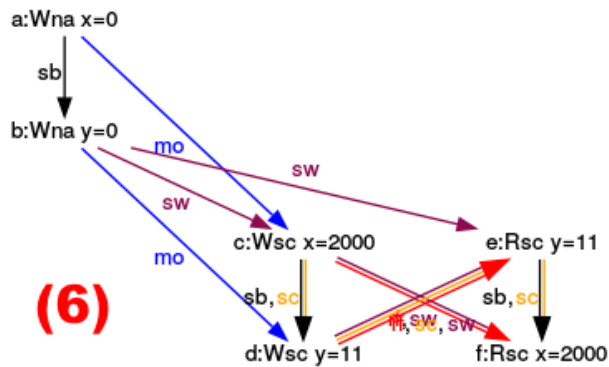
Execution for (y = 11, x = 2000)

I'm not done with my analysis. I'm interested in answering the question: Which sequence of instructions corresponds to which of the six graphs?

6.4.3.2 Sequence of Instructions

I have assigned to each sequence of instructions the corresponding graph.
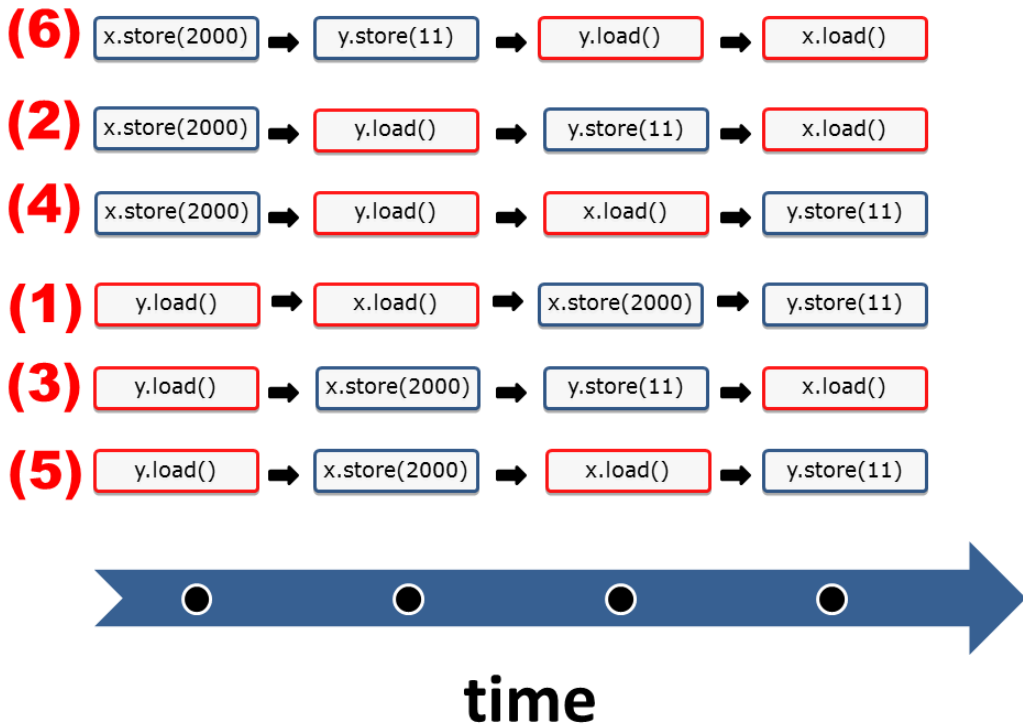
Sequence of instructions

Let me start with the more straightforward cases.

- **(1)**: It's quite simple to assign the graph (1) to the sequence (1). In the sequence (1) x and y have the values 0 because `y.load()` and `x.load()` are executed before the operations `x.store(2000)` and `y.store(11)`.

- **(6)**: The reasoning for the execution (6) is similar. y has the value 11 and x the value 2000 because all load operations happen after all store operations.

- **(2), (3), (4), (5)**: Now to the more interesting cases in which y has the value 0, and x has the value 2000. The yellow arrows (sc) in the graph are the key to my reasoning because they stand for the sequence of instructions. For example, let's look at execution (2).

    - **(2)**: The sequence of the yellow arrows (sc) in the graph (2) is: write x = 2000 ⇒ read y = 0 ⇒ write y = 11 ⇒ read x = 2000. This sequence corresponds to the sequence of instructions of the second interleaving of threads (2).

Let's break the sequential consistency with the acquire-release semantic.

6.4.4 CppMem: Atomics with Acquire-Release Semantic

The synchronization in the acquire-release semantic occurs between atomic operations on the same atomic. This synchronization of atomic operations on the same atomic is in contrast to the sequential consistency of synchronization between threads. Due to this fact, the acquire-release semantic is more lightweight and, therefore, faster.

Here is the program with acquire-release semantic.

ongoing optimization with atomics (acquire-release semantic)

```cpp
// ongoingOptimizationAcquireRelease.cpp

#include <atomic>
#include <iostream>
#include <thread>

std::atomic<int> x{0};
std::atomic<int> y{0};

void writing(){
  x.store(2000, std::memory_order_relaxed);
  y.store(11, std::memory_order_release);
}

void reading(){
  std::cout << y.load(std::memory_order_acquire) << " ";
  std::cout << x.load(std::memory_order_relaxed) << '\n';
}

int main(){
  std::thread thread1(writing);
  std::thread thread2(reading);
  thread1.join();
  thread2.join();
};
```

At first glance, you notice that all operations are atomic, so the program is well-defined. But the second glance shows more; the atomic operations on y are attached with the flag std::memory_order_release (line 12) and std::memory_order_acquire (line 16). In contrast to that, the atomic operations on x are annotated with std::memory_order_relaxed (lines 11 and 17). So there are no synchronization and ordering constraints for x. The answer to the possible values for x and y can only be given by y.

It holds:

- y.store(11,std::memory_order_release) **synchronizes-with** y.load(std::memory_order_acquire)

- `x.store(2000,std::memory_order_relaxed)` **is visible before** `y.store(11, std::memory_order_-release)`
- `y.load(std::memory_order_acquire)` **is visible before** `x.load(std::memory_order_relaxed)`

I elaborate a little bit more on these three statements. The key idea is that the store of `y` in line 12 synchronizes with the load of `y` in line 16. This is because the operations take place on the same atomic, and they use the acquire-release semantic. `y` uses `std::memory_order_release` in line 12 and `std::memory_order_acquire` in line 16. The pairwise operation on `y` has another imposing property. They establish a kind of barrier relative to `y`. So `x.store(2000, std::memory_order_relaxed)` cannot be executed after `y.store(std::memory_order_release)` and `x.load()` cannot be executed before `y.load()`.

The reasoning in the acquire-release semantic case is more sophisticated than in the case of the previous sequential consistency, but the possible values for `x` and `y` are the same. Only the combination `y == 11` and `x == 0` is not possible.

There are three different interleavings of the threads possible, which produce the three different combinations of the values `x` and `y`.

- `thread1` is executed before `thread2`.
- `thread2` is executed before `thread1`.
- `thread1` executes `x.store(2000)` before `thread2` is executed.

To make a long story short, here are all possible values for `x` and `y`.

Possible values for the atomics(acquire-release semantic)

y	x	Values possible?
0	0	Yes
11	0	
0	2000	Yes
11	2000	Yes

Once more. Let's verify our thinking with CppMem.

6.4.4.1 CppMem

Here is the corresponding program.

CppMem: atomics(acquire-release semantic

```
1   int main(){
2     atomic_int x = 0;
3     atomic_int y = 0;
4     {{{ {
5         x.store(2000, memory_order_relaxed);
6         y.store(11, memory_order_release);
7         }
8     ||| {
9         y.load(memory_order_acquire);
10        x.load(memory_order_relaxed);
11        }
12    }}}
13  }
```

We already know that all results are possible except for (y = 11, x = 0).

6.4.4.1.1 Possible executions

I only refer to the three graphs with consistent execution. The graphs show an acquire-release semantic between the store-release of y and the load-acquire operation of y. It makes no difference if the reading of y **(rf)** takes place in the main thread or a separate thread. The graphs also show the *synchronizes-with* relation using a **sw** annotated arrow.

6.4.4.1.2 Execution for (y = 0, x = 0)
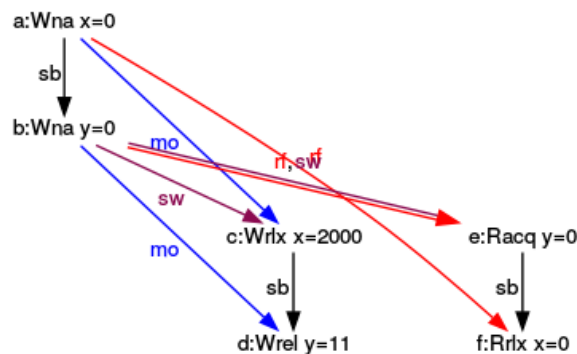
Execution for (y = 0, x = 0)

6.4.4.1.3 Execution for (y = 0, x = 2000)
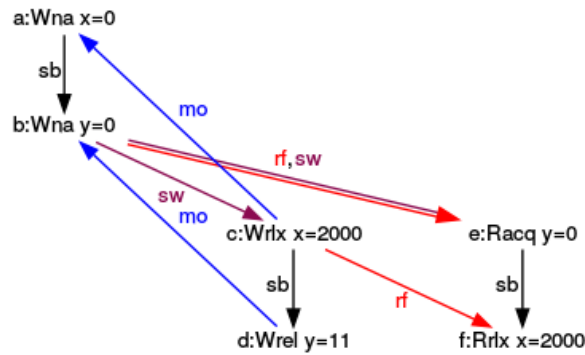
Execution for (y = 0, x = 2000)

6.4.4.1.4 Execution for (y = 11, x = 2000)
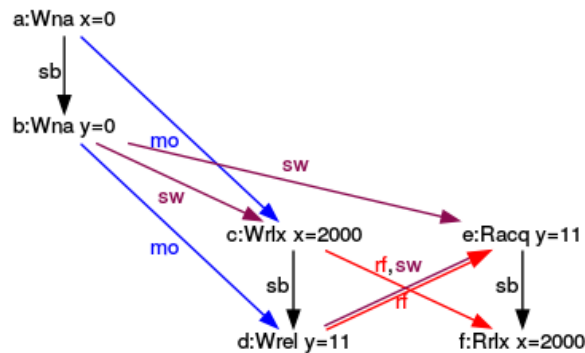
Execution for (y = 11, x = 2000)

x does not have to be atomic. This was my first and wrong assumption. See why.

6.4.5 CppMem: Atomics with Non-atomics

A typical misunderstanding in applying the acquire-release semantic is to assume that the acquire operation is waiting for the release operation. Based on this wrong assumption, you may think that x does not has to be an atomic variable, and we can further optimize the program.

ongoing optimization with atomics with non-atomics

```cpp
1   // ongoingOptimizationAcquireReleaseBroken.cpp
2
3   #include <atomic>
4   #include <iostream>
5   #include <thread>
6
7   int x = 0;
8   std::atomic<int> y{0};
9
10  void writing(){
11    x = 2000;
12    y.store(11, std::memory_order_release);
13  }
14
15  void reading(){
16    std::cout << y.load(std::memory_order_acquire) << " ";
17    std::cout << x << '\n';
18  }
19
20  int main(){
21    std::thread thread1(writing);
22    std::thread thread2(reading);
23    thread1.join();
24    thread2.join();
25  };
```

The program has a data race on x and, therefore, undefined behavior. The acquire-release semantic guarantees if y.store(11, std::memory_order_release) (line 12) is executed before y.load(std::memory_order_acquire) (line 16), that x = 2000 (line 11) is executed before the reading of x (line 17). If not the reading of x is executed simultaneously as the writing of x. So we have concurrent access to a shared variable, and one of them is a write operation. That is by definition a data race.

To make my point more clear, let me use CppMem.

6.4.5.1 CppMem

CppMem: atomics and non-atomics

```
1    int main(){
2      int x = 0;
3      atomic_int y = 0;
4      {{{ {
5          x = 2000;
6          y.store(11, memory_order_release);
7          }
8      ||| {
9          y.load(memory_order_acquire);
10         x;
11         }
12     }}}
13   }
```

The data race occurs when one thread is writing `x = 2000` and the other thread is reading `x`. We get a **dr** symbol (data race) on the corresponding yellow arrow.

A data race
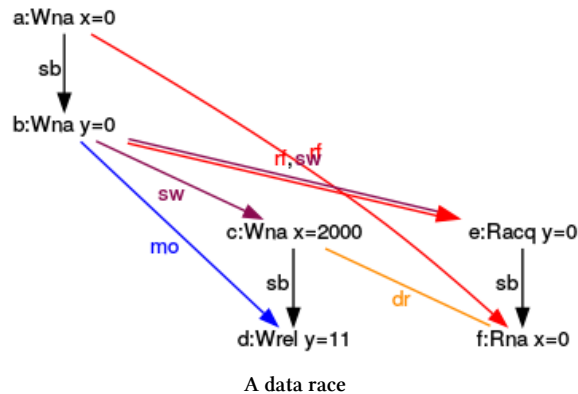
The final step in the process of ongoing optimization is still missing: relaxed semantic.

6.4.6 CppMem: Atomics with Relaxed Semantic

With the relaxed semantic, we have no synchronization and ordering constraints on atomic operations. Only the atomicity of the operations is guaranteed.

Ongoing optimization with atomics (relaxed semantic)

```cpp
// ongoingOptimizationRelaxedSemantic.cpp

#include <atomic>
#include <iostream>
#include <thread>

std::atomic<int> x{0};
std::atomic<int> y{0};

void writing(){
  x.store(2000, std::memory_order_relaxed);
  y.store(11, std::memory_order_relaxed);
}

void reading(){
  std::cout << y.load(std::memory_order_relaxed) << " ";
  std::cout << x.load(std::memory_order_relaxed) << '\n';
}

int main(){
  std::thread thread1(writing);
  std::thread thread2(reading);
  thread1.join();
  thread2.join();
};
```

For the relaxed semantic, my fundamental questions are straightforward to answer. These are my questions:

1. Does the program have well-defined behavior?

2. Which values for x and y are possible?

On the one hand, all operations on x and y are atomic, so the program is well-defined. On the other hand, there are no restrictions on the possible interleavings of threads. The result may be that thread2 sees the operations on thread1 in a different order. This is the first time in our process of ongoing optimization that thread2 can display x == 0 and y == 11 and, therefore, all combinations of x and y are possible.

Possible values for the atomics(relaxed semantic)

y	x	Values possible?
0	0	Yes
11	0	Yes
0	2000	Yes
11	2000	Yes

Now I'm curious how the graph of CppMem looks like for `x == 0` and `y == 11`?

6.4.6.1 CppMem

CppMem: atomics (relaxed semantic)

```cpp
int main(){
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ {
      x.store(2000, memory_order_relaxed);
      y.store(11, memory_order_relaxed);
      }
  ||| {
      y.load(memory_order_relaxed);
      x.load(memory_order_relaxed);
      }
  }}}
}
```

That was the CppMem program. Now to the graph that produces the counter-intuitive behavior.
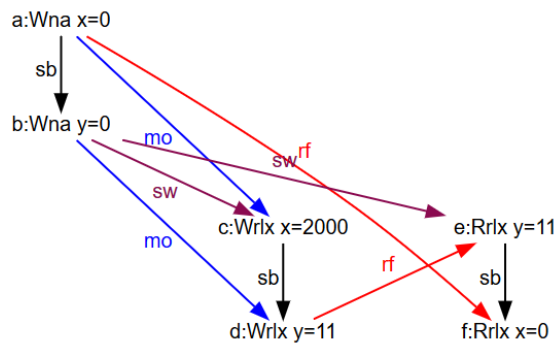
Execution for (y = 11, x = 0)

x reads the value 0 (line 10) but y reads the value 11 (line 9). This happens, although the writing of x (line 5) is sequenced before the writing of y (line 6).

6.4.7 Conclusion

Using a small program and successively improve it was quite enlightening. First, with each step, more interleavings of threads were possible; therefore, more different values for x and y were possible. Second, the challenge of the program increases with each improvement. Even for this small program, CppMem provides invaluable services.

6.5 Fast Synchronization of Threads

When you want to synchronize threads more than once, you can use condition variables, `std::atomic_flag`, `std::atomic<bool>`, or semaphores. In this section, I want to answer the question: Which variant is the fastest?

To get comparable numbers, I implement a ping-pong game. One thread executes a `ping` function, and the other thread a `pong` function. For simplicity reasons, I call the thread executing the `ping` function the ping thread and the other thread the pong thread. The ping thread waits for the notification of the pong thread and sends the notification back to the pong thread. The game stops after 1'000'000 ball changes. I perform each game five times to get comparable performance numbers.

About the Numbers

I made my performance test at the end of 2020 with the brand new Visual Studio compiler 19.28 because it already supported synchronization with atomics (`std::atomic_flag` and `std::atomic`) and semaphores. Additionally, I compiled the examples with maximum optimization (`/Ox`). The performance number should only give a rough idea of the relative performance of the various ways to synchronize threads. When you want the exact number on your platform, you have to repeat the tests.

Let me start the comparison with condition variables.

6.5.1 Condition Variables

Multiple time synchronization with a condition variable

```cpp
1  // pingPongConditionVariable.cpp
2
3  #include <condition_variable>
4  #include <iostream>
5  #include <atomic>
6  #include <thread>
7
8  bool dataReady{false};
9
10 std::mutex mutex_;
11 std::condition_variable condVar1;
12 std::condition_variable condVar2;
13
14 std::atomic<int> counter{};
15 constexpr int countlimit = 1'000'000;
16
17 void ping() {
```
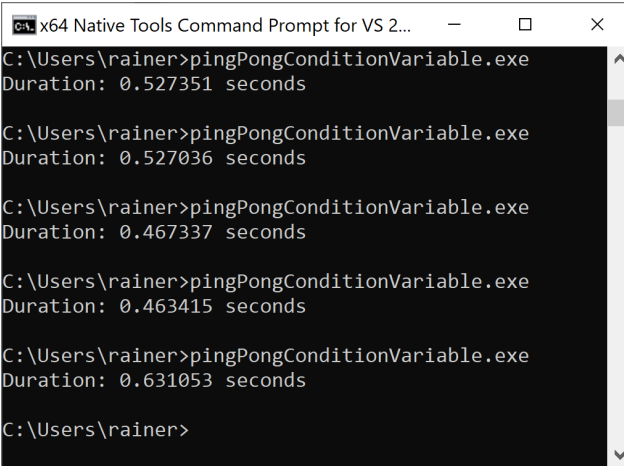
```
18
19      while(counter <= countlimit) {
20          {
21              std::unique_lock<std::mutex> lck(mutex_);
22              condVar1.wait(lck, []{return dataReady == false;});
23              dataReady = true;
24          }
25          ++counter;
26          condVar2.notify_one();
27      }
28  }
29
30  void pong() {
31
32      while(counter <= countlimit) {
33          {
34              std::unique_lock<std::mutex> lck(mutex_);
35              condVar2.wait(lck, []{return dataReady == true;});
36              dataReady = false;
37          }
38          condVar1.notify_one();
39      }
40
41  }
42
43  int main(){
44
45      auto start = std::chrono::system_clock::now();
46
47      std::thread t1(ping);
48      std::thread t2(pong);
49
50      t1.join();
51      t2.join();
52
53      std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
54      std::cout << "Duration: " << dur.count() << " seconds" << '\n';
55  }
```

I use two condition variables in the program: condVar1 and condVar2. The ping thread waits for the notification of condVar1 and sends its notification with condVar2. dataReady protects against spurious and lost wakeups. The ping-pong game ends when counter reaches the countlimit. The nofication_-one calls (lines 26 and 38) and the counter are thread-safe and are, therefore, outside the critical region.

These are the numbers.

Multiple time synchronization with condition variables

The average execution time is 0.52 seconds.

Porting this workflow to `std::atomic_flag` in C++20 is straightforward.

6.5.2 `std::atomic_flag`

Here is the same workflow using first two and then one atomic flag.

6.5.2.1 Two Atomic Flags

In the following program, I replace the waiting on the condition variable with the waiting on the atomic flag and the notification of the condition variable with the atomic flag setting followed by the notification.

Multiple time synchronization with two atomic flags

```cpp
1   // pingPongAtomicFlags.cpp
2
3   #include <iostream>
4   #include <atomic>
5   #include <thread>
6
7   std::atomic_flag condAtomicFlag1{};
8   std::atomic_flag condAtomicFlag2{};
9
10  std::atomic<int> counter{};
11  constexpr int countlimit = 1'000'000;
```
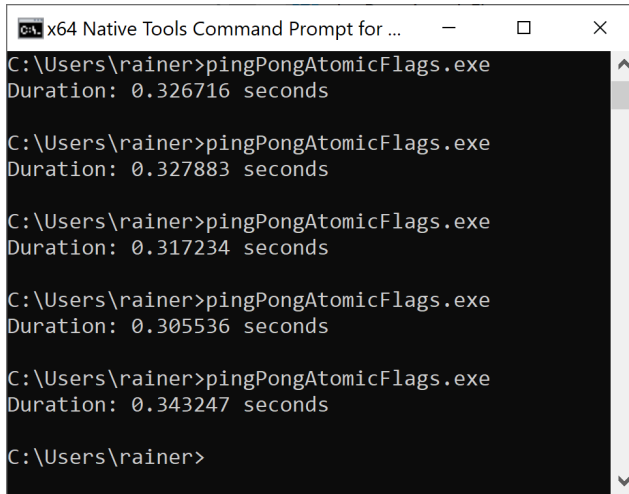
```
12
13    void ping() {
14        while(counter <= countlimit) {
15            condAtomicFlag1.wait(false);
16            condAtomicFlag1.clear();
17
18            ++counter;
19
20            condAtomicFlag2.test_and_set();
21            condAtomicFlag2.notify_one();
22        }
23    }
24
25    void pong() {
26        while(counter <= countlimit) {
27            condAtomicFlag2.wait(false);
28            condAtomicFlag2.clear();
29
30            condAtomicFlag1.test_and_set();
31            condAtomicFlag1.notify_one();
32        }
33    }
34
35    int main() {
36
37         auto start = std::chrono::system_clock::now();
38
39        condAtomicFlag1.test_and_set();
40        std::thread t1(ping);
41        std::thread t2(pong);
42
43        t1.join();
44        t2.join();
45
46        std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
47        std::cout << "Duration: " << dur.count() << " seconds" << '\n';
48
49    }
```

A call condAtomicFlag1.wait(false) (line 15) blocks if the atomic flag's value is false. On the contrary, it returns if condAtomicFlag1 has the value true. The boolean value serves as a kind of predicate and must, therefore, set back to false (line 15). Before the notification (line 21) is sent to the pong thread, condAtomicFlag1 is set to true (line 20). The initial setting of condAtomicFlag1 (line 29) to true starts the game.

Thanks to `std::atomic_flag` the game ends earlier.

Multiple time synchronization with two atomic flags

On average, a game takes 0.32 seconds.

When you analyze the program, you may recognize that one atomics flag is sufficient for the workflow.

6.5.2.2 One Atomic Flags

Using one atomic flag makes the workflow easier to understand.

Multiple time synchronization with one atomic flag
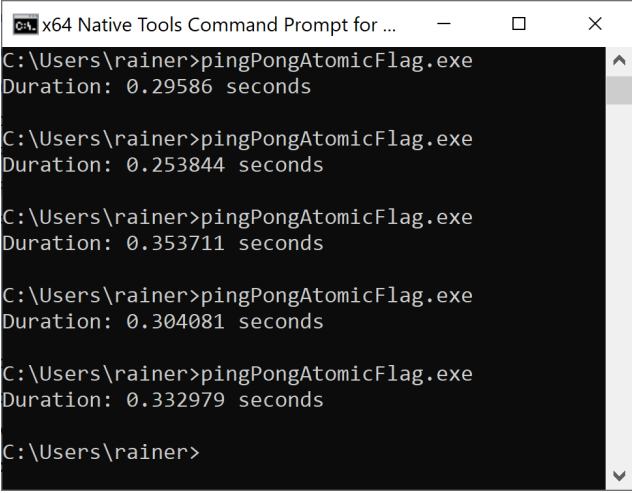
```cpp
1  // pingPongAtomicFlag.cpp
2
3  #include <iostream>
4  #include <atomic>
5  #include <thread>
6
7  std::atomic_flag condAtomicFlag{};
8
9  std::atomic<int> counter{};
10 constexpr int countlimit = 1'000'000;
11
12 void ping() {
13     while(counter <= countlimit) {
14         condAtomicFlag.wait(true);
15         condAtomicFlag.test_and_set();
```

```
16
17              ++counter;
18
19              condAtomicFlag.notify_one();
20          }
21  }
22
23  void pong() {
24      while(counter <= countlimit) {
25          condAtomicFlag.wait(false);
26          condAtomicFlag.clear();
27          condAtomicFlag.notify_one();
28      }
29  }
30
31  int main() {
32
33      auto start = std::chrono::system_clock::now();
34
35      condAtomicFlag.test_and_set();
36      std::thread t1(ping);
37      std::thread t2(pong);
38
39      t1.join();
40      t2.join();
41
42      std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
43      std::cout << "Duration: " << dur.count() << " seconds" << '\n';
44
45  }
```

In this case, the ping thread blocks on true, but the pong thread blocks on false. From the performance perspective, using one or two atomic flags makes no difference.

Multiple time synchronization with one atomic flag

The average execution time is 0.31 seconds.

I used in this example `std::atomic_flag` such as an atomic boolean. Let's give it another try with `std::atomic<bool>`.

6.5.3 `std::atomic<bool>`

The following C++20 implementation is based on `std::atomic`.

Multiple time synchronization with an atomic bool
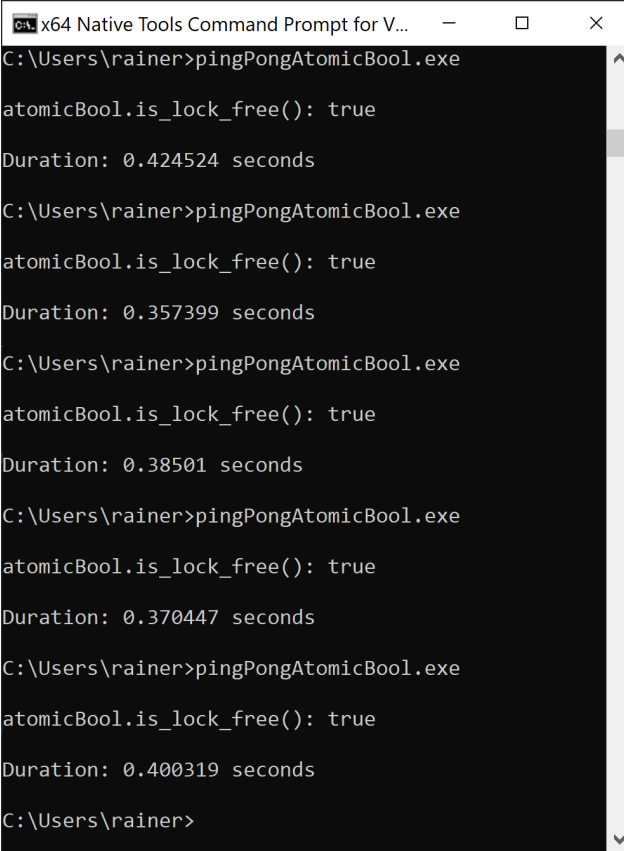
```cpp
// pingPongAtomicBool.cpp

#include <iostream>
#include <atomic>
#include <thread>

std::atomic<bool> atomicBool{};

std::atomic<int> counter{};
constexpr int countlimit = 1'000'000;

void ping() {
    while(counter <= countlimit) {
        atomicBool.wait(true);
        atomicBool.store(true);

        ++counter;
```

```
18
19          atomicBool.notify_one();
20      }
21  }
22
23  void pong() {
24      while(counter <= countlimit) {
25          atomicBool.wait(false);
26          atomicBool.store(false);
27          atomicBool.notify_one();
28      }
29  }
30
31  int main() {
32
33      std::cout << std::boolalpha << '\n';
34
35      std::cout << "atomicBool.is_lock_free(): "
36                << atomicBool.is_lock_free()  << '\n';
37
38      std::cout << '\n';
39
40      auto start = std::chrono::system_clock::now();
41
42      atomicBool.store(true);
43      std::thread t1(ping);
44      std::thread t2(pong);
45
46      t1.join();
47      t2.join();
48
49      std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
50      std::cout << "Duration: " << dur.count() << " seconds" << '\n';
51
52  }
```

`std::atomic<bool>` can internally use a locking mechanism such as a mutex. My Windows runtime is lock-free.

Multiple time synchronization with an atomic bool

On average, the execution time is 0.38 seconds.

From the readability perspective, this implementation based on `std::atomic` is straightforward to understand. This observation also holds for the following implementation of the ping-pong game based on semaphores.

6.5.4 Semaphores

Semaphores promises to be faster than condition variables. Let's see if this is true.

Multiple time synchronization with semaphores

```cpp
1  // pingPongSemaphore.cpp
2
3  #include <iostream>
4  #include <semaphore>
5  #include <thread>
6
7  std::counting_semaphore<1> signal2Ping(0);
8  std::counting_semaphore<1> signal2Pong(0);
9
10  std::atomic<int> counter{};
11  constexpr int countlimit = 1'000'000;
12
13  void ping() {
14      while(counter <= countlimit) {
15          signal2Ping.acquire();
16          ++counter;
17          signal2Pong.release();
18      }
19  }
20
21  void pong() {
22      while(counter <= countlimit) {
23          signal2Pong.acquire();
24          signal2Ping.release();
25      }
26  }
27
28  int main() {
29
30      auto start = std::chrono::system_clock::now();
31
32      signal2Ping.release();
33      std::thread t1(ping);
34      std::thread t2(pong);
35
36      t1.join();
37      t2.join();
38
39      std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
40      std::cout << "Duration: " << dur.count() << " seconds" << '\n';
41
42  }
```

The program `pingPongsemaphore.cpp` uses two semaphores: `signal2Ping` and `signal2Pong` (lines 7 and 8). Both can have the two values 0 and 1 and are initialized with 0. This means when the value is 0 for the semaphore `signal2Ping`: a call `signal2Ping.release()` (lines 24 and 32) set the value to 1 and is, therefore, a notification; a `signal2Ping.acquire()` (line 15) call blocks until the value becomes 1. The same argumentation holds for the second semaphore `signal2Pong`.

Multiple time synchronization with semaphores

On average, the execution time is 0.33 seconds.

6.5.5 All Numbers

As expected, condition variables are the slowest way, and atomic flag the fastest way to synchronize threads. The performance of a `std::atomic<bool>` is in between. There is a downside with `std::atomic<bool>`. `std::atomic_flag` is the only atomic data type that is always lock-free. Semaphores impressed me most because they are nearly as fast as atomic flags.

	Execution Time				
	Condition Variables	**Two Atomic Flags**	**One Atomic Flag**	**Atomic Boolean**	**Semaphores**
Execution Time	0.52	0.32	0.31	0.38	0.33

6.6 Variations of Futures

Before I create variations of the future from section `co_return`, we should understand its control flow. Comments make the control flow transparent. Additionally, I provide a link to the presented programs on online compilers.

Control flow of an eager future

```cpp
1    // eagerFutureWithComments.cpp
2
3    #include <coroutine>
4    #include <iostream>
5    #include <memory>
6
7    template<typename T>
8    struct MyFuture {
9        std::shared_ptr<T> value;
10       MyFuture(std::shared_ptr<T> p): value(p) {
11           std::cout << "    MyFuture::MyFuture" << '\n';
12       }
13       ~MyFuture() {
14           std::cout << "    MyFuture::~MyFuture" << '\n';
15       }
16       T get() {
17           std::cout << "    MyFuture::get" << '\n';
18           return *value;
19       }
20
21       struct promise_type {
22           std::shared_ptr<T> ptr = std::make_shared<T>();
23           promise_type() {
24               std::cout << "        promise_type::promise_type" << '\n';
25           }
26           ~promise_type() {
27               std::cout << "        promise_type::~promise_type" << '\n';
28           }
29           MyFuture<T> get_return_object() {
30               std::cout << "        promise_type::get_return_object" << '\n';
31               return ptr;
32           }
33           void return_value(T v) {
34               std::cout << "        promise_type::return_value" << '\n';
35               *ptr = v;
36           }
37           std::suspend_never initial_suspend() {
```

```
38                std::cout << "           promise_type::initial_suspend" << '\n';
39                return {};
40            }
41            std::suspend_never final_suspend() noexcept {
42                std::cout << "           promise_type::final_suspend" << '\n';
43                return {};
44            }
45            void unhandled_exception() {
46                std::exit(1);
47            }
48        };
49    };
50
51    MyFuture<int> createFuture() {
52        std::cout << "createFuture" << '\n';
53        co_return 2021;
54    }
55
56    int main() {
57
58        std::cout << '\n';
59
60        auto fut = createFuture();
61        auto res = fut.get();
62        std::cout << "res: " << res << '\n';
63
64        std::cout << '\n';
65
66    }
```

The call `createFuture` (line 60) causes the creating of the instance of `MyFuture` (line 59). Before `MyFuture`'s constructor call (line 10) is completed, the promise `promise_type` is created, executed, and destroyed (lines 20 - 48). The promise uses in each step of its control flow the awaitable `std::suspend_never` (lines 36 and 40) and, hence, never pauses. To save the result of the promise for the later `fut.get()` call (line 60), it has to be allocated. Furthermore, the used `std::shared_ptrs` ensure (lines 9 and 21) that the program does not cause a memory leak. As a local, `fut` goes out of scope in line 65, and the C++ run time calls its destructor.

You can try out the program on the Compiler Explorer[20].

_____

[20]https://godbolt.org/z/Y9naEx

```
                promise_type::promise_type
                promise_type::get_return_object
                promise_type::initial_suspend
        createFuture
                promise_type::return_value
                promise_type::final_suspend
                promise_type::~promise_type
            MyFuture::MyFuture
            MyFuture::get
        res: 2021


            MyFuture::~MyFuture
```

An eager future

The presented coroutine runs immediately and is, therefore, eager. Furthermore, the coroutine runs in the thread of the caller.

Let's make the coroutine lazy.

6.6.1 A Lazy Future

A lazy future is a future that runs only if asked for the value. Let's see what I have to change in the eager coroutine, presented in eagerFutureWithComments.cpp, to make it lazy.

Control flow of a lazy future

```cpp
1   // lazyFuture.cpp
2
3   #include <coroutine>
4   #include <iostream>
5   #include <memory>
6
7   template<typename T>
8   struct MyFuture {
9       struct promise_type;
10      using handle_type = std::coroutine_handle<promise_type>;
11
12      handle_type coro;
13
14      MyFuture(handle_type h): coro(h) {
15          std::cout << "    MyFuture::MyFuture" << '\n';
16      }
```

```
17        ~MyFuture() {
18            std::cout << "    MyFuture::~MyFuture" << '\n';
19            if ( coro ) coro.destroy();
20        }
21
22        T get() {
23            std::cout << "    MyFuture::get" << '\n';
24            coro.resume();
25            return coro.promise().result;
26        }
27
28        struct promise_type {
29            T result;
30            promise_type() {
31                std::cout << "        promise_type::promise_type" << '\n';
32            }
33            ~promise_type() {
34                std::cout << "        promise_type::~promise_type" << '\n';
35            }
36            auto get_return_object() {
37                std::cout << "        promise_type::get_return_object" << '\n';
38                return MyFuture{handle_type::from_promise(*this)};
39            }
40            void return_value(T v) {
41                std::cout << "        promise_type::return_value" << '\n';
42                result = v;
43            }
44            std::suspend_always initial_suspend() {
45                std::cout << "        promise_type::initial_suspend" << '\n';
46                return {};
47            }
48            std::suspend_always final_suspend() noexcept {
49                std::cout << "        promise_type::final_suspend" << '\n';
50                return {};
51            }
52            void unhandled_exception() {
53                std::exit(1);
54            }
55        };
56    };
57
58    MyFuture<int> createFuture() {
59        std::cout << "createFuture" << '\n';
60        co_return 2021;
61    }
```

```
62
63    int main() {
64
65        std::cout << '\n';
66
67        auto fut = createFuture();
68        auto res = fut.get();
69        std::cout << "res: " << res << '\n';
70
71        std::cout << '\n';
72
73    }
```

Let's first study the promise. The promise always suspends at the beginning (line 44) and the end (line 48). Furthermore, the member function get_return_object (line 36) creates the return object that is returned to the caller of the coroutine createFuture (line 58). The future MyFuture is more interesting. It has a handle coro (line 12) to the promise. MyFuture uses the handle to manage the promise. It resumes the promise (line 24), asks the promise for the result (line 25), and finally destroys it (line 19). The resumption of the coroutine is necessary because it never runs automatically (line 44). When the client invokes fut.get() (line 68) to ask for the result of the future, it implicitly resumes the promise (line 24).

You can try out the program on the Compiler Explorer[21].

```
        promise_type::promise_type
        promise_type::get_return_object
    MyFuture::MyFuture
        promise_type::initial_suspend
    MyFuture::get
createFuture
        promise_type::return_value
        promise_type::final_suspend
res: 2021

    MyFuture::~MyFuture
        promise_type::~promise_type
```

A lazy future

What happens if the client is not interested in the result of the future? Let's try it out.

---

[21]https://godbolt.org/z/EejWcj

The client does not resume the coroutine

```cpp
int main() {

    std::cout << '\n';

    auto fut = createFuture();
    // auto res = fut.get();
    // std::cout << "res: " << res << '\n';

    std::cout << '\n';

}
```

As you may guess, the promise never runs, and the member functions `return_value` and `final_suspend` are not executed.

```
        promise_type::promise_type
        promise_type::get_return_object
MyFuture::MyFuture
        promise_type::initial_suspend


MyFuture::~MyFuture
        promise_type::~promise_type
```

A lazy future that is not started

⚠ Lifetime Challenges of Coroutines

One of the challenges of dealing with coroutines is to handle the lifetime of the coroutine. In the previous program `eagerFutureWithComments.cpp`, I stored the coroutine result in a `std::shared_ptr`. This is critical because the coroutine is executed eagerly.

In this program `lazyFuture.cpp`, the call `final_suspend` always suspends (line 48): `std::suspend_always final_suspend()`. Consequently, the promise outlives the client, and a `std::shared_ptr` is not necessary anymore. Returning `std::suspend_never` from the function `final_suspend` would cause, in this case, undefined behavior because the client would outlive the promise. Hence, the lifetime of the result ends, bevor the client asks for it.

Let's vary the coroutine further and run the promise in a separate thread.

6.6.2 Execution on Another Thread

The coroutine is fully suspended before entering the coroutine `createFuture` (line 67), because the member function `initial_suspend` returns `std::suspend_always` (line 52). Consequently, the promise can run on another thread.

Executing the promise on another thread

```cpp
// lazyFutureOnOtherThread.cpp

#include <coroutine>
#include <iostream>
#include <memory>
#include <thread>

template<typename T>
struct MyFuture {
    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;
    handle_type coro;

    MyFuture(handle_type h): coro(h) {}
    ~MyFuture() {
        if ( coro ) coro.destroy();
    }

    T get() {
        std::cout << "    MyFuture::get:  "
                  << "std::this_thread::get_id(): "
                  << std::this_thread::get_id() << '\n';

        std::thread t([this] { coro.resume(); });
        t.join();
        return coro.promise().result;
    }

    struct promise_type {
        promise_type(){
            std::cout << "        promise_type::promise_type:  "
                      << "std::this_thread::get_id(): "
                      << std::this_thread::get_id() << '\n';
        }
        ~promise_type(){
            std::cout << "        promise_type::~promise_type:  "
                      << "std::this_thread::get_id(): "
                      << std::this_thread::get_id() << '\n';
```

```
39              }
40
41              T result;
42              auto get_return_object() {
43                  return MyFuture{handle_type::from_promise(*this)};
44              }
45              void return_value(T v) {
46                  std::cout << "          promise_type::return_value:  "
47                            << "std::this_thread::get_id(): "
48                            << std::this_thread::get_id() << '\n';
49                  std::cout << v << '\n';
50                  result = v;
51              }
52              std::suspend_always initial_suspend() {
53                  return {};
54              }
55              std::suspend_always final_suspend() noexcept {
56                  std::cout << "          promise_type::final_suspend:  "
57                            << "std::this_thread::get_id(): "
58                            << std::this_thread::get_id() << '\n';
59                  return {};
60              }
61              void unhandled_exception() {
62                  std::exit(1);
63              }
64          };
65      };
66
67      MyFuture<int> createFuture() {
68          co_return 2021;
69      }
70
71      int main() {
72
73          std::cout << '\n';
74
75          std::cout << "main:  "
76                    << "std::this_thread::get_id(): "
77                    << std::this_thread::get_id() << '\n';
78
79          auto fut = createFuture();
80          auto res = fut.get();
81          std::cout << "res: " << res << '\n';
82
83          std::cout << '\n';
```

```
84
85   }
```

I added a few comments to the program that show the id of the running thread. The program `lazyFutureOnOtherThread.cpp` is quite similar to the previous program `lazyFuture.cpp`. The main difference is the member function `get` (line 19). The call `std::thread t([this] { coro.resume(); });` (line 24) resumes the coroutine on another thread.

You can try out the program on the Wandbox[22] online compiler.

```
main:  std::this_thread::get_id(): 139819561723776
        promise_type::promise_type:  std::this_thread::get_id(): 139819561723776
    MyFuture::get:  std::this_thread::get_id(): 139819561723776
        promise_type::return_value:  std::this_thread::get_id(): 139819456755456
        promise_type::final_suspend:  std::this_thread::get_id(): 139819456755456
res: 2021

        promise_type::~promise_type:  std::this_thread::get_id(): 139819561723776
```

Execution on another thread

I want to add a few additional remarks about the member function `get`. It is crucial that the promise, resumed in a separate thread, finishes before it returns `coro.promise().result`.

The member function `get` using `std::thread`

```cpp
T get() {
    std::thread t([this] { coro.resume(); });
    t.join();
    return coro.promise().result;
}
```

Where I to join the thread `t` after the call `return coro.promise().result`, the program would have undefined behavior. In the following implementation of the function `get`, I use a `std::jthread`. Since `std::jthread` automatically joins when it goes out of scope. This is too late.

The member function `get` using `std::jthread`

```cpp
T get() {
    std::jthread t([this] { coro.resume(); });
    return coro.promise().result;
}
```

In this case, the client likely gets its result before the promise prepares it using the member function `return_value`. Now, `result` has an arbitrary value, and therefore so does `res`.

---

[22]https://wandbox.org/permlink/jFVVj80Gxu6bnNkc

```
main:  std::this_thread::get_id(): 139913381070720
       promise_type::promise_type:  std::this_thread::get_id(): 139913381070720
   MyFuture::get:  std::this_thread::get_id(): 139913381070720
       promise_type::return_value:  std::this_thread::get_id(): 139913276102400
       promise_type::final_suspend:  std::this_thread::get_id(): 139913276102400
res: -1

       promise_type::~promise_type:  std::this_thread::get_id(): 139913381070720
```

<p align="center">Execution on another thread</p>

There are other possibilities to ensure that the thread is done before the return call.

- Create a `std::jthread` in its scope.

`std::jthread` has its own scope

```cpp
T get() {
    {
        std::jthread t([this] { coro.resume(); });
    }
    return coro.promise().result;
}
```

- Make `std::jthread` a temporary object

`std::jthread` as a temporary

```cpp
T get() {
    std::jthread([this] { coro.resume(); });
    return coro.promise().result;
}
```

In particular, I don't like the last solution because it may take you a few seconds to recognize that I just called the constructor of `std::jthread`.

6.7 Modification and Generalization of a Generator

Before I modify and generalize the generator for an infinite data stream, I want to present it as a
starting point of our journey. I intentionally put many output operations in the source code and only
ask for three values. This simplification and visualization should help to understand the control flow.

Generator generating an infinite data stream

```cpp
// infiniteDataStreamComments.cpp

#include <coroutine>
#include <memory>
#include <iostream>

template<typename T>
struct Generator {

    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;

    Generator(handle_type h): coro(h) {
        std::cout << "        Generator::Generator" << '\n';
    }
    handle_type coro;

    ~Generator() {
        std::cout << "        Generator::~Generator" << '\n';
        if ( coro ) coro.destroy();
    }
    Generator(const Generator&) = delete;
    Generator& operator = (const Generator&) = delete;
    Generator(Generator&& oth): coro(oth.coro) {
        oth.coro = nullptr;
    }
    Generator& operator = (Generator&& oth) {
        coro = oth.coro;
        oth.coro = nullptr;
        return *this;
    }
    int getNextValue() {
        std::cout << "        Generator::getNextValue" << '\n';
        coro.resume();
        return coro.promise().current_value;
    }
    struct promise_type {
```

```
38              promise_type() {
39                  std::cout << "              promise_type::promise_type" << '\n';
40              }
41
42              ~promise_type() {
43                  std::cout << "              promise_type::~promise_type" << '\n';
44              }
45
46              std::suspend_always initial_suspend() {
47                  std::cout << "              promise_type::initial_suspend" << '\n';                    \
48
49                  return {};
50              }
51              std::suspend_always final_suspend() noexcept {
52                  std::cout << "              promise_type::final_suspend" << '\n';
53                  return {};
54              }
55              auto get_return_object() {
56                  std::cout << "              promise_type::get_return_object" << '\n';
57                  return Generator{handle_type::from_promise(*this)};
58              }
59
60              std::suspend_always yield_value(int value) {
61                  std::cout << "              promise_type::yield_value" << '\n';
62                  current_value = value;
63                  return {};
64              }
65              void return_void() {}
66              void unhandled_exception() {
67                  std::exit(1);
68              }
69
70              T current_value;
71          };
72
73      };
74
75      Generator<int> getNext(int start = 10, int step = 10) {
76          std::cout << "    getNext: start" << '\n';
77          auto value = start;
78          while (true) {
79              std::cout << "    getNext: before co_yield" << '\n';
80              co_yield value;
81              std::cout << "    getNext: after co_yield" << '\n';
82              value += step;
```

```
83        }
84    }
85
86    int main() {
87
88        auto gen = getNext();
89        for (int i = 0; i <= 2; ++i) {
90            auto val = gen.getNextValue();
91            std::cout << "main: " << val << '\n';
92        }
93
94    }
```

Executing the program on the Compiler Explorer[23] makes the control flow transparent.

```
                    promise_type::promise_type
                    promise_type::get_return_object
                Generator::Generator
                    promise_type::initial_suspend
                Generator::getNextValue
            getNext: start
            getNext: before co_yield
                    promise_type::yield_value
        main: 10
                Generator::getNextValue
            getNext: after co_yield
            getNext: before co_yield
                    promise_type::yield_value
        main: 20
                Generator::getNextValue
            getNext: after co_yield
            getNext: before co_yield
                    promise_type::yield_value
        main: 30
                Generator::~Generator
                    promise_type::~promise_type
```

Generator generating an infinite data stream

Let's analyze the control flow.

The call getNext() (line 87) triggers the creation of the Generator<int>. First, the promise_type (line 38) is created, and the following get_return_object call (line 54) creates the generator (line 56) and

[23]https://godbolt.org/z/cTW9Gq

stores it in a local variable. The result of this call is returned to the caller when the coroutine is suspended the first time. The initial suspension happens immediately (line 48). Because the member function call `initial_suspend` returns an Awaitable `std::suspend_always` (line 48), the control flow continues with the coroutine `getNext` until the instruction `co_yield value` (line 79). This call is mapped to the call `yield_value(int value)` (line 59) and the current value is prepared `current_value = value` (line 61). The member function `yield_value(int value)` returns the Awaitable `std::suspend_always` (line 59). Consequently, the execution of the coroutine pauses, and the control flow goes back to the `main` function, and the for loop starts (line 89). The call `gen.getNextValue()` (line 89) starts the execution of the coroutine by resuming the coroutine, using `coro.resume()` (line 34). Further, the function `getNextValue()` returns the current value that was prepared using the previously invoked member function `yield_value(int value)` (line 59). Finally, the generated number is displayed in line 90 and the for loop continues. In the end, the generator and the promise are destructed.

After this detailed analysis, I want to make a first modification of the control flow.

6.7.1 Modifications

My code snippets and line numbers are all based on the previous program `infiniteDataStreamComments.cpp`. I only show the modifications.

6.7.1.1 The Coroutine is Not Resumed

When I disable the resumption of the coroutine (`gen.getNextValue()` in line 89) and the display of its value (line 90), the coroutine immediately pauses.

Not resuming the coroutine

```cpp
int main() {

    auto gen = getNext();
    for (int i = 0; i <= 2; ++i) {
        // auto val = gen.getNextValue();
        // std::cout << "main: " << val << '\n';
    }

}
```

The coroutine never runs. Consequently, the generator and its promise are created and destroyed.

```
    promise_type::promise_type
    promise_type::get_return_object
Generator::Generator
    promise_type::initial_suspend
Generator::~Generator
    promise_type::~promise_type
```

Not resuming the coroutine

6.7.1.2 `initial_suspend` **Never Suspends**

In the program, the member function `initial_suspend` returns the Awaitable `std::suspend_always` (line 46). As its name suggests, the Awaitable `std::suspends_always` causes the coroutine to pause immediately. Let me return `std::suspend_never` instead of `std::suspend_always`.

`initial_suspend` **suspends never**

```cpp
std::suspend_never initial_suspend() {
    std::cout << "                    promise_type::initial_suspend" << '\n';
    return {};
}
```

In this case, the coroutine runs immediately and pauses when the function `yield_value` (line 59) is invoked. A subsequent call `gen.getNextValue()` (line 89) resumes the coroutine and triggers the execution of the member function `yield_value` once more. The result is that the start value `10` is ignored, and the coroutine returns the values `20`, `30`, and `40`.

```
              promise_type::promise_type
              promise_type::get_return_object
         Generator::Generator
              promise_type::initial_suspend
     getNext: start
     getNext: before co_yield
              promise_type::yield_value
         Generator::getNextValue
     getNext: after co_yield
     getNext: before co_yield
              promise_type::yield_value
main: 20
         Generator::getNextValue
     getNext: after co_yield
     getNext: before co_yield
              promise_type::yield_value
main: 30
         Generator::getNextValue
     getNext: after co_yield
     getNext: before co_yield
              promise_type::yield_value
main: 40
         Generator::~Generator
              promise_type::~promise_type
```

Don't Resuming the Coroutine

6.7.1.3 `yield_value` Never Suspends

The member function `yield_value` (line 59) is triggered by the call `co_yield value` and prepares the `current_value` (line 61). The function returns the Awaitable `std::suspend_always` (line 62) and, therefore, pauses the coroutine. Consequently, a subsequent call `gen.getNextValue` (line 89) has to resume the coroutine. When I change the return value of the member function `yield_value` to `std::suspend_never`, let me see what happens.

`yield_value` never suspends

```cpp
std::suspend_never yield_value(int value) {
    std::cout << "                promise_type::yield_value" << '\n';
    current_value = value;
    return {};
}
```

As you may guess, the while loop (lines 77 - 82) runs forever, and the coroutine does not return anything.

```
                promise_type::promise_type
                promise_type::get_return_object
            Generator::Generator
                promise_type::initial_suspend
            Generator::getNextValue
        getNext: start
        getNext: before co_yield
                promise_type::yield_value
        getNext: after co_yield
        getNext: before co_yield
                promise_type::yield_value
        getNext: after co_yield
        getNext: before co_yield
                promise_type::yield_value
        getNext: after co_yield
        getNext: before co_yield
                promise_type::yield_value
        getNext: after co_yield
        getNext: before co_yield
                promise_type::yield_value
        getNext: after co_yield
        getNext: before co_yield
                promise_type::yield_value
        getNext: after co_yield
```

`yield_value` Never Suspends

It is straightforward to restructure the generator `infiniteDataStreamComments.cpp` so that it produces a finite number of values.

6.7.2 Generalization

You may wonder why I never used the full generic potential of `Generator`. Let me adjust its implementation to produce the successive elements of an arbitrary container of the Standard Template Library.

Generator successively returning each element

```cpp
// coroutineGetElements.cpp

#include <coroutine>
#include <memory>
#include <iostream>
#include <string>
#include <vector>

template<typename T>
struct Generator {

    struct promise_type;
    using handle_type = std::coroutine_handle<promise_type>;

    Generator(handle_type h): coro(h) {}

    handle_type coro;

    ~Generator() {
        if ( coro ) coro.destroy();
    }
    Generator(const Generator&) = delete;
    Generator& operator = (const Generator&) = delete;
    Generator(Generator&& oth): coro(oth.coro) {
        oth.coro = nullptr;
    }
    Generator& operator = (Generator&& oth) {
        coro = oth.coro;
        oth.coro = nullptr;
        return *this;
    }
    T getNextValue() {
        coro.resume();
        return coro.promise().current_value;
    }
    struct promise_type {
        promise_type() {}

        ~promise_type() {}

        std::suspend_always initial_suspend() {
            return {};
        }
        std::suspend_always final_suspend() noexcept {
```

```
45                     return {};
46                 }
47             auto get_return_object() {
48                     return Generator{handle_type::from_promise(*this)};
49             }
50
51             std::suspend_always yield_value(const T value) {
52                 current_value = value;
53                     return {};
54             }
55              void return_void() {}
56             void unhandled_exception() {
57                 std::exit(1);
58             }
59
60             T current_value;
61         };
62
63     };
64
65     template <typename Cont>
66     Generator<typename Cont::value_type> getNext(Cont cont) {
67         for (auto c: cont) co_yield c;
68     }
69
70     int main() {
71
72         std::cout << '\n';
73
74         std::string helloWorld = "Hello world";
75         auto gen = getNext(helloWorld);
76         for (int i = 0; i < helloWorld.size(); ++i) {
77             std::cout << gen.getNextValue() << " ";
78         }
79
80         std::cout << "\n\n";
81
82         auto gen2 = getNext(helloWorld);
83         for (int i = 0; i < 5 ; ++i) {
84             std::cout << gen2.getNextValue() << " ";
85         }
86
87         std::cout << "\n\n";
88
89         std::vector myVec{1, 2, 3, 4 ,5};
```

```
90        auto gen3 = getNext(myVec);
91        for (int i = 0; i < myVec.size() ; ++i) {
92            std::cout << gen3.getNextValue() << " ";
93        }
94
95        std::cout << '\n';
96
97   }
```

In this example, the generator is instantiated and used three times. In the first two cases, gen (line 76) and gen2 (line 83) are initialized with std::string helloWorld, while gen3 uses a std::vector<int> (line 91). The output of the program should not be surprising. Line 78 returns all characters of the string helloWorld successively, line 85 only the first five characters, and line 93 the elements of the std::vector<int>.

You can try out the program on the Compiler Explorer[24].

```
H e l l o    w o r l d

H e l l o

1 2 3 4 5
```

A generator successively returning each element

To make it short. The implementation of the Generator<T> is almost identical to the previous one. The crucial difference with the previous program is the coroutine getNext.

getNext

```
template <typename Cont>
Generator<typename Cont::value_type> getNext(Cont cont) {
    for (auto c: cont) co_yield c;
}
```

getNext is a function template that takes a container as an argument and iterates in a range-based for loop through all elements of the container. After each iteration, the function template pauses. The return type Generator<typename Cont::value_type> may look surprising to you. Cont::value_type is a dependent template parameter, for which the parser needs a hint. By default, the compiler assumes a non-type if it could be interpreted as a type or a non-type. For this reason, I have to put typename in front of Cont::value_type.

---

[24]https://godbolt.org/z/j9znva

6.8 Various Job Workflows

Before I modify the workflow from section `co_await`, I want to make the awaiter workflow more transparent.

6.8.1 The Transparent Awaiter Workflow

I added a few comments to the program `startJob.cpp`.

Starting a job on request (including comments)

```cpp
// startJobWithComments.cpp

#include <coroutine>
#include <iostream>

struct MySuspendAlways {
    bool await_ready() const noexcept {
        std::cout << "        MySuspendAlways::await_ready" << '\n';
        return false;
    }
    void await_suspend(std::coroutine_handle<>) const noexcept {
        std::cout << "        MySuspendAlways::await_suspend" << '\n';

    }
    void await_resume() const noexcept {
        std::cout << "        MySuspendAlways::await_resume" << '\n';
    }
};

struct MySuspendNever {
    bool await_ready() const noexcept {
        std::cout << "        MySuspendNever::await_ready" << '\n';
        return true;
    }
    void await_suspend(std::coroutine_handle<>) const noexcept {
        std::cout << "        MySuspendNever::await_suspend" << '\n';

    }
    void await_resume() const noexcept {
        std::cout << "        MySuspendNever::await_resume" << '\n';
    }
};

struct Job {
```

```cpp
35        struct promise_type;
36        using handle_type = std::coroutine_handle<promise_type>;
37        handle_type coro;
38        Job(handle_type h): coro(h){}
39        ~Job() {
40            if ( coro ) coro.destroy();
41        }
42        void start() {
43            coro.resume();
44        }
45
46
47        struct promise_type {
48            auto get_return_object() {
49                return Job{handle_type::from_promise(*this)};
50            }
51            MySuspendAlways initial_suspend() {
52                std::cout << "    Job prepared" << '\n';
53                return {};
54            }
55            MySuspendAlways final_suspend() noexcept {
56                std::cout << "    Job finished" << '\n';
57                return {};
58            }
59            void return_void() {}
60            void unhandled_exception() {}
61
62        };
63    };
64
65    Job prepareJob() {
66        co_await MySuspendNever();
67    }
68
69    int main() {
70
71        std::cout << "Before job" << '\n';
72
73        auto job = prepareJob();
74        job.start();
75
76        std::cout << "After job" << '\n';
77
78    }
```

First of all, I replaced the predefined Awaitables `std::suspend_always` and `std::suspend_never` with Awaitables `MySuspendAlways` (line 6) and `MySuspendNever` (line 20). I use them in lines 51, 55, and 66. The Awaitables mimic the behavior of the predefined Awaitables but additionally write a comment. Due to the use of `std::cout`, the member functions `await_ready`, `await_suspend`, and `await_resume` cannot be declared as `constexpr`.

The screenshot of the program execution shows the control flow nicely, which you can directly observe on the Compiler Explorer[25].

```
Before job
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
        MySuspendAlways::await_resume
        MySuspendNever::await_ready
        MySuspendNever::await_resume
    Job finished
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
After job
```

Starting a job on request (including comments)

The function `initial_suspend` (line 51) is executed at the beginning of the coroutine and the function `final_suspend` at its end (line 55). The call `prepareJob()` (line 73) triggers the creation of the coroutine object, and the function call `job.start()` its resumption and, hence, completion (line 74). Consequently, the members `await_ready`, `await_suspend`, and `await_resume` of `MySuspendAlways` are executed. When you don't resume the Awaitable such as the coroutine object returned by the member function `final_suspend`, the function `await_resume` is not processed. In contrast, the Awaitable's `MySuspendNever` function is immediately ready because `await_ready` returns `true` and, hence, does not suspend.

Thanks to the comments, you should have an elementary understanding of the awaiter workflow. Now, it's time to vary it.

6.8.2 Automatically Resuming the Awaiter

In the previous workflow, I explicitly started the job.

---

[25]https://godbolt.org/z/T5rcE4

Explicitly starting the job

```cpp
int main() {

    std::cout <<  "Before job" << '\n';

    auto job = prepareJob();
    job.start();

    std::cout <<  "After job" <<  '\n';

}
```

This explicit invoking of `job.start()` was necessary because `await_ready` in the Awaitable `MySuspendAlways` always returned `false`. Now let's assume that `await_ready` can return `true` or `false` and the job is not explicitly started. A short reminder: When `await_ready` returns `true`, the function `await_resume` is directly invoked but not `await_suspend`.

Automatically Resuming the Awaiter

```cpp
 1  // startJobWithAutomaticResumption.cpp
 2
 3  #include <coroutine>
 4  #include <functional>
 5  #include <iostream>
 6  #include <random>
 7
 8  std::random_device seed;
 9  auto gen = std::bind_front(std::uniform_int_distribution<>(0,1),
10                             std::default_random_engine(seed()));
11
12  struct MySuspendAlways {
13      bool await_ready() const noexcept {
14          std::cout << "        MySuspendAlways::await_ready"  << '\n';
15          return gen();
16      }
17      bool await_suspend(std::coroutine_handle<> handle) const noexcept {
18          std::cout << "        MySuspendAlways::await_suspend"  << '\n';
19          handle.resume();
20          return true;
21
22      }
23      void await_resume() const noexcept {
24          std::cout << "        MySuspendAlways::await_resume"  << '\n';
25      }
```

```cpp
26    };
27
28    struct Job {
29        struct promise_type;
30        using handle_type = std::coroutine_handle<promise_type>;
31        handle_type coro;
32        Job(handle_type h): coro(h){}
33        ~Job() {
34            if ( coro ) coro.destroy();
35        }
36
37        struct promise_type {
38            auto get_return_object() {
39                return Job{handle_type::from_promise(*this)};
40            }
41            MySuspendAlways initial_suspend() {
42                std::cout << "    Job prepared" << '\n';
43                return {};
44            }
45            std::suspend_always final_suspend() noexcept {
46                std::cout << "    Job finished" << '\n';
47                return {};
48            }
49            void return_void() {}
50            void unhandled_exception() {}
51
52        };
53    };
54
55    Job performJob() {
56        co_await std::suspend_never();
57    }
58
59    int main() {
60
61        std::cout << "Before jobs" << '\n';
62
63        performJob();
64        performJob();
65        performJob();
66        performJob();
67
68        std::cout << "After jobs" << '\n';
69
70    }
```

First of all, the coroutine is now called `performJob` and runs automatically. `gen` (line 9) is a random number generator for the numbers 0 or 1. It uses for its job the default random engine, initialized with the seed. Thanks to `std::bind_front`[26], I can bind it together with the `std::uniform_int_distribution` to get a callable which, when used, gives me a random number 0 or 1.

I removed in this example the Awaitables with predefined Awaitables from the C++ standard, except the Awaitable `MySuspendAlways` as the return type of the member function `initial_suspend` (line 41). `await_ready` (line 13) returns a boolean. When the boolean is `true`, the control flow jumps directly to the member function `await_resume` (line 23), when `false`, the coroutine is immediately suspended and, therefore, the function `await_suspend` runs (line 17). The function `await_suspend` gets the handle to the coroutine and uses it to resume the coroutine (line 19). Instead of returning the value `true`, `await_suspend` can also return `void`.

The following screenshot shows: When `await_ready` returns `true`, the function `await_resume` is called, when `await_ready` returns `false`, the function `await_suspend` is also called.

You can try out the program on the Compiler Explorer[27].

---

[26]https://en.cppreference.com/w/cpp/utility/functional/bind_front
[27]https://godbolt.org/z/8b1Y14

```
Before jobs
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
After jobs
```

Automatically Resuming the Awaiter

Let me improve the presented program more and resume the awaiter on a separate thread.

6.8.3 Automatically Resuming the Awaiter on a Separate Thread

The following program is based on the previous one.

Automatically Resuming the Awaiter on a Seperate Thread

```cpp
1  // startJobWithAutomaticResumptionOnThread.cpp
2
3  #include <coroutine>
4  #include <functional>
5  #include <iostream>
6  #include <random>
7  #include <thread>
8  #include <vector>
9
10 std::random_device seed;
11 auto gen = std::bind_front(std::uniform_int_distribution<>(0,1),
12                            std::default_random_engine(seed()));
```

```
13
14    struct MyAwaitable {
15        std::jthread& outerThread;
16        bool await_ready() const noexcept {
17            auto res = gen();
18            if (res) std::cout << " (executed)" << '\n';
19            else std::cout << " (suspended)" << '\n';
20            return res;
21        }
22        void await_suspend(std::coroutine_handle<> h) {
23            outerThread = std::jthread([h] { h.resume(); });
24        }
25        void await_resume() {}
26    };
27
28
29    struct Job{
30        static inline int JobCounter{1};
31        Job() {
32            ++JobCounter;
33        }
34
35        struct promise_type {
36            int JobNumber{JobCounter};
37            Job get_return_object() { return {}; }
38            std::suspend_never initial_suspend() {
39                std::cout << "    Job " << JobNumber << " prepared on thread "
40                          << std::this_thread::get_id();
41                return {};
42            }
43            std::suspend_never final_suspend() noexcept {
44                std::cout << "    Job " << JobNumber << " finished on thread "
45                          << std::this_thread::get_id() << '\n';
46                return {};
47            }
48            void return_void() {}
49            void unhandled_exception() { }
50        };
51    };
52
53    Job performJob(std::jthread& out) {
54        co_await MyAwaitable{out};
55    }
56
57    int main() {
```

```
58
59      std::vector<std::jthread> threads(8);
60      for (auto& thr: threads) performJob(thr);
61
62  }
```

The main difference with the previous program is the new awaitable `MyAwaitable`, used in the coroutine `performJob` (line 54). On the contrary, the coroutine object returned from the coroutine `performJob` is straightforward. Essentially, its member functions `initial_suspend` (line 38) and `final_suspend` (line 43) return the predefined awaitable `std::suspend_never`. Additionally, both functions show the `JobNumber` of the executed job and the thread ID on which it runs. The screenshot shows which coroutine runs immediately and which one is suspended. Thanks to the thread id, you can observe that suspended coroutines are resumed on a different thread.

You can try out the program on the Wandbox[28].


```
Job 1 prepared on thread 140434982274944 (executed)
Job 1 finished on thread 140434982274944
Job 2 prepared on thread 140434982274944 (suspended)
Job 3 prepared on thread 140434982274944 (suspended)
Job 4 prepared on thread 140434982274944 (suspended)
Job 2 finished on thread 140434877310720
Job 5 prepared on thread 140434982274944 (executed)
Job 5 finished on thread 140434982274944
Job 6 prepared on thread 140434982274944 (suspended)
Job 7 prepared on thread 140434982274944 (suspended)
Job 3 finished on thread 140434868918016
Job 8 prepared on thread 140434982274944 (executed)
Job 8 finished on thread 140434982274944
Job 4 finished on thread 140434860525312
Job 6 finished on thread 140434852132608
Job 7 finished on thread 140434843739904
```

Automatically Resuming the Awaiter on a Separate Thread

Let me discuss the interesting control flow of the program. Line 59 creates eight default-constructed threads, which the coroutine `performJob` (line 53) takes by reference. Further, the reference becomes the argument for creating `MyAwaitable{out}` (line 54). Depending on the value of `res` (line 17), and, therefore, the return value of the function `await_ready`, the Awaitable continues (`res` is `true`) to run or is suspended (`res` is `false`). In case `MyAwaitable` is suspended, the function `await_suspend` (line 22) is executed. Thanks to the assignment of `outerThread` (line 23), it becomes a running thread. The running threads must outlive the lifetime of the coroutine. For this reason, the threads have the scope of the `main` function.

---

[28]https://wandbox.org/permlink/skHgWKF0SYAwp8Dm

Distilled Information

- Calculating the sum of a vector can be done in various ways. You can do it sequentially or concurrently with maximum and minimum sharing of data. The performance numbers differ drastically.

- Thread-safe initialization of a singleton is the classical use-case for thread-safe initialization of a shared variable. There are many ways to do it, with varying performance characteristics.

- I start with a small program and successively improve it by weakening the memory ordering. I verify each step of my process of ongoing optimization with CppMem. CppMem is an interactive tool for exploring the behavior of small code snippets using the C++ memory model.

- There are many ways in C++20 to synchronize threads. You can use condition variables, `std::atomic_flag`, `std::atomic<bool>`, or semaphores. I discuss the performance numbers of various ping-pong games.

- Thanks to the new keyword `co_return`, I can implement in the section variations of futures an eager future, a lazy future, or a future running in a separate thread. Heavily used comments make its workflow transparent.

- `co_yield` enables it to create infinite data streams. In the case study modification and generalization of a generator, the infinite data streams become finite and generic.

- The case study of various job workflows presents a few coroutines that are automatically resumed if necessary. `co_await` makes this possible.

7. The Future: C++23

Cippi predicts the future

This chapter is about the future of C++: C++23. In this chapter, my intent is not to be as precise as the other chapters in this book. That's for two reasons. First, not all of the presented features make it into the C++23 standard. Second, if a feature makes it into the C++23 standard, the interface of that feature changes very likely. This holds in particular true for the executors. I update this book regularly and, therefore, reflect the revised and new proposals in this chapter.

This chapter's goal is quite simple: to give you an idea about the upcoming concurrency features in C++23.
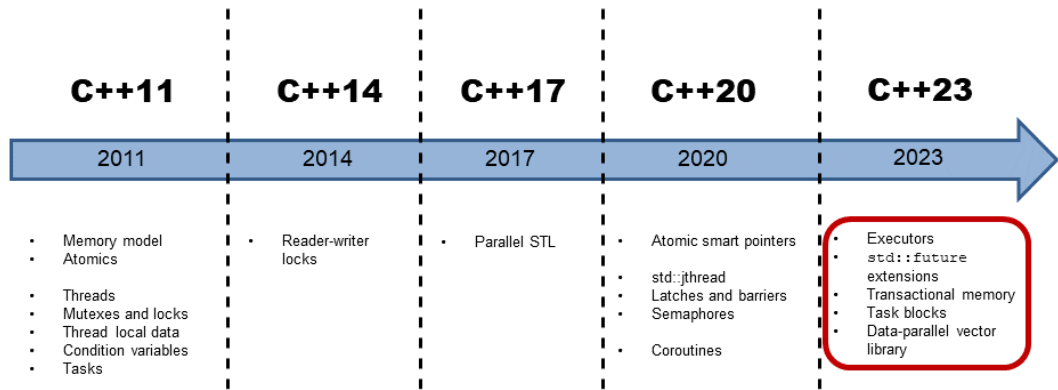
Concurrency in C++23

7.1 Executors

Executors are the basic building block for execution in C++ and fulfill a similar role for execution, such as allocators for the containers in C++. Functions such as async, the parallel algorithms of the Standard Template Library, the then continuation of futures, the run member functions of task blocks, or the post, dispatch, or defer calls of the Networking TS[1] use them. Also, execution is a fundamental concern of programming. There is no standardized way to perform an execution.

Here is the introductory example of the proposal P0761[2].

Various parallel_for implementations

```cpp
void parallel_for(int facility, int n, function<void(int)> f) {
  if(facility == OPENMP) {
    #pragma omp parallel for
    for(int i = 0; i < n; ++i) {
      f(i);
    }
  }
else if(facility == GPU) {
  parallel_for_gpu_kernel<<<n>>>(f);
}
else if(facility == THREAD_POOL) {
  global_thread_pool_variable.submit(n, f);
  }
}
```

This parallel_for function has a few issues.

- A simple function such as parallel_for is **complex** to maintain. The complexity gets worse and worse if new algorithms or new parallel paradigms should be supported.

- Each branch of the function has different **synchronization** properties. OpenMP[3] may block until all spawning threads are done, GPU runs typically asynchronously, and a thread pool may block or not. Insufficient synchronization may end in a data race or deadlock. In the best case, you get a race condition.

- The parallel_for loop is too **restrictive**. For example, there is no way to use your thread pool instead of the global thread pool in the function: global_thread_pool_variable.submit(n, f);.

---

[1]https://en.cppreference.com/w/cpp/experimental
[2]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0761r2.pdf
[3]https://en.wikipedia.org/wiki/OpenMP

7.1.1 A long Way

In October 2018 many proposals were written for executors, and many design decisions are still open. The expectation is that they are part of C++23. There is the chance the one-way execution is even standardized with C++20. This chapter is mainly based on the proposals to the design of executors P0761[4], and to their formal description in P0443[5] and P1244[6]. P0443 (A Unified Executors Proposal for C++) proposes the one-way execution, which may be part of C++20, and P1244 (Dependent Execution for a Unified Executors Proposal for C++) propose the dependent execution, which may be part of C++23. This chapter also refers to the relatively new "Modest Executor Proposal" P1055[7].

7.1.2 What is an Executor?

First of all. What is an executor? An executor consists of a set of rules about **where**, **when** and **how** to run a callable unit.

- **Where:** The callable may run on an internal or external processor, and that the result is read back from the internal or external processor.
- **When:** The callable may run immediately or be scheduled for a later time.
- **How:** The callable may run on a CPU or GPU or even be executed in a vectorized way.

More formally, each executor has properties associated with the performed execution function.

7.1.2.1 Executor Properties

You can associate these properties with an executor in two ways: `execution::require`, or `execution::prefer`.

1. **Directionality:** The execution function can be of kind "fire and forget" (`execution::oneway`), return a future (`execution::twoway`), or return a continuation (`execution::then`).
2. **Cardinality:** The execution function can create one (`execution::single`) or multiple execution agents (`execution::bulk`).
3. **Blocking:** The function may block or not. There are three mutually-exclusive blocking properties: `execution::blocking.never`, `execution::blocking.possibly`, and `execution::blocking.always`.
4. **Continuations:** The task may be performed on the client's calling thread (`execution::continuation`) or not (`execution::not_continuation`).
5. **Future task submission:** Specify if outstanding work is likely (`exection::outstanding_-work.tracked`) or not (`execution::outstanding_work.untracked`):
6. **Bulk forward progress guarantees:** Specifies the the mutually-exclusive forward progress guarantees of execution agents created in bulk: `execution::bulk_sequenced_execution`, `execution::bulk_-parallel_execution`, and `execution::bulk_unsequenced_execution`.

---

[4] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0761r2.pdf
[5] http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0443r7.html
[6] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1244r0.html
[7] http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1055r0.pdf

7. **Thread execution mapping guarantees**: Should each execution agent be mapped to a new thread (`execution::new_thread_execution_mapping`) or not (`execution::thread_execution_mapping`).

8. **Allocators**: Associates an allocator (`execution::allocator`) with an executor.

You can even define your properties.

Executors are the Building Blocks

Because the executors are the building blocks for execution, the concurrency and parallelism features of C++ heavily depend on them. This holds for the extended futures, the extensions for networking N4734[8], but also the parallel algorithms of the STL, and the new concurrency features in C++20/23 such as latches and barriers, coroutines, transactional memory, and task blocks.

7.1.3 First Examples

7.1.3.1 Using an Executor

Here are a few code snippets showing the usage of executors:

7.1.3.1.1 The promise `std::async`

Executing an `std::async`

```cpp
// get an executor through some means
my_executor_type my_executor = ...

// launch an async using my executor
auto future = std::async(my_executor, [] {
    std::cout << "Hello world, from a new execution agent!" << '\n';
});
```

7.1.3.1.2 The STL Algorithm `std::for_each`

---

[8]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4734.pdf

Performing `std::for_each` in parallel on `my_executor`

```
// get an executor through some means
my_executor_type my_executor = ...

// execute a parallel for_each "on" my executor
std::for_each(std::execution::par.on(my_executor),
              data.begin(), data.end(), func);
```

7.1.3.1.3 Networking TS: Accepting a Client Connection with the Default System Executor

Using the system executor to accept new connections

```
// obtain an acceptor (a listening socket) through some means
tcp::acceptor my_acceptor = ...

// perform an asynchronous operation to accept a new connection
acceptor.async_accept(
    [](std::error_code ec, tcp::socket new_connection)
    {
      ...
    }
  );
```

7.1.3.1.4 Networking TS: Accepting a Client Connection with a Thread Pool Executor

Using a thread pool executor to accept new connections

```
// obtain an acceptor (a listening socket) through some means
tcp::acceptor my_acceptor = ...

// obtain an executor for a specific thread pool
auto my_thread_pool_executor = ...

// perform an asynchronous operation to accept a new connection
acceptor.async_accept(
    std::experimental::net::bind_executor(my_thread_pool_executor,
      [](std::error_code ec, tcp::socket new_connection)
      {
        ...
      }
    )
  );
```

The function `std::experimental::net::bind_executor` from the networking TS N4734[9] allows it to use a specific executor. In this case, the completion handler runs on a thread pool and executes the lambda function.

To use an executor, you have to obtain it.

7.1.3.2 Obtaining an Executor

There are various ways to obtain an executor.

7.1.3.2.1 From the Execution Context `static_thread_pool`

An executor from an execution context

```cpp
// create a thread pool with 4 threads
static_thread_pool pool(4);

// get an executor from the thread pool
auto exec = pool.executor();

// use the executor on some long-running task
auto task1 = long_running_task(exec);
```

7.1.3.2.2 From the Execution Policy `std::execution::par`

An executor from an execution policy

```cpp
// get par's associated executor
auto par_exec = std::execution::par.executor();

// use the executor on some long-running task
auto task2 = long_running_task(par_exec);
```

7.1.3.2.3 From the System Executor

This is the default executor that usually uses a thread for the execution. It is used if another one is not specified.

7.1.3.2.4 From an Executor Adapter

---

[9]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4734.pdf

An executor from an executor adaptor

```cpp
// get an executor from a thread pool
auto exec = pool.executor();

// wrap the thread pool's executor in a logging_executor
logging_executor<decltype(exec)> logging_exec(exec);

// use the logging executor in a parallel sort
std::sort(std::execution::par.on(logging_exec), my_data.begin(), my_data.end());
```

logging_executor is in the code snippet a wrapper for the pool executor.

7.1.4 Goals of an Executor Concept

What are the goals of an executor concept according to the proposal P1055[10]?

1. **Batchable**: control the trade-off between the cost of the callable transition and its size.

2. **Heterogenous**: allow the callable to run on heterogeneous contexts and get the result back.

3. **Orderable**: specify the order in which the callables are invoked. The goal includes ordering guarantees such as LIFO[11] (Last In, First Out), FIFO[12] (First In, First Out) execution, priority or time constraints, or even sequential execution.

4. **Controllable**: the callable has to be targetable to a specific computing resource, deferred, or even canceled.

5. **Continuable**: to control asynchronous callable signals are needed. These signals have to indicate whether the result is available, whether an error occurred, when the callable is done or if the callee wants to cancel the callable. The explicit starting of the callable or the stopping of the starting callable should also be possible.

6. **Layerable**: hierarchies allow capabilities to be added without increasing the simpler use-cases' complexity.

7. **Usable**: ease of use for the implementer and the user should be the primary goal.

8. **Composable**: allows a user to extend the executors for features that are not part of the standard.

9. **Minimal**: nothing should exist on the executor concepts that could be added externally in a library on top of the concept.

---

[10]http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1055r0.pdf
[11]https://en.wikipedia.org/wiki/Stack_(abstract_data_type)
[12]https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)

7.1.5 Terminology

The Proposal P0761[13] defines a few new terms for the execution of a callable:

- **Execution resource**: is an instance of hardware or software capability capable of executing a callable. An execution unit can range from a SIMD vector unit to an entire runtime managing a large collection of threads. Execution resources such as a CPU or a GPU are heterogeneous because they have different freedoms and restrictions.

- **Execution context**: a program object representing a specific collection of execution resources and the execution agents within those resources. Typical examples are a thread pool or a distributed or a heterogeneous runtime.

- **Execution agent**: is a unit of execution of a specific execution context that is mapped to a single invocation of a callable on an execution resource. Typical examples are a CPU thread or a GPU execution unit.

- **Executor**: is an object associated with a specific execution context. It provides one or more execution functions for creating execution agents from a callable function object.

7.1.6 Execution Functions

According to the previous paragraph, an executor provides one or more execution function for creating execution agents from a callable. An executor has to support at least one of the six following functions.

The execution functions of an executor

Name	Cardinality	Direction
execute	single	oneway
twoway_execute	single	twoway
then_execute	single	then
bulk_execute	bulk	oneway
bulk_twoway_execute	bulk	twoway
bulk_then_execute	bulk	then

Each execution function has two properties: cardinality and direction.

- Cardinality
  - `single`: creates one execution agents
  - `bulk`: creates a group of execution agents
- Direction
  - `oneway`: creates an execution agent and does not return a result

---

[13]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0761r2.pdf

    – `twoway`: creates an execution agent and does return a future that can be used to wait for execution to complete

    – `then`: creates an execution agent and does return a future that can be used to wait for execution to complete. The execution agent begins execution after a given future `pred` becomes ready.

Let me explain the execution functions more informally.

All of them take a callable.

7.1.6.1 Single Cardinality

The single cardinality is straightforward. A `oneway` execution function is a fire-and-forget job and returns `void`. It's pretty similar to a [fire and forget future](#), but it does not automatically block in the future's destructor. A `twoway` execution function returns you a future that you can use to pick up the result. This behaves similarly to a `std::promise` that gives you back the handle to the associated `std::future`. In the `then` case, it is a kind of continuation. It gives you back a future, but the execution agent runs only if the provided future `pred` is `ready`.

7.1.6.2 Bulk Cardinality

The bulk cardinality case is more complicated. These functions create a group of execution agents, and each of these execution agents calls the given callable `f`. They return the result of a result factory and not the result of a single callable `f` invoked by the execution agents. The first parameter of `f` is the shape parameter, which is an integral type and stands for the index of the agent's type. Further arguments are the result factory if it is a `twoway` executor and a shape factory, which all agents share. The lifetime of the shared parameter created by the shared factory is bound to the agents' lifetime. Both are called factories because they produce their value by executing a callable and run before the agents. The client is responsible for disambiguating the correct result via this result factory.

When the function `bulk_then_execute` is used, the callable `f` takes its predecessor future as an additional argument. The callable `f` takes the result, the shared parameter, and the predecessor by reference because no agent is an owner.

7.1.6.3 `execution::require`

How can you be sure that your executor supports the specific execution function?

In the special case, you know it.

Use an oneway single executor

```cpp
void concrete_context(const my_oneway_single_executor& ex)
{
  auto task = ...;
  ex.execute(task);
}
```

In the general case, you can use the function `execution::require` to ask for it.

Ask for an twoway_execute single executor

```cpp
template<class Executor>
void generic_context(const Executor& ex)
{
  auto task = ...;

  // ensure .toway_execute() is available with execution::require()
  execution::require(ex, execution::single, execution::twoway).toway_execute(task);
}
```

7.1.7 A Prototype Implementation

Based on the proposal P0443R5[14], a prototype implementation of the executor proposal is available. This prototype implementation helped me a lot to get a deeper understanding of the bulk cardinality in particular.

A prototype implementation of the executors

```cpp
1   // executor.cpp
2
3   #include <atomic>
4   #include <experimental/thread_pool>
5   #include <iostream>
6   #include <utility>
7
8   namespace execution = std::experimental::execution;
9   using std::experimental::static_thread_pool;
10  using std::experimental::executors_v1::future;
11
12  int main(){
13
14      static_thread_pool pool{4};
```

_____

[14]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0443r5.html

```cpp
15          auto ex = pool.executor();
16
17          // One way, single.
18          ex.execute([]{ std::cout << "We made it!" << '\n'; });
19
20          std::cout << '\n';
21
22          // Two way, single.
23          future<int> f1 = ex.twoway_execute([]{ return 42; });
24          f1.wait();
25          std::cout << "The result is: " << f1.get() << '\n';
26
27          std::cout << '\n';
28
29          // One way, bulk.
30          ex.bulk_execute([](int n, int& sha){
31              std::cout << "part " << n << ": " << "shared: " << sha <<  "\n";
32              }, 8,
33              []{ return 0; }
34          );
35
36          std::cout << '\n';
37
38          // Two way, bulk, void result.
39          future<void> f2 = ex.bulk_twoway_execute(
40              [](int n, std::atomic<short>& m){
41                  std::cout << "async part " << n ;
42                  std::cout << " atom: " << m++ << '\n';
43              }, 8,
44              []{},
45              []{
46                  std::atomic<short> atom(0);
47                  return std::ref(atom);
48              }
49          );
50          f2.wait();
51          std::cout << "bulk result available" << '\n';
52
53          std::cout << '\n';
54
55          // Two way, bulk, non-void result.
56          future<double> f3 = ex.bulk_twoway_execute(
57              [](int n, double&, int &){
58                  std::cout << "async part " << n << " ";
59                  std::cout << std::this_thread::get_id() << '\n';
```
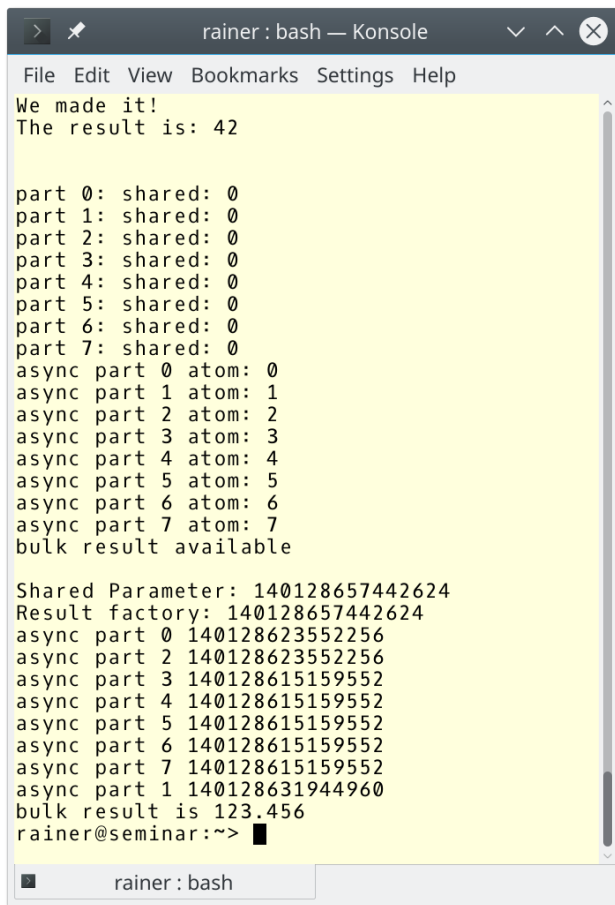
```
60            }, 8,
61            []{
62                std::cout << "Result factory: "
63                              << std::this_thread::get_id() << '\n';
64                return 123.456; },
65            []{
66                std::cout << "Shared Parameter: "
67                              << std::this_thread::get_id() << '\n';
68                return 0; }
69        );
70        f3.wait();
71        std::cout << "bulk result is " << f3.get() << '\n';
72
73   }
```

The program uses as executor a thread pool of four threads (lines 14 and 15). Lines 18 and 23 uses execution functions of single cardinality and create two agents of single cardinality. The second one is a `twoway` execution function and returns, therefore, a result.

The remaining execution functions in lines 30, 39, and 56 are of bulk cardinality. Each function creates eight agents (lines 32, 43, and 60). In the first case, the callable displays the index `n` and the shared value `sha` which the shared factory in line 33 creates. The next execution function `bulk_twoway_execute` is more interesting. Although its result factory returns `void`, the shared state is the atomic variable `atom`. Each agent increments its value by one (line 42). Thanks to the result factory, the last execution function (lines 56 to 69) returns the result `123.456`. It's quite interesting to see how many threads are involved in the execution of the callable and the execution of the result and the shared factory. The program's output shows that result and shared factory run in the same thread but the agents in a different thread.

A prototype implementation of executors

7.2 Extended Futures

Tasks in the form of promises and futures have an ambivalent reputation in C++11. On the one hand, they are a lot easier to use than threads or condition variables; on the other hand, they have a significant deficiency. They cannot be composed. C++20/23 overcomes this deficiency.

I have written about tasks in the form of `std::async`, `std::packaged_task`, or `std::promise` and `std::future`. The details are here: tasks. With C++20/23 we may get extended futures.

7.2.1 Concurrency TS v1

7.2.1.1 `std::future`

The name extended futures is quite easy to explain. First, the C++11 `std::future` interface was extended; second, there are new functions for creating special futures that are composable. I start with my first point.

The extended future has three new member functions:

- The unwrapping constructor that unwraps the outer future of a wrapped future (`future<future<T>>`).
- The predicate `is_ready` that returns if a shared state is available.
- The member function `then` attaches a continuation to a future.

At first, the state of a future can be `valid` or `ready`.

7.2.1.1.1 valid versus ready

- **valid**: a future is valid if it has a shared state (with a promise). This does not have to be the case because you can default-construct a `std::future` without a promise.
- **ready**: a future is ready if the shared state is available or put differently if the promise has already produced its value.

Therefore, (`valid == true`) is a requirement for (`ready == true`).

My mental model of promise and future is that they are the endpoints of a data channel.

Promise: sender   **Future: receiver**

set   get
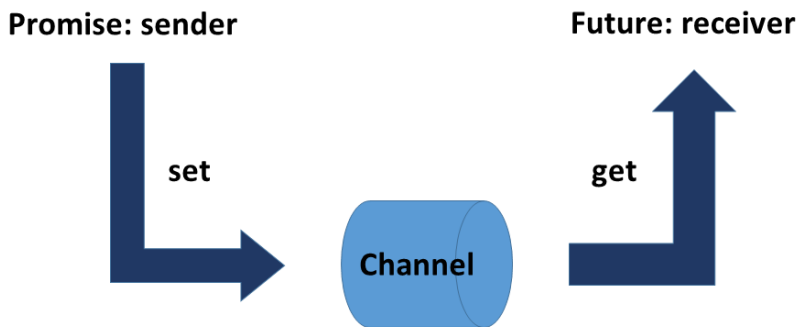
Channel

Tasks as data channels between communication endpoints

Now the difference between valid and ready becomes quite natural. The future is valid if there is a data channel to a promise. The future is ready if the promise has already put its value into the data channel.

Now to the member function `then` for the continuation of a future.

7.2.1.1.2 Continuations with `then`

`then` empowers you to attach a future to another future. It often happens that a future is packed into another future. The job of the unwrapping constructor is it to unwrap the outer future.

The proposal N3721

Before I show the first code snippet, I have to say a few words about proposal N3721[15]. Most of this section is from the proposal on "Improvements for std::future<T> and Related APIs". This includes my examples. Strangely, the original authors frequently did not use the final `get` call to get the result from the future. Therefore, I added the `res.get` call to the examples and saved the result in a variable `myResult`. Additionally, I fixed a few typos.

Continuations with `std::future`

```
1   #include <future>
2   using namespace std;
3   int main() {
4
5     future<int> f1 = async([]() { return 123; });
6     future<string> f2 = f1.then([](future<int> f) {
7       return to_string(f.get());      // here .get() won't block
8     });
9
10    auto myResult= f2.get();
11
12  }
```

There is a subtle difference between the `to_string(f.get())` call (line 7) and the `f2.get()` call in line 10. As I already mentioned in the code snippet: the first call is non-blocking, and the second call is blocking. The `f2.get()` call waits until the result of the future-chain is available. This statement also holds for chains such as `f1.then(...).then(...).then(...).then(...)` as it holds for the composition of extended futures. The final `f2.get()` call is blocking.

7.2.1.2 `std::async, std::packaged_task,` and `std::promise`

There is not much to say about the extensions of `std::async`, `std::package_task`, and `std::promise`. I only have to add that in C++20/23 all three return extended futures.

The composition of futures is more exciting. Now we can compose asynchronous tasks.

---

[15]https://isocpp.org/files/papers/N3721.pdf

7.2.1.3 Creating new Futures

C++20 gets four new functions for creating special futures. These functions are `std::make_ready_-future`, `std::make_execptional_future`, `std::when_all`, and `std::when_any`. First, let's look at the functions `std::make_ready_future`, and `std::make_exceptional_future`.

7.2.1.3.1 `std::make_ready_future` and `std::make_exceptional_future`

Both functions create an immediately ready future. In the first case, the future has a value; in the second case an exception. What seems to be strange at first actually makes much sense. In C++11 the creation of a ready future requires a promise. This is necessary even if the shared state is immediately available.

Creating a future with make_ready_future

```cpp
future<int> compute(int x) {
  if (x < 0) return make_ready_future<int>(-1);
  if (x == 0) return make_ready_future<int>(0);
  future<int> f1 = async([]() { return do_work(x); });
  return f1;
}
```

Hence the result must only be calculated by a promise if (x > 0) holds.

A short remark: both functions are the pendant to the return function in a monad. Now let's begin with future composition.

7.2.1.3.2 `std::when_any` and `std::when_all`

Both functions have a lot in common.

First, let's look at the input.

when_any and when_all

```cpp
template < class InputIt >
auto when_any(InputIt first, InputIt last)
    -> future<when_any_result<
                std::vector<typename std::iterator_traits<InputIt>::value_type>>>;

template < class... Futures >
auto when_any(Futures&&... futures)
    -> future<when_any_result<std::tuple<std::decay_t<Futures>...>>>;

template < class InputIt >
auto when_all(InputIt first, InputIt last)
    -> future<std::vector<typename std::iterator_traits<InputIt>::value_type>>;
```

```
template < class... Futures >
auto when_all(Futures&&... futures)
    -> future<std::tuple<std::decay_t<Futures>...>>;
```

Both functions accept a pair of iterators for a future range or an arbitrary number of futures. The big difference is that in the case of the pair of iterators, the futures have to be of the same type; while in the case of the arbitrary number of futures, the futures can have different types, and even `std::future` and `std::shared_future` can be used.

The function's output depends on whether a pair of iterators or an arbitrary number of futures (variadic template) was used. Both functions return a future. If a pair of iterators was used, you get a future of futures in a `std::vector`: `future<vector<future<R>>>`. If you use a variadic template, you get a future of futures in a `std::tuple`: `future<tuple<future<R0>, future<R1>, ... >>`.

This covers their commonalities. The future that both functions return is ready if all input futures (`when_all`), or if any of the input futures (`when_any`) are ready.

The following two examples show the usage of `std::when_all` and `std::when_any`.

7.2.1.3.3 `std::when_all`

Future composition with std::when_all

```
1   #include <future>
2
3   using namespace std;
4
5   int main() {
6
7     shared_future<int> shared_future1 = async([] { return intResult(125); });
8     future<string> future2 = async([]() { return stringResult("hi"); });
9
10    future<tuple<shared_future<int>, future<string>>> all_f =
11          when_all(shared_future1, future2);
12
13    future<int> result = all_f.then(
14          [](future<tuple<shared_future<int>, future<string>>> f){
15            return doWork(f.get());
16          });
17
18    auto myResult = result.get();
19
20  }
```

The future `all_f` (line 10) composes both the future `shared_future1` (line 7) and `future2` (line 8). The future `result` in line 13 is executed if all underlying futures are ready. In this case, the future `all_f` in line 15 is executed. The result is in the future `result` and can be used in line 18.

7.2.1.3.4 `std::when_any`

Future composition with std::when_any

```cpp
#include <future>
#include <vector>

using namespace std;

int main(){

  vector<future<int>> v{ .... };
  auto future_any = when_any(v.begin(), v.end());

  when_any_result<vector<future<int>>> result = future_any.get();

  future<int>& ready_future = result.futures[result.index];

  auto myResult = ready_future.get();

}
```

The future in `when_any` can be taken by `result` in line 11. `result` provides the information indicating which input future is ready. If you don't use `when_any_result`, you have to ask each future if ready. That is tedious.

`future_any` is the future that is ready if one of its input futures is ready. `future_any.get()` in line 11 returns the future `result`. By using `result.futures[result.index]` (line 13) you have the `ready_future` and thanks to `ready_future.get()` you can ask for the result of the job.

Either the already standardized futures or the concurrency TS v1 futures[16] are "not as generic, expressive or powerful as they should be" P0701r1[17]. Additionally, the executors as basic building blocks for executing something have to be unified with the new futures.

7.2.2 Unified Futures

What are the disadvantages of the already standardized and the futures from the concurrency TS 1?

---

[16]http://en.cppreference.com/w/cpp/experimental/concurrency
[17]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0701r1.html

7.2.2.1 Disadvantages

The already mentioned document gives an excellent description of the deficiencies of the futures.

7.2.2.1.1 `future`/`promise` Should Not Be Coupled to `std::thread` Execution Agents

C++11 had only one executor: `std::thread`. Consequently, futures and `std::thread` were inseparable. This changed with C++17 and the parallel algorithms of the STL. This changes even more with the new executors, which you can use to configure the future. For example, the future may run in a separate thread, in a thread pool, or sequentially.

7.2.2.1.2 Where are `.then` Continuations are Invoked?

Imagine you have a simple continuation, such as in the following example.

Continuations with `std::future`

```
future<int> f1 = async([]() { return 123; });
future<string> f2 = f1.then([](future<int> f) {
    return to_string(f.get());
});
```

The question is: Where should the continuation run? There are a few possibilities today:

1. Consumer Side: The consumer execution agent always executes the continuation.

2. Producer Side: The producer execution agent always executes the continuation.

3. inline_executor semantics: If the shared state is ready when the continuation is set, the consumer thread executes the continuation. If the shared state is not ready when the continuation is set, the producer thread executes the continuation.

4. thread_executor semantics: A new `std::thread` executes the continuation.

In particular, the first two possibilities have a significant drawback: they block. In the first case, the consumer blocks until the producer is ready. In the second case, the producer blocks, until the consumer is ready.

Here are a few nice use-cases of executor propagation from the document P0701r1[18]:

---

[18]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0701r1.html

Executor propagation

```cpp
auto i = std::async(thread_pool, f).then(g).then(h);
// f, g and h are executed on thread_pool.

auto i = std::async(thread_pool, f).then(g, gpu).then(h);
// f is executed on thread_pool, g and h are executed on gpu.

auto i = std::async(inline_executor, f).then(g).then(h);
// h(g(f())) are invoked in the calling execution agent.
```

7.2.2.1.3 Passing futures to `.then` Continuations is Unwieldy

Because the future is passed to the continuation and not its value, the syntax is quite complicated. First, once more, the correct but verbose version.

Continuations with `std::future`

```cpp
std::future<int> f1 = std::async([]() { return 123; });
std::future<std::string> f2 = f1.then([](std::future<int> f) {
    return std::to_string(f.get());
});
```

Now, I assume that I can pass the value because `to_string` is overloaded on `std::future<int>`.

Continuations with `std::future passing the value`

```cpp
std::future<int> f1 = std::async([]() { return 123; });
std::future<std::string> f2 = f1.then(std::to_string);
```

7.2.2.1.4 `when_all` and `when_any` Return Types are Unwieldy

The chapter to `std::when_all` and `std::when_any` shows their quite complicated usage.

7.2.2.1.5 Conditional Blocking in futures Destructor Must Go

Fire and forget futures look very promising but have a big drawback. A future that is created by `std::async` waits on its destructor until its promise is done. What seems to be concurrent runs sequentially. According to document P0701r1, this is not acceptable and error-prone.

I describe the peculiar behavior of Fire and Forget Futures in the referenced chapter.

7.2.2.1.6 Immediate Values and future Values Should Be Easy to Composable

In C++11, there is no convenient way to create a future. We have to start with a promise.

Creating a future in the current standard

```
std::promise<std::string> p;
std::future<std::string> fut = p.get_future();
p.set_value("hello");
```

This may change with the function `std::make_ready_future` concurreny TS v1.

Creating a future in the concurrency TS v1

```
std::future<std::string> fut = make_ready_future("hello");
```

Using future and non-future arguments would make our job even more comfortable.

Using future and non-future arguments

```cpp
bool f(std::string, double, int);


std::future<std::string> a = /* ... */;
std::future<int>    c = /* ... */;

std::future<bool> d1 = when_all(a, make_ready_future(3.14), c).then(f);
// f(a.get(), 3.14, c.get())

std::future<bool> d2 = when_all(a, 3.14, c).then(f);
// f(a.get(), 3.14, c.get())
```

Neither the syntactic form d1 nor the syntactic form d2 is possible with the concurrency ts v1.

7.2.2.2 Five New Concepts

There are five new concepts for futures and promises in proposal 1054R0[19].

- **FutureContinuation**, invocable objects that are called with the value or exception of a future as an argument.
- **SemiFuture**, which can be bound to an executor, an operation that produces a `ContinuableFuture` (`f = sf.via(exec)`).
- **ContinuableFuture**, which refines SemiFuture and instances can have one `FutureContinuation` attached to them (`f.then(c)`), which is executed on the future's associated executor when the future becomes ready.
- **SharedFuture**, which refines `ContinuableFuture` and instances can have multiple `FutureContinuations` attached to them.
- **Promise**, each associated with a future and makes the future ready with either a value or an exception.

The paper also provides the declaration of these new concepts.

---

[19]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1054r0.html

The five new concepts for futures and promises

```cpp
template <typename T>
struct FutureContinuation
{
  // At least one of these two overloads exists:
  auto operator()(T value);
  auto operator()(exception_arg_t, exception_ptr exception);
};

template <typename T>
struct SemiFuture
{
  template <typename Executor>
  ContinuableFuture<Executor, T> via(Executor&& exec) &&;
};

template <typename Executor, typename T>
struct ContinuableFuture
{
  template <typename RExecutor>
  ContinuableFuture<RExecutor, T> via(RExecutor&& exec) &&;

  template <typename Continuation>
  ContinuableFuture<Executor, auto> then(Continuation&& c) &&;
};

template <typename Executor, typename T>
struct SharedFuture
{
  template <typename RExecutor>
  ContinuableFuture<RExecutor, auto> via(RExecutor&& exec);

  template <typename Continuation>
  SharedFuture<Executor, auto> then(Continuation&& c);
};

template <typename T>
struct Promise
{
  void set_value(T value) &&;

  template <typename Error>
  void set_exception(Error exception) &&;
  bool valid() const;
};
```

Based on the declaration of the concepts, here are a few observations:

- A `FutureContinuation` can be invoked with a value or with an exception. It is a callable unit that consumes the value or exception of a future
- All futures (`SemiFuture`, `ContinuableFuture`, and `SharedFuture`) have a member function `via` that excepts an executor and returns a `ContinuableFuture`. `via` allows it to convert from one future type to a different one by using a different executor.
- Only a `ContinuableFuture` or a `SharedFuture` have a `then` continuation member function. The `then` member function takes a `FutureContinuation` and returns a `ContinuableFuture`.
- A `SharedFuture` is a non-uniquely owned future that is copyable.
- A Promise can set a value or an exception.

7.2.2.3 Future Work

The proposal 1054R0[20] left a few questions open.

- Forward progress guarantees for futures and promises.
- Requirements on synchronization for the use of futures and promises from non-concurrent execution agents.
- `std::future`/`std::promise` interoperability.
- Future unwrapping, both `future<future<T>>` and more advanced forms.
- `when_all`/`when_any`/`when_n`.
- `async`.

7.3 Transactional Memory

Transactional memory is based on the idea of a transaction from database theory. Transactional memory makes working with threads a lot easier for two reasons: first data races and deadlocks disappear, and second transactions are composable.

A transaction is an action that has the following properties **A**tomicity, **C**onsistency, **I**solation, and **D**urability (ACID). Except for the durability or storing the result of an action, all properties hold for transactional memory in C++. Now three short questions are left.

7.3.1 ACI(D)

What do atomicity, consistency, and isolation mean for an atomic block consisting of some statements?

---

[20]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1054r0.html

An atomic block

```
atomic{
    statement1;
    statement2;
    statement3;
}
```

Atomicity

Either all or none of the statements in the block are performed.

Consistency

The system is always in a consistent state. All transactions establish a total order.

Isolation

Each transaction runs in total isolation from other transactions.

How do these properties apply? A transaction remembers its initial state and is performed without synchronization. If a conflict happens during its execution, the transaction is interrupted and restored to its initial state. This rollback causes the transaction to be executed again. If the transaction's initial state still exists at the end of the transaction, the transaction is committed. Conflicts are typically detected with a tagged state reference.

A transaction is a kind of speculative action that is only committed if the initial state holds. In contrast to a mutex, it is an optimistic approach. A transaction is performed without synchronization. It is only be published if no conflict occurs. A mutex is a pessimistic approach. First, the mutex ensures that no other thread can enter the critical region. Next, the thread enters the critical region if it is the exclusive owner of the mutex, and hence all other threads are blocked.

C++ supports transactional memory in two flavors: synchronized blocks and atomic blocks.

7.3.2 Synchronized and Atomic Blocks

So far I only wrote about transactions. Now I write about synchronized blocks and atomic blocks. Both can be encapsulated in each other. To be more specific synchronized blocks are not transactions because they can execute transaction-unsafe. An example for a transaction-unsafe code would be a code like the output to the console, which can not be undone. For this reason, synchronized blocks are often called relaxed blocks.

7.3.2.1 Synchronized Blocks

Synchronized blocks behave like a global lock that protects them. This means that all synchronized blocks follow a total order, and in particular: all changes to a synchronized block are available in the next synchronized block. There is a *synchronizes-with* relation between the synchronized blocks because of the commit of the transaction *synchronizes-with* the next start of a transaction. Synchronized blocks can not cause a deadlock because they create a total order. While a classical

mutex protects a critical region of the program, a global lock of a synchronized block protects the total program.

This is the reason the following program is *well-defined*:

A synchronized block

```cpp
// synchronized.cpp

#include <iostream>
#include <vector>
#include <thread>

int i= 0;

void increment(){
  synchronized{
    std::cout << ++i << " ,";
  }
}

int main(){

  std::cout << '\n';

  std::vector<std::thread> vecSyn(10);
  for(auto& thr: vecSyn)
    thr = std::thread([]{ for(int n = 0; n < 10; ++n) increment(); });
  for(auto& thr: vecSyn) thr.join();

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

}
```

Although the variable `i` in line 7 is a global variable and the synchronized block operations are transaction-unsafe, the program is well-defined. Ten threads concurrently invoke the function `increment` (line 21) ten times, incrementing the variable `i` in line 11. Access to `i` and `std::cout` happens in total order. This is the characteristic of the synchronized block.

The program returns the expected result. The values for `i` are written in an increasing sequence, separated by a comma. For completeness, here is the output.

```
Datei   Bearbeiten   Ansicht   Lesezeichen   Einstellungen   Hilfe
rainer@suse:~> synchronized

1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12 ,13 ,14 ,15 ,16 ,17 ,18 ,19 ,20 ,21 ,22 ,23 ,24 ,25 ,26 ,27 ,28 ,29
0 ,31 ,32 ,33 ,34 ,35 ,36 ,37 ,38 ,39 ,40 ,41 ,42 ,43 ,44 ,45 ,46 ,47 ,48 ,49 ,50 ,51 ,52 ,53 ,54 ,55 ,56
7 ,58 ,59 ,60 ,61 ,62 ,63 ,64 ,65 ,66 ,67 ,68 ,69 ,70 ,71 ,72 ,73 ,74 ,75 ,76 ,77 ,78 ,79 ,80 ,81 ,82 ,83
4 ,85 ,86 ,87 ,88 ,89 ,90 ,91 ,92 ,93 ,94 ,95 ,96 ,97 ,98 ,99 ,100 ,

rainer@suse:~> █

                        rainer : bash
```

Incrementing with synchronized blocks

What about data races? You can have them with synchronized blocks. A small modification of the source code is sufficient to introduce a data race.

A data race with a synchronized block

```cpp
1    // nonsynchronized.cpp
2
3    #include <chrono>
4    #include <iostream>
5    #include <vector>
6    #include <thread>
7
8    using namespace std::chrono_literals;
9    using namespace std;
10
11   int i= 0;
12
13   void increment(){
14     synchronized{
15       cout << ++i << " ,";
16       this_thread::sleep_for(1ns);
17     }
18   }
19
20   int main(){
21
22     cout << '\n';
23
24     vector<thread> vecSyn(10);
25     vector<thread> vecUnsyn(10);
26
27     for(auto& thr: vecSyn)
28       thr = thread([]{ for(int n = 0; n < 10; ++n) increment(); });
29     for(auto& thr: vecUnsyn)
30       thr = thread([]{ for(int n = 0; n < 10; ++n) cout << ++i << " ,"; });
31
32     for(auto& thr: vecSyn) thr.join();
```

```
33      for(auto& thr: vecUnsyn) thr.join();
34
35      cout << "\n\n";
36
37  }
```

I let the synchronized block sleep for a nanosecond (line 16). At the same time, I access the output stream std::cout without a synchronized block (line 30). In total, 20 threads increment the global variable i, half of them without synchronization. The output shows the issue.

A data race with synchronized blocks

I put red circles around the issues in the output. These are the locations where at least two threads write ʻstd::cout at the same time. The C++11 standard guarantees that the characters are written atomically; that is not an issue. What is worse is that the variable iʻ is written by at least two threads. This is a data race; hence, the program has undefined behavior. If you look carefully at the output, you see a data race in action. The final result for the counter is 199 but should be 200. This means one of the intermediate values of the counter was overwritten.

The total order of synchronized blocks also holds for atomic blocks.

7.3.2.2 Atomic Blocks

You can execute transaction-unsafe code in a synchronized block but not in an atomic block. Atomic blocks are available in three forms: atomic_noexcept, atomic_commit, and atomic_cancel. The three suffixes _noexcept, _commit, and _cancel define how an atomic block manages an exception:

atomic_noexcept

    If an exception is thrown, std::abort is called, and the program aborts.

atomic_cancel

    In the default case, std::abort is called. This does not hold if a transaction-safe exception is thrown that is responsible for ending the transaction. In this case, the transaction is canceled, put to its initial state, and the exception is thrown.

atomic_commit
>   If an exception is thrown, the transaction is committed.

Transaction-safe exceptions are: `std::bad_alloc`[21], `std::bad_array_length`[22], `std::bad_array_new_length`[23], `std::bad_cast`[24], `std::bad_typeid`[25], `std::bad_exception`[26], `std::exception`[27], and all exceptions that are derived from one of these.

7.3.3 `transaction_safe` versus `transaction_unsafe` Code

You can declare a function as `transaction_safe` or attach the `transaction_unsafe` attribute to it.

transaction_safe versus transaction_unsafe

```cpp
int transactionSafeFunction() transaction_safe;

[[transaction_unsafe]] int transactionUnsafeFunction();
```

`transaction_safe` belongs to the type of the function. What does `transaction_safe` mean? A `transaction_safe` function is, according to the proposal N4265[28], a function that has a `transaction_safe` definition. This holds if the following properties **do not** apply to its definition:

- It has a `volatile` parameter or a `volatile` variable.
- It has `transaction-unsafe` statements.
- If the function uses a constructor or destructor of a class in its body that has a `volatile` non-static member.

Of course, this definition of `transaction_safe` is not sufficient because it uses the term `transaction_unsafe`. You can read the proposal N4265[29] for the details.

---

[21]http://en.cppreference.com/w/cpp/memory/new/bad_alloc
[22]https://www.cs.helsinki.fi/group/boi2016/doc/cppreference/reference/en.cppreference.com/w/cpp/memory/new/bad_array_length.html
[23]http://en.cppreference.com/w/cpp/memory/new/bad_array_new_length
[24]http://en.cppreference.com/w/cpp/types/bad_cast
[25]http://en.cppreference.com/w/cpp/types/bad_typeid
[26]http://en.cppreference.com/w/cpp/error/bad_exception
[27]http://en.cppreference.com/w/cpp/error/exception
[28]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4265.html
[29]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4265.html

7.4 Task Blocks

Task blocks use the well-known fork-join paradigm for the parallel execution of tasks. They are already part of the Technical Specification for C++ Extension Parallelism Version 2[30]; therefore, it's pretty probable that we get them with C++20.

Who invented it in C++? Both Microsoft with its Parallel Patterns Library (PPL)[31] and Intel with its Threading Building Blocks (TBB)[32] were involved in the proposal N4441[33]. Additionally, Intel used its experience with their Cilk Plus library[34].

The name fork-join is quite easy to explain.

7.4.1 Fork and Join

The most straightforward approach to explain the fork-join paradigm is a graphic.
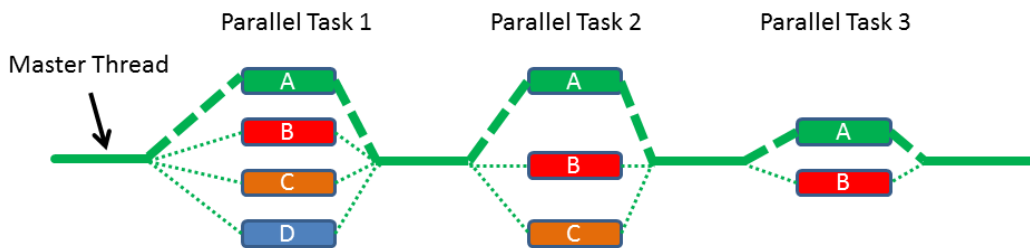
The fork-join paradigm

How does it work?

The creator invokes `define_task_block` or `define_task_block_restore_thread`. This call creates a task block that can create tasks, or it can wait for their completion. The synchronization is at the end of the task block. The creation of a new task is the fork phase; the task block's synchronization is the join phase of the workflow. Admittedly that was a simple description. Let's have a look at a piece of code.

---

[30]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4742.html
[31]https://en.wikipedia.org/wiki/Parallel_Patterns_Library
[32]https://en.wikipedia.org/wiki/Threading_Building_Blocks
[33]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf
[34]https://en.wikipedia.org/wiki/Cilk

Define a task block

```cpp
template <typename Func>
int traverse(node& n, Func && f){
    int left = 0, right = 0;
    define_task_block(
        [&](task_block& tb){
            if (n.left) tb.run([&]{ left = traverse(*n.left, f); });
            if (n.right) tb.run([&]{ right = traverse(*n.right, f); });
        }
    );
    return f(n) + left + right;
}
```

traverse is a function template that invokes function f on each node of its tree. The keyword define_-task_block defines the task block. The task block tb can start a new task in this block. This exactly happens at the left and right branches of the tree in lines 6 and 7. Line 9 is the end of the task block and hence the synchronization point.

HPX (High Performance ParalleX)

The above example is from the documentation for the HPX (High-Performance ParalleX)[35] framework, which is a general-purpose C++ runtime system for parallel and distributed applications of any scale. HPX has already implemented many in this chapter presented features of the upcoming C++20/23 standards.

You can define a task block by using either the function define_task_block or the function define_-task_block_restore_thread.

7.4.2 `define_task_block` versus `define_task_block_restore_thread`

The subtle difference is that the function define_task_block_restore_thread guarantees in contrast to the function define_task_block that the creator thread of the task block is the same thread that runs after the task block.

---

[35]http://stellar.cct.lsu.edu/projects/hpx/

`define_task_block` versus `define_task_block_restore_thread`

```
1   ...
2   define_task_block([&](auto& tb){
3     tb.run([&]{[] func(); });
4     define_task_block_restore_thread([&](auto& tb){
5       tb.run([&]([]{ func2(); });
6       define_task_block([&](auto& tb){
7         tb.run([&]{ func3(); }
8       });
9       ...
10      ...
11    });
12    ...
13    ...
14  });
15  ...
16  ...
```

Task blocks ensure that the outermost task block's creator thread (lines 2 - 14) is the same thread that runs the statements after finishing the task block. This means that the thread that executes line 2 is the same thread that executes lines 15 and 16. This guarantee does not hold for nested task blocks; therefore, the task block's creator thread in lines 6 - 8 does not automatically execute lines 9 and 10. If you need that guarantee, you should use the function `define_task_block_restore_thread` (line 4). It holds that the creator thread executing line 4 is the same thread running lines 12 and 13.

7.4.3 The Interface

A task block has a minimal interface. You can not construct, destroy, copy, or move an object of the class `task_block`; you have to use either function `define_task_block` or `define_task_block_restore_-thread`. The task_block `tb` is in the scope of the defined task block active and can start new tasks (`tb.run`) or wait (`tb.wait`) until the task is done.

The minimal interface of a task block

```
1   define_task_block([&](auto& tb){
2     tb.run([&]{ process(x1, x2) });
3     if (x2 == x3) tb.wait();
4     process(x3, x4);
5   });
```

What is the code snippet doing? In line 2 a new task is started. This task needs the data `x1` and `x2`. Line 4 uses the data `x3` and `x4`. If `x2 == x3` is true, the variables must be protected from shared access. This is why the task block `tb` waits until the task in line 2 is done.

If the functions `task_block::run` or `task_block::wait` detect that an exception is pending within the current task block they throw an exception of kind `task_cancelled_exception`.

7.4.4 The Scheduler

The scheduler manages which thread is running. This means that it is no longer the responsibility of the programmer to decide who executes the task. Threads are just an implementation detail.

There are two strategies for executing the newly created task. The parent represents the creator thread and the child the new task.

Child stealing
> The scheduler steals the task and executes it.

Parent stealing
> The task block tb itself executes the task. Now the scheduler steals the parent.

Proposal N4441[36] supports both strategies.

---

[36]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf

7.5 Data-Parallel Vector Library

The data-parallel vector library provides data-parallel (SIMD) programming via vector types. SIMD[37] stands for **S**ingle **I**nstruction **M**ultiple **D**ata and means that one operation is performed on many data in parallel. Each modern processor architecture provides SIMD extensions, but there exists no standardized C++ interface to use them. The following table gives an overview of various SIMD extensions.

SIMD Extensions

Architecture	SIMD Extension	Register Width
ARM 32 / 64	NEON	128 bit
Power / PowerPC	AltiVec	128 bit
	VSX	128 bit
x86 / AMD 64	MMX / 3DNow	64 bit
	SSE	128 bit
	AVX / AVX2 / AVX-512	128 bit / 256 bit / 512 bit

For example, the SSE[38] extension allows it to add four 32 bit `int`s in parallel. The effect is that it is four times faster than adding four `int`s sequential because single and vector operations are equally fast on modern CPU's.

---

[37] https://en.wikipedia.org/wiki/SIMD
[38] https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

🔑 Auto-Vectorisation

Auto-vectorisation is the compiler's job to produce the best machine code for the given architecture. Often there are many obstacles to generate the most efficient code for a given machine:

According to the chapter Using Automatic Vectorization[39] to the INTEL C++ Compiler they are:

- Non-contiguous memory access lead to inefficient loads/stores
- Data dependencies between iteration steps.
- Countable (the number of iterations of a loop can be determined before the loop is entered)

There are more challenges for auto-vectorisation

- Dependent execution or function calls in loops
- Outer loops (loops containing other loops)
- Threads (calls to mutexes or atomics)

The SIMD extension in C++ consists now of two data-parallel vectors and special operations on them.

7.5.1 Data-Parallel Vectors

The data-parallel vectors are `std::simd` and `std::simd_mask`:

```cpp
template< class T, class Abi = simd_abi::compatible<T> >
class simd;

template< class T, class Abi = simd_abi::compatible<T> >
class simd_mask;
```

- `T` is the element type of the vectors.
- `Abi` stands for the number of elements and storage.

The element type of `simd_mask` is `bool` and `simd_mask<T, AbT>::size()` is always `simd<T, AbT>::size()`.

7.5.2 The Interface of the Data-Parallel Vectors

The data-parallel vector library supports tags for the ABI and alignment, various operations, and traits.

---

[39] https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-using-automatic-vectorization

7.5.2.1 Tags

The tags let you specify and ask for the ABI and the alignment.

7.5.2.1.1 ABI Tags

The ABI tag indicates a choice of size and binary representation for objects of the vector type.

ABI tags

ABI tags	Description
scalar	Tag type for storing a single element.
fixed_size	Tag type for storing a specified number of elements.
compatible	Tag type that ensures ABI compatibility.
native	Tag type that is most efficient.
max_fixed_size	The maximum number of elements guaranteed to be supported by fixed.

7.5.2.1.2 Alignment Tags

The alignment tags indicate the alignment of the elements and the vector type.

Alignment tags

Alignment tags	Description
element_aligned_tag and element_aligned	Tag type that indicates the alignment of the elements.
vector_aligned_tag and vector_aligned	Tag type that indicates the alignment of the vector type.
overaligned_tag and overaligned	Tag type that indicates the specified alignment.

7.5.2.2 Where Expression

The where expression abstracts the operation of selected elements of a given object of arithmetic or data-parallel type.

Where expression

Where expression	Description
`const_where_expression`	Selects elements with non-mutating operations.
`where_expression`	Selects elements with mutating operations.
`where`	Generated `const_where_expression` and `where expression`.

7.5.2.3 Casts

Cast operations cast element-wise or split or concat between SIMD objects and single ones.

Casts

Where expression	Description
`simd_cast` and `static_simd_cast`	Performs a element-wise static cast.
`to_fixed_size`, `to_compatible`, and `to_native`	Performs an element-wise ABI cast.
`split`	Splits a single SIMD object to multiple objects.
`concat`	Concatenates multiple SIMD objects into a single SIMD object.

7.5.2.4 Algorithms

The SIMD algorithms are applied to two SIMD objects and return a SIMD object.

Algorithms

Algorithms	Description
`min`	Applies the `min` operation element-wise and returns a SIMD object.
`max`	Applies the `min` operation element-wise and returns a SIMD object.
`minmax`	Applies the `minmax` operation element-wise and returns a SIMD object.
`clamp`	Applies the `clamp` operation element-wise and returns a SIMD object.

The clamp algorithm applies element-wise the following operation for i in $[0, \text{size}()]$:

This means, the resulting SIMD object `sim` will have the following values:

- `v[i] < lo[i]`: `sim[i] = lo[i]`
- `lo[i] < v[i] < hi[i]`: `sim[i] = vi[i]`
- `hi[i] < v[i]`: `sim[i] = hi[i]`

7.5.2.5 Reduction

Reduction reduces the SIMD vector to a single element.

Reduction

Reduction	Description
reduce	Reduces the SIMD vector to a single element.
hmin	Returns the minimum element.
hmax	Return the maximum element.

7.5.2.6 Mask Reduction

Applies the predicate only to these elements of the SIMD vector `v[i]` for which the mask `m[i] == true` holds.

Mask Reduction

Mask Reduction	Description
all_of, any_of, none_of, and some_of	Applies the predicates on each element `v[i]` for which the mask `m[i] == true` holds.
popcount	Return the number of `true` values.
find_first_set and find_last_set	Return the position of the first or last true value.

7.5.2.7 Traits

Traits give you SIMD characteristics at compile time.

	Traits
Traits	**Description**
`is_abi_tag` and `is_abi_tag_v`	Checks if a type is an ABI tag type.
`is_simd` and `is_simd_v`	Checks if a type is SIMD type.
`is_simd_mask` and `is_simd_mask_v`	Checks if a type is a simd_mask type.
`is_simd_flag_type` and `is_simd_flag_type_v`	Checks if a type is a SIMD flag type.
`simd_size` and `simd_size_v`	Returns the number of elements of a given element type and ABI.
`memory_alignment` and `memory_alignment_v`	Returns an appropriate alignment for `vector_aligned`.
`abi_for_size` and `abi_for_size_t`	Returns an ABI type for a given element type and number of elements.

Distilled Information

- It isn't easy to make predictions about the future, but I try.
- An executor consists of a set of rules about where, when, and how to run a callable. They are the basic building block to execute and specify if callables should run on an arbitrary thread, a thread pool, or even single-threaded without concurrency.
- Tasks called promises and futures, introduced in C++11, have a lot to offer, but they also have drawbacks: tasks are not composable into powerful workflows. That limitation does not hold for the extended futures in C++23. Therefore, an extended future becomes ready when its predecessor becomes ready, when any one of its predecessors becomes ready, or when all of its predecessors becomes ready.
- Transactional memory is based on the ideas underlying transactions in database theory. A transaction is an action that provides the first three properties of ACID database transactions: **A**tomicity, **C**onsistency, and **I**solation. The new standard has transactional memory in two flavors: synchronized blocks and atomic blocks. Both behave as if a global lock protected them.
- Task blocks implement the fork-join paradigm in C++. They consist of a fork phase in which you launch tasks and a join phase in which you synchronize them.
- The data-parallel vector library provides data-parallel (SIMD) programming via vector types. SIMD means that one operation is performed on many data in parallel.

Patterns

8. Patterns and Best Practices
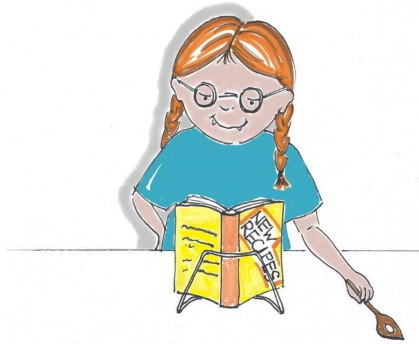
Cippi uses proven knowledge from her ancestors

This chapter aims to give you an idea of what patterns are and what they are suitable for. My pragmatic view is informal and is wearing C++ glasses. For a more formal and comprehensive discussion of this topic, I provide links to further literature.

First of all: What is a pattern?

Pattern
> "Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution." Christopher Alexander[1]

To say it more informally. A pattern is a well-established and documented solution to a design challenge in a specific domain.

8.1 History

The father of patterns is the already mentioned Cristopher Alexander, whose patterns about the nature of human-centered design for towns, buildings, and construction, in general, were seminal for software design. 1994, the so-called **Gang of Four** (Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides) published their book Design Patterns: Elements of Reusable Object-Oriented Software[2]. The book includes 23 software design patterns aimed at object-oriented software design. The patterns fall into three categories creational, structural, and behavioral. The book defined the vocabulary for the entire software industry. Here are a few of the most known design patterns:

---

[1]https://en.wikipedia.org/wiki/Christopher_Alexander
[2]https://en.wikipedia.org/wiki/Design_Patterns

- Creational
  - Factory member function pattern
  - Singleton pattern
- Structural
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Proxy
- Behavioral
  - Command
  - Iterator
  - Observer
  - Strategy
  - Template method
  - Visitor

One year later, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal published their highly influential book Pattern-Oriented Software-Architecture: A System of Patterns[3], or in short POSA. This book was the starting point of a series of five books. It was published in 1995 and has three categories of patterns: architectural patterns, design patterns, and idioms. Many of the patterns are now common terminology:

- Architectural Patterns
  - Layers
  - Pipes-and_Filters
  - Broker
  - Model-View-Controller
- Design Patterns
  - Master-Slave
  - Publish-Subscriber
- Idioms
  - Counted-Pointer

What is the difference between these three categories? The focus of the architectural patterns is the entire software system. They are more abstract than the design patterns, which define the interplay between the subsystems. Idioms are implementations of an architectural or design pattern in a given programming language. They have the lowest abstraction level of all three.

Each of the books two to five of the POSA series has a different focus. They deal with "Patterns for Concurrent and Networked Objects" (volume 2), with "Patterns for Resource Management" (volume 3), with "A Pattern Language for Distributed Computing" (volume 4), and "On Patterns and Pattern Languages" (volume 5). The book sections Synchronization Patterns and Concurrent Architecture are strongly influenced by the second volume of the series.

---

[3]https://www.wiley.com/WileyCDA/Section/id-406899.html

8.2 Invaluable Value

Patterns added priceless value to software development in general. Of course, this also holds for concurrency in particular. The added value boils down to three points. A well-defined terminology, improved documentation, and learning from the best.

The **well-defined terminology** means that software developers can now use common and unambiguous vocabulary. Misunderstandings or verbose explanations are mainly stories of the past. Suppose a software developer asks for advice about implementing a family of similar algorithms to exchange them during runtime. In that case, the answer may be as short as use the strategy pattern. If the software developer knows the strategy pattern, he can immediately think about its consequences; if not, he can look it up in the literature.

The **documentation improves** in two aspects. First, the documentation about the software system, whether in a graphical or textual description, improves because if I read that the observer pattern is used, I know that the system has a kind of Subject/Observer structure. This means that the observer will register or unregister themselves on the subject, and the subject will send notifications to all of its observers if necessary. Secondly, I can directly jump into the source code and search for keywords such as observer, subject or notify because I have an idea about the concrete implementation.

Patterns are just about **learning from the best**. You learn from the documented experience from the best and do not repeat their mistakes. They provide proven solutions for typical problems and help to master complexity. Each pattern includes information on when you should use it, the consequences of using it, how to implement it, and its known usages.

8.3 Pattern versus Best Practices

You may wonder that I write in this part of the book also about best practices. Why? Honestly, I often had an intensive fight if a method such as immutable values or pure functions is a pattern or just best practice. Patterns are documented best practices and, therefore, quite related. I learned from these fight lessons.

- Both terms can not be precisely distinguished.
- If the practice I describe is a well-defined pattern, I put it in the pattern bucket.
- If the practice has a tip character and no formal defined structure, I put it in the best practices bucket.
- Today's best practices may become tomorrow's well-defined patterns.

8.4 Anti-Pattern

A pattern stands for best practices, an anti-pattern stands for a lesson learned, or to use the words from Andrew Koenig[4]: "Those that describe a bad solution to a problem which resulted in a bad

---

[4]https://en.wikipedia.org/wiki/Andrew_Koenig_(programmer)

situation." If you carefully read the literature on concurrency patterns, you often find the double-checked locking pattern. The double-checked locking pattern's general idea is, in one sentence, the thread-safe initialization of shared state in an optimized way. This shared state is typically a singleton[5]. I intentionally put the double-checked locking pattern in the case studies chapter of this book to emphasize it explicitly: naively usage of the double-checked locking pattern may end in undefined behavior. The issues of the double-checked locking pattern boil essentially down to the issues of the singleton pattern.

If you want to use the singleton pattern, you have to think about the following challenges:

- First and foremost, the singleton is a global object. Due to this fact, the singleton usage is most of the time not visible in the interface. The effect is that you have a hidden dependency in the code which uses the singleton.

- A singleton is a static object and is, therefore, once created, never destroyed. Its lifetime ends with the lifetime of the program.

- If your static class member, such as a singleton, depends on another static member defined in another translation unit, you have no guarantee which one is initialized first. The probability that it fails is 50 %.

- A singleton is often used when just an instance of a class would also do the job. Many developers use the singleton to prove that they know their design pattern.

Distilled Information

- Patterns are documented best practices from the best. They "... express a relation between a certain context, a problem, and a solution." Christopher Alexander[6].
- Patterns provide more than just best practices. They provide a well-defined terminology and improve documentation.
- A anti-pattern stands for a lesson learned or a proven way to shot yourself in the foot.

---

[5]https://en.wikipedia.org/wiki/Singleton_pattern
[6]https://en.wikipedia.org/wiki/Christopher_Alexander

9. Synchronization Patterns

Cippi directs the train

The main concern when you deal with concurrency is shared, mutable state or as Tony Van Eerd[1] put it in his CppCon 2014 talk "Lock-free by Example": "Forget what you learned in Kindergarten (ie stop Sharing)".

Mutable, shared state

A necessary condition for a data race is a mutable, shared state. If you handle sharing or mutation, no data race can happen. This is exactly the focus of the next two sections: Dealing with Sharing and Dealing with Mutation.

---

[1]https://github.com/tvaneerd

9.1 Dealing with Sharing

If you don't share, no data races can happen. Not sharing means that your thread works on local variables. This can be achieved by copying the value, by using thread-specific storage, or by transferring the result of a thread to its associated future via a protected data channel. The patterns in this section are quite obvious, but for completeness, I will present them with a short explanation.

9.1.1 Copied Value

If a thread gets its arguments by copy and not by reference, there is no need to synchronize access to any data. No data races and no lifetime issues are possible.

9.1.1.1 Data Races with References

The following program creates three threads. One thread gets its argument by copy, the other by reference, and the last by constant reference.

Data races with references
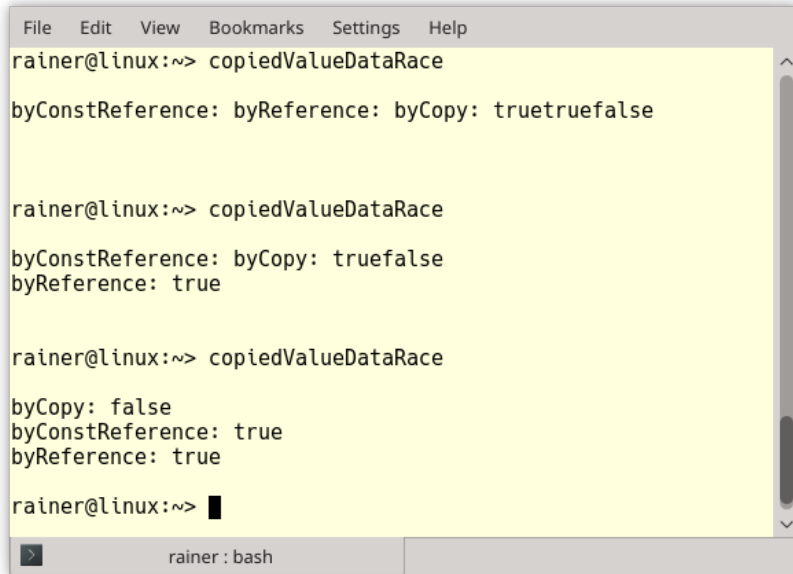
```cpp
1   // copiedValueDataRace.cpp
2
3   #include <functional>
4   #include <iostream>
5   #include <string>
6   #include <thread>
7
8   using namespace std::chrono_literals;
9
10  void byCopy(bool b){
11      std::this_thread::sleep_for(1ms);
12      std::cout << "byCopy: " << b << '\n';
13  }
14
15  void byReference(bool& b){
16      std::this_thread::sleep_for(1ms);
17      std::cout << "byReference: " << b << '\n';
18  }
19
20  void byConstReference(const bool& b){
21      std::this_thread::sleep_for(1ms);
22      std::cout << "byConstReference: " << b << '\n';
23  }
24
25  int main(){
```

```
26
27        std::cout << std::boolalpha << '\n';
28
29        bool shared{false};
30
31        std::thread t1(byCopy, shared);
32        std::thread t2(byReference, std::ref(shared));
33        std::thread t3(byConstReference, std::cref(shared));
34
35        shared = true;
36
37        t1.join();
38        t2.join();
39        t3.join();
40
41        std::cout << '\n';
42
43    }
```

Each thread sleeps for one millisecond (lines 11, 16, and 21) before displaying the boolean value. Only thread t1 has a local copy of the boolean and has, therefore, no data race. The program's output shows that the boolean values of thread t2 and t3 are modified without synchronization.

Data races with references

I made my example `copiedValueDataRace.cpp` an assumption that is trivial for a boolean but not for a more complex type. Passing the argument by copy is data race free if the argument is a so-called value object.

Value Object

A value object is an object whose equality is not based on identity but based on its state. Value objects should be immutable so that they stay equal for their lifetime if created equal. If you pass a value object by copy to a thread, there is no need to synchronize its access. Due to the article ValueObject[2] from Martin Fowler, you can think of two classes of objects: value objects and reference objects.

9.1.1.1.1 When a reference is actually a copy

You may think that the thread `t3` in the previous example `copiedValueDataRace.cpp` can just be replaced with `std::thread t3(byConstReference, shared)`. The program compiles and runs but what seems like a reference is a copy. The reason is that `std::decay`[3] is applied to each argument of

---

[2]https://martinfowler.com/bliki/ValueObject.html
[3]https://en.cppreference.com/w/cpp/types/decay

the thread. `std::decay` performs lvalue-to-rvalue, array-to-pointer, and function-to-pointer implicit conversions to its type `T`. In particular, it invokes in this case `std::remove_reference`[4] on the type `T`.

The program `perConstReference.cpp` uses a non-copyable type `NonCopyableClass`.

Implicit copying of a per-reference thread argument

```cpp
// perConstReference.cpp

#include <thread>

class NonCopyableClass{
    public:

    // the compiler generated default constructor
    NonCopyableClass() = default;

    // disallow copying
    NonCopyableClass& operator = (const NonCopyableClass&) = delete;
    NonCopyableClass (const NonCopyableClass&) = delete;

};

void perConstReference(const NonCopyableClass& nonCopy){}

int main(){

    NonCopyableClass nonCopy;

    perConstReference(nonCopy);

    std::thread t(perConstReference, nonCopy);
    t.join();

}
```

The object `nonCopy` (line 21) is not copyable. This is fine if I invoke the function `perConstReference` with the argument `nonCopy` because the function accepts its argument per constant reference. Using the same function in the thread `t` (line 25) causes GCC 6 to generate a verbose compiler error with more than 300 lines:

---

[4]https://en.cppreference.com/w/cpp/types/remove_reference

A verbose error message

The error message's essential part is in the middle of the screenshot in red rounded rectangle: "error: use of deleted function". The copy-constructor of the class `NonCopyableClass` is not available.

Use of deleted function

9.1.1.2 Lifetime Issues with References

If your thread uses its argument by reference and you detach the thread, you have to be extremely careful. The small program `copiedValueLifetimeIssues.cpp` has undefined behavior.

Lifetime issues with references
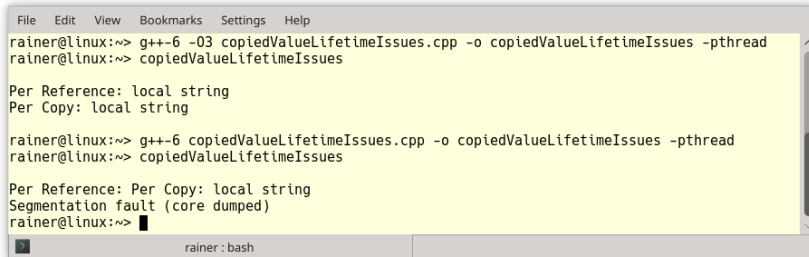
```cpp
// copiedValueLifetimeIssues.cpp

#include <iostream>
#include <string>
#include <thread>

void executeTwoThreads(){

    const std::string localString("local string");

    std::thread t1([localString]{
        std::cout << "Per Copy: " << localString << '\n';
    });

     std::thread t2([&localString]{
        std::cout << "Per Reference: " << localString << '\n';
    });

    t1.detach();
    t2.detach();
}

using namespace std::chrono_literals;

int main(){

    std::cout << '\n';

    executeTwoThreads();

    std::this_thread::sleep_for(1s);

    std::cout << '\n';

}
```

executeTwoThreads (lines 7 - 21) starts two threads. Both threads are detached (lines 19 and 20) and print the local variable localString (line 9). The first thread captures the local variable by copy, and the second the local variable by reference. For simplicity reasons, in both cases, I used a lambda expression to bind the arguments.

Because the executeTwoThreads function doesn't wait until the two threads have finished, the thread t2 refers to the local string, which is bound to the lifetime of the invoking function. This causes

undefined behavior. Curiously, with GCC 6 the maximum optimized executable `-O3` seems to work, and the non-optimized executable crashes.

Lifetime issues with references

9.1.1.3 Further Information

- ValueObject[5]
- Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing[6]

9.1.2 Thread-Specific Storage

Thread-specific or thread-local storage allows multiple threads to use local storage via a global access point. By using the storage specifier `thread_local`, a variable becomes a thread-local variable. This means you can use the thread-local variable without synchronization.

Here is a typical use-case. Assume you want to calculate the sum of all elements of a vector `randValues`. Doing it with a range-based for-loop is straightforward.

```
// calculateWithLoop.cpp

...

unsigned long long sum = { };
for (auto n: randValues) sum += n;
```

But your PC has four cores. Therefore, you make a concurrent program out of the sequential program.

```
// threadLocalSummation.cpp

...

thread_local unsigned long long tmpSum = 0;

void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end) {
    for (auto i = beg; i < end; ++i){
        tmpSum += val[i];
    }
    sum.fetch_add(tmpSum, std::memory_order_relaxed);
}

...

std::atomic<unsigned long long> sum{};

std::thread t1(sumUp, std::ref(sum), std::ref(randValues), 0, fir);
std::thread t2(sumUp, std::ref(sum), std::ref(randValues), fir, sec);
std::thread t3(sumUp, std::ref(sum), std::ref(randValues), sec, thi);
std::thread t4(sumUp, std::ref(sum), std::ref(randValues), thi, fou);
```

You put the range-based for-loop into a function, let each thread calculate a fourth of the sum in the thread-local variable `tmpSum`. The line `sum.fetch_add(tmpSum, std::memory_order_relaxed)` finally sums up all values in the atomic `sum`.

I already discussed the various ways to sum up all vector elements in the chapter Calculating the Sum of a Vector. You can study the entire program there.

🔑 Use the Algorithms of the Standard Template Library.

You should not write a loop if an algorithm of the standard template library can do the job. In this case, it's the job of `std::accumulate`[7] to sum up all elements of the vector: `sum = std::accumulate(randValues.begin(), randValues.end(), 0)`. Since C++17, you can use the `std::reduce` algorithm which is the parallelisable version of `std::accumulate`: `sum = std::reduce(std::execution::par, randValues.begin(), randValues.end(), 0)`.

9.1.2.1 Further Information

- Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects[8]

---

[7]https://en.cppreference.com/w/cpp/algorithm/accumulate
[8]https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/

9.1.3 Future

C++11 provides futures and promise in three flavors: `std::async`, `std::packaged_task`, and the pair `std::promise` and `std::future`. The term promise goes back to the seventies. The future is a read-only placeholder for the value which a promise sets. From the synchronization perspective, a promise/future pair's critical property is that a protected data channel connects both.

There are a few decisions to make when implementing a future.

- A future can ask for its value implicitly or explicitly with the get call, such as in C++.
- A future can eagerly or lazily start the computation. Only the promise `std::async` supports lazy evaluation via launch policies.

```cpp
auto lazyOrEager = std::async([]{ return "LazyOrEager"; });
auto lazy = std::async(std::launch::deferred, []{ return "Lazy"; });
auto eager = std::async(std::launch::async, []{ return "Eager"; });

lazyOrEager.get();
lazy.get();
eager.get();
```

If I don't specify a launch policy, it's up to the system to start the job eager or lazy. Using the launch policy `std::launch::async`, a new thread is created, and the promise immediately starts its job. This is in contrast to the launch policy `std::launch::deferred`. The call `eager.get()` starts the promise. Additionally, the promise is executed in the thread requesting the result with `get`.

- A future can block or throw an exception if the value of the promise is not available. C++11 blocks, in this case, the `wait` or `get` call. Additionally, you can wait with a timeout on the promise (`wait_for` and `wait_until`).
- There are various ways to implement futures: coroutines[9], generators[10], or channels[11].

9.1.3.1 Further Information

- Futures and promises[12]

9.2 Dealing with Mutation

If you don't write and read data concurrently, no data race can happen. The easiest way to achieve this is by immutable values. Additionally, to this best practice, there are two typical strategies.

---

[9]https://en.wikipedia.org/wiki/Coroutine
[10]https://en.wikipedia.org/wiki/Generator_(computer_programming)
[11]https://en.wikipedia.org/wiki/Channel_(programming)
[12]https://en.wikipedia.org/wiki/Futures_and_promises

First, protect the critical sections by a lock such as a scoped lock or strategized locking. In object-oriented design, the critical section is typically an object including its interface. The Thread-Safe Interface protects the entire object. If the variable is never modified, synchronization is no need by using an expensive lock or an atomic. In this case, you only have to ensure that the variable is initialized in a thread-safe way. I already discussed a few possibilities in the previous section thread-safe initialization and the case studies about thread-safe initialization of a singleton, including the infamous double-checked locking pattern.

Second, the modifying thread just signals when it is done with its work. This is the strategy of guarded suspension.

9.2.1 Scoped Locking

Scoped locking is the idea of RAII applied to a mutex. Scoped locking is also known as synchronized block and guard. The key idea of this idiom is to bind the resource acquisition and release to an object's lifetime. As the name suggests, the lifetime of the object is scoped. Scoped means that the C++ run time is responsible for object destruction and, therefore, for releasing the resource.

The class ScopedLock implements scoped locking.

Scoped Locking

```cpp
1   // scopedLock.cpp
2
3   #include <iostream>
4   #include <mutex>
5   #include <new>
6   #include <string>
7
8   class ScopedLock{
9     private:
10       std::mutex& mut;
11     public:
12       explicit ScopedLock(std::mutex& m): mut(m){
13           mut.lock();
14           std::cout <<  "Lock the mutex: " << &mut <<   '\n';
15       }
16       ~ScopedLock(){
17           std::cout << "Release the mutex: " << &mut << '\n';
18           mut.unlock();
19       }
20   };
21
22   int main(){
23
24     std::cout << '\n';
```

```
25
26     std::mutex mutex1;
27     ScopedLock scopedLock1{mutex1};
28
29     std::cout << "\nBefore local scope" << '\n';
30     {
31       std::mutex mutex2;
32       ScopedLock scopedLock2{mutex2};
33     }
34     std::cout << "After local scope" << '\n';
35
36     std::cout << "\nBefore try-catch block" << '\n';
37     try{
38         std::mutex mutex3;
39         ScopedLock scopedLock3{mutex3};
40         throw std::bad_alloc();
41     }
42     catch (std::bad_alloc& e){
43         std::cout << e.what();
44     }
45     std::cout << "\nAfter try-catch block" << '\n';
46
47     std::cout << '\n';
48
49 }
```
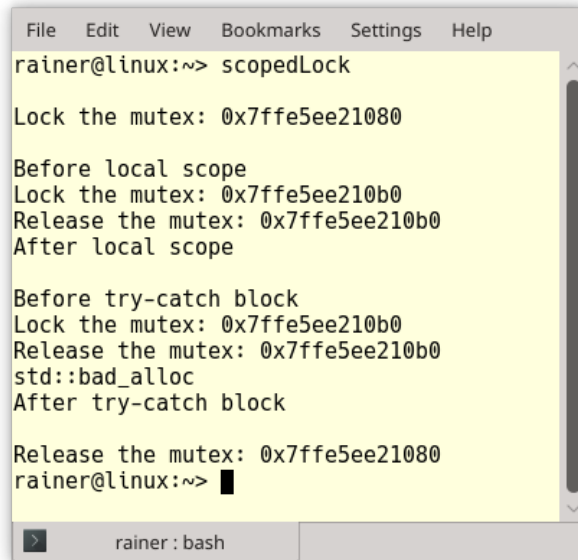
ScopedLock gets its mutex by reference (line 12). The mutex is locked in the constructor (line 13) and unlocked in the destructor (line 18). Thanks to the RAII idiom, the object's destruction and, therefore, the unlocking of the mutex is done automatically.

Scoped locking has the following advantages and disadvantages.

- Advantage:
    - Robustness, because the locks are automatically acquired and released
- Disadvantages:
    - Recursive locking of a std::mutex is undefined behavior and may typically cause a deadlock
    - Locks are not automatically released if the c function longjmp[13] is used; longjpm does not call C++ destructors of scoped objects

---

[13]https://en.cppreference.com/w/cpp/utility/program/longjmp

Deterministic destruction with scoped locking

The scope of scopedLock1 ends at the end of the main function. Consequentially, mutex1 is unlocked. The same holds for mutex2 in line 31 and mutex3 in line 38. They are automatically unlocked at the end of there local scopes (lines 33 and 41). For mutex3, the destructor of the scopedLock3 is also invoked if an exception occurs. Interestingly, mutex3 reuses the memory of mutex2 because both have the same address.

C++17 supports locks in four variations. C++ has a std::lock_guard / std::scoped_lock for the simple, and a std::unique_lock / std::shared_lock for the advanced use cases such as the explicit locking or unlocking of the mutex. You can read the details in the subsection locks.

9.2.1.1 Further Information

- Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects[14]

9.2.2 Strategized Locking

Assume you write code such as a library, which should be used in various domains, including concurrent ones. To be on the safe side, you protect the critical sections with a lock. If your library now runs in a single-threaded environment, you have a performance issue because you implemented an expensive synchronization mechanism that is unnecessary. Now, strategized locking comes to your rescue.

---

[14]https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/

Strategized locking is the idea of the strategy pattern applied to locking. This means putting your locking strategy into an object and making it into a pluggable component of your system. First of all: What is the strategy pattern?

9.2.2.1 Strategy Pattern
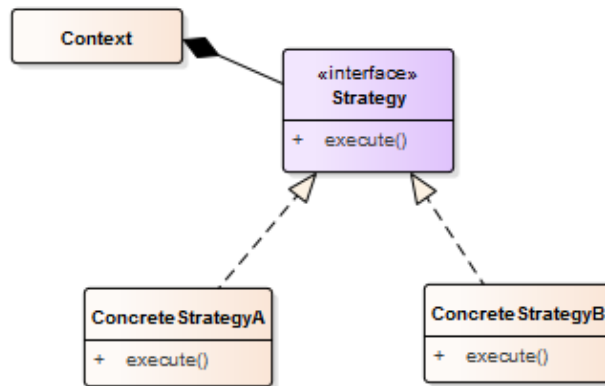
The Strategy Pattern

The strategy pattern is one of the book's classic behavioral patterns Design Patterns: Elements of Reusable Object-Oriented Software[15]. It's also known as a policy. The critical idea is defining a family of algorithms and encapsulating each algorithm in an object. Consequently, the algorithms become pluggable components.

The Strategy Pattern

```cpp
1   // strategy.cpp
2
3   #include <iostream>
4   #include <memory>
5
6   class Strategy {
7   public:
8       virtual void operator()() = 0;
9       virtual ~Strategy() = default;
10  };
11
12  class Context {
13      std::shared_ptr<Strategy> _strat;
14  public:
15      explicit Context() : _strat(nullptr) {}
```

[15]https://en.wikipedia.org/wiki/Design_Patterns
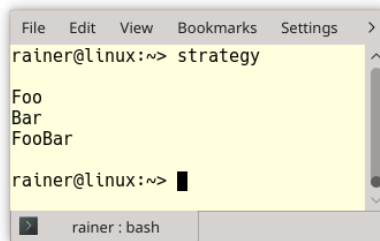
```
16        void setStrategy(std::shared_ptr<Strategy> strat)  { _strat = strat; }
17        void strategy()  { if (_strat) (*_strat)(); }
18    };
19
20    class Strategy1 : public Strategy {
21        void operator()() override {
22            std::cout << "Foo" << '\n';
23        }
24    };
25
26    class Strategy2 : public Strategy {
27        void operator()() override {
28            std::cout << "Bar" << '\n';
29        }
30    };
31
32    class Strategy3 : public Strategy {
33        void operator()() override {
34            std::cout << "FooBar" << '\n';
35        }
36    };
37
38    int main() {
39
40        std::cout << '\n';
41
42        Context con;
43
44        con.setStrategy( std::shared_ptr<Strategy>(new Strategy1) );
45        con.strategy();
46
47        con.setStrategy( std::shared_ptr<Strategy>(new Strategy2) );
48        con.strategy();
49
50        con.setStrategy( std::shared_ptr<Strategy>(new Strategy3) );
51        con.strategy();
52
53        std::cout << '\n';
54
55    }
```

The abstract class Strategy in lines 6 to 10 defines the strategy. Each particular strategy, such as Strategy1 (line 20), Strategy2 (line 26), or Strategy3 (line 32), has to support the function call operator (line 8). The Context is the user of various strategies. It sets the particular strategy in line 16 and

executes it in line 17. Because the Context executes the strategy via a pointer to the Strategy class, the overridden member functions of Strategy1, Strategy2, and Strategie3 can be private. The context con uses various strategies.

The Strategy Pattern applied

9.2.2.2 Implementation

Strategized locking often applies scoped locking. There are two typical ways to implement strategized locking: run-time polymorphism (object-orientation) or compile-time polymorphism (templates). Both ways improve the customization and extension of the locking strategy, ease the maintenance of the system, and support the reuse of components. Also, implementing the strategized locking at run-time or compile-time polymorphism differ in various aspects

- Advantages:
    - Run-time Polymorphism
        * allows it to configure the locking strategy during run time.
        * is easier to understand for developers who have an object-oriented background.
    - Compile-Time Polymorphism
        * has no abstraction penalty.
        * has a flat hierarchy.
- Disadvantages:
    - Run-time Polymorphism
        * needs an additional pointer indirection.
        * may have a deep derivation hierarchy.
    - Compile-Time Polymorphism
        * may generate a very lengthy message in the error case, but the use of concepts such as BasicLockable in C++20 causes concise error messages.

After this theoretical discussion, I implement the strategized locking in both variations. The strategized locking supports, in my example, no-locking, exclusive locking, and shared locking. For simplicity reasons, I used internally already existing mutexes. Additionally, the strategized locking models scoped locking.

9.2.2.2.1 Runtime Polymorphism

The program strategizedLockingRuntime.cpp presents three different mutexes.

Strategized Locking via Runtime Polymorphism

```cpp
1    // strategizedLockingRuntime.cpp
2
3    #include <iostream>
4    #include <mutex>
5    #include <shared_mutex>
6
7    class Lock {
8    public:
9        virtual void lock() const = 0;
10       virtual void unlock() const = 0;
11   };
12
13   class StrategizedLocking {
14       Lock& lock;
15   public:
16       StrategizedLocking(Lock& l): lock(l){
17           lock.lock();
18       }
19       ~StrategizedLocking(){
20           lock.unlock();
21       }
22   };
23
24   struct NullObjectMutex{
25       void lock(){}
26       void unlock(){}
27   };
28
29   class NoLock : public Lock {
30       void lock() const override {
31           std::cout << "NoLock::lock: " << '\n';
32           nullObjectMutex.lock();
33       }
34       void unlock() const override {
35           std::cout << "NoLock::unlock: " << '\n';
36            nullObjectMutex.unlock();
37       }
38       mutable NullObjectMutex nullObjectMutex;
39   };
40
41   class ExclusiveLock : public Lock {
42       void lock() const override {
43           std::cout << "    ExclusiveLock::lock: " << '\n';
44           mutex.lock();
```

```
45          }
46          void unlock() const override {
47              std::cout << "     ExclusiveLock::unlock: " << '\n';
48              mutex.unlock();
49          }
50          mutable std::mutex mutex;
51      };
52
53      class SharedLock : public Lock {
54          void lock() const override {
55              std::cout << "        SharedLock::lock_shared: " << '\n';
56              sharedMutex.lock_shared();
57          }
58          void unlock() const override {
59              std::cout << "        SharedLock::unlock_shared: " << '\n';
60              sharedMutex.unlock_shared();
61          }
62          mutable std::shared_mutex sharedMutex;
63      };
64
65      int main() {
66
67          std::cout << '\n';
68
69          NoLock noLock;
70          StrategizedLocking stratLock1{noLock};
71
72          {
73              ExclusiveLock exLock;
74              StrategizedLocking stratLock2{exLock};
75              {
76                  SharedLock sharLock;
77                  StrategizedLocking startLock3{sharLock};
78              }
79          }
80
81          std::cout << '\n';
82
83      }
```

The class `StrategizedLocking` has a lock (line 14). `StrategizedLocking` models scoped locking and, therefore, locks the mutex in the constructor (line16) and unlocks it in the destructor (line 19). `Lock` (lines 7 - 11) is an abstract class and defines all derived classes' interface. These are the classes `NoLock` (line 29), `ExclusiveLock` (line 41), and `SharedLock` (line 53). `SharedLock` invokes `lock_shared`

(line 56) and `unlock_shared` on its `std::shared_mutex`. Each of these locks holds one of the mutexes `NullObjectMutex` (line 38), `std::mutex` (line 50), or `std::shared_mutex` (line 62). `NullObjectMutex` is a *no-op* placeholder. The mutexes are declared as `mutable`. Therefore, they are usable in constant member functions such as `lock` and `unlock`.

Null Object

The class `NullObjectMutex` is an example of a Null Object Pattern[16], which is often quite helpful. It consists of empty member functions, and it is just a placeholder so that the optimizer can delete it.

9.2.2.2.2 Compile-Time Polymorphism

The template-based implementation is quite similar to the object-oriented-based implementation.

Strategized Locking via Compile-Time Polymorphism

```cpp
1    // strategizedLockingCompileTime.cpp
2
3    #include <iostream>
4    #include <mutex>
5    #include <shared_mutex>
6
7
8    template <typename Lock>
9    class StrategizedLocking {
10       Lock& lock;
11   public:
12       StrategizedLocking(Lock& l): lock(l){
13           lock.lock();
14       }
15       ~StrategizedLocking(){
16           lock.unlock();
17       }
18   };
19
20   struct NullObjectMutex {
21       void lock(){}
22       void unlock(){}
23   };
24
25   class NoLock{
```

---

[16]https://en.wikipedia.org/wiki/Null_object_pattern

```
26    public:
27        void lock() const {
28            std::cout << "NoLock::lock: " << '\n';
29            nullObjectMutex.lock();
30        }
31        void unlock() const {
32            std::cout << "NoLock::unlock: " << '\n';
33             nullObjectMutex.lock();
34        }
35        mutable NullObjectMutex nullObjectMutex;
36    };
37
38    class ExclusiveLock {
39    public:
40        void lock() const {
41            std::cout << "    ExclusiveLock::lock: " << '\n';
42            mutex.lock();
43        }
44        void unlock() const {
45            std::cout << "    ExclusiveLock::unlock: " << '\n';
46            mutex.unlock();
47        }
48        mutable std::mutex mutex;
49    };
50
51    class SharedLock {
52    public:
53        void lock() const {
54            std::cout << "        SharedLock::lock_shared: " << '\n';
55            sharedMutex.lock_shared();
56        }
57        void unlock() const {
58            std::cout << "        SharedLock::unlock_shared: " << '\n';
59            sharedMutex.unlock_shared();
60        }
61        mutable std::shared_mutex sharedMutex;
62    };
63
64    int main() {
65
66        std::cout << '\n';
67
68        NoLock noLock;
69        StrategizedLocking<NoLock> stratLock1{noLock};
70
```

```
71      {
72          ExclusiveLock exLock;
73          StrategizedLocking<ExclusiveLock> stratLock2{exLock};
74          {
75              SharedLock sharLock;
76              StrategizedLocking<SharedLock> startLock3{sharLock};
77          }
78      }
79
80      std::cout << '\n';
81
82  }
```

The programs `strategizedLockingRuntime.cpp` and `strategizedLockingCompileTime.cpp` produce the same output:

StrategizedLocking

The locks `NoLock` (line 25), `ExclusiveLock` (line 38), and `SharedLock` (line 51) have no abstract base class. The consequence is that `StrategizedLocking` can be instantiated with an object that does not support the right interface. This instantiation would end in a compile-time error. This loophole is closed with C++20. Instead of `template <typename Lock> class StrategizedLocking` you can use the concept `BasicLockable`: `template <BasicLockable Lock> class StrategizedLocking`. This means that all used locks have to support the concept[17] `BasicLockable`. A concept is a named requirement, and

---

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

many concepts are already defined in the C++20 concepts library[18]. The concept `BasicLockable`[19] is only used in the text of the C++20 standard. Consequently, I define and use the concept `BasicLockable` in the following improved implementation of the strategized locking at compile time.

Strategized Locking via Compile-Time Polymorphism using the concept `BasicLockable`

```cpp
1    // strategizedLockingCompileTimeWithConcepts.cpp
2
3    #include <iostream>
4    #include <mutex>
5    #include <shared_mutex>
6
7    template <typename T>
8    concept BasicLockable = requires(T lo) {
9        lo.lock();
10       lo.unlock();
11   };
12
13   template <BasicLockable Lock>
14   class StrategizedLocking {
15       Lock& lock;
16   public:
17       StrategizedLocking(Lock& l): lock(l){
18           lock.lock();
19       }
20       ~StrategizedLocking(){
21           lock.unlock();
22       }
23   };
24
25   struct NullObjectMutex {
26       void lock(){}
27       void unlock(){}
28   };
29
30   class NoLock{
31   public:
32       void lock() const {
33           std::cout << "NoLock::lock: " << '\n';
34           nullObjectMutex.lock();
35       }
36       void unlock() const {
37           std::cout << "NoLock::unlock: " << '\n';
```

---

[18]https://en.cppreference.com/w/cpp/concepts
[19]https://en.cppreference.com/w/cpp/named_req

```
38              nullObjectMutex.lock();
39          }
40          mutable NullObjectMutex nullObjectMutex;
41      };
42
43      class ExclusiveLock {
44      public:
45          void lock() const {
46              std::cout << "    ExclusiveLock::lock: " << '\n';
47              mutex.lock();
48          }
49          void unlock() const {
50              std::cout << "    ExclusiveLock::unlock: " << '\n';
51              mutex.unlock();
52          }
53          mutable std::mutex mutex;
54      };
55
56      class SharedLock {
57      public:
58          void lock() const {
59              std::cout << "        SharedLock::lock_shared: " << '\n';
60              sharedMutex.lock_shared();
61          }
62          void unlock() const {
63              std::cout << "        SharedLock::unlock_shared: " << '\n';
64              sharedMutex.unlock_shared();
65          }
66          mutable std::shared_mutex sharedMutex;
67      };
68
69      int main() {
70
71          std::cout << '\n';
72
73          NoLock noLock;
74          StrategizedLocking<NoLock> stratLock1{noLock};
75
76          {
77              ExclusiveLock exLock;
78              StrategizedLocking<ExclusiveLock> stratLock2{exLock};
79              {
80                  SharedLock sharLock;
81                  StrategizedLocking<SharedLock> startLock3{sharLock};
82              }
```

```
83        }
84
85        std::cout << '\n';
86
87  }
```

Lines (7 - 11) define the concept `BasicLockable`. `BasicLockable` requires that an object `lo` of type `T` that `lo` support the member functions `lock` and `unlock`. The use of the concept is straightforward. Instead of `typename`, I use the concept `BasicLockable` in the template declaration of `StrategizedLocking` (line 13). What happens, when I rename the member function `unlock` (line 49) of the class `ExclusiveLock` into `unlck()`. The compilation fails and the compiler says essentially that the constraints for the class `StrategizedLocking` are not satisfied because the call `lo.unlock()` would be invalid.

You can read more details about concepts in my blogposts on ModernesCpp/concepts[20] or in my C++20[21] book.

9.2.2.3 Further Information

- Design Patterns: Elements of Reusable Object-Oriented Software[22]
- Strategy Pattern[23]
- Null Object Pattern[24]
- Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects[25]

9.2.3 Thread-Safe Interface

The thread-safe interface fits very well when the critical sections are just objects. The naive idea to protect all member functions with a lock causes, in the best case, a performance issue and, in the worst case, a deadlock. The small pseudocode makes my point clear.

---

[20]https://www.modernescpp.com/index.php/tag/concepts
[21]https://leanpub.com/c20
[22]https://en.wikipedia.org/wiki/Design_Patterns
[23]https://en.wikipedia.org/wiki/Strategy_pattern
[24]https://en.wikipedia.org/wiki/Null_object_pattern
[25]https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/

```
struct Critical{
    void memberFunction1(){
        lock(mut);
        memberFunction2();
        ...
    }
    void memberFunction2(){
        lock(mut);
        ...
    }
    mutex mut;
};

Critical crit;
crit.memberFunction1();
```

Calling `crit.memberFunction1` causes the mutex `mut` being locked twice. For simplicity reasons the `lock` is a scoped lock. Here are the two issues:

1. when `lock` is a recursive lock, the second `lock(mut)` in `memberFunction2` is redundant.

2. when `lock` is a non-recursive lock, the second `lock(mut)` in `memberFunction2` leads to undefined behavior. Most of the time, you get a deadlock.

The thread-safe interface overcomes both issues. Here is the straightforward idea:

- All interface member functions (`public`) should use a lock.

- All implementation member functions (`protected` and `private`) must not use a lock.

- The interface member functions call only `protected` or `private` member functions but no `public` member functions.

The `threadSafeInterface.cpp` program shows its usage.

The Thread-Safe Interface

```
1    // threadSafeInterface.cpp
2
3    #include <iostream>
4    #include <mutex>
5    #include <thread>
6
7    class Critical{
8
9    public:
10       void interface1() const {
11           std::lock_guard<std::mutex> lockGuard(mut);
12           implementation1();
```

```cpp
13          }
14      void interface2(){
15          std::lock_guard<std::mutex> lockGuard(mut);
16          implementation2();
17          implementation3();
18          implementation1();
19      }
20  private:
21      void implementation1() const {
22          std::cout << "implementation1: "
23                      << std::this_thread::get_id() << '\n';
24      }
25      void implementation2(){
26          std::cout << "    implementation2: "
27                      << std::this_thread::get_id() << '\n';
28      }
29      void implementation3(){
30          std::cout << "         implementation3: "
31                      << std::this_thread::get_id() << '\n';
32      }
33

34
35  mutable std::mutex mut;
36
37  };
38
39  int main(){
40
41      std::cout << '\n';
42
43      std::thread t1([]{
44          const Critical crit;
45          crit.interface1();
46      });
47
48      std::thread t2([]{
49          Critical crit;
50          crit.interface2();
51          crit.interface1();
52      });
53
54      Critical crit;
55      crit.interface1();
56      crit.interface2();
57
```

```
58      t1.join();
59      t2.join();
60
61      std::cout << '\n';
62
63  }
```

Three threads, including the main thread, use instances of `Critical`. Thanks to the thread-safe interface, all calls to the public API are synchronized. The mutex `mut` in line 35 is mutable and can be used in the constant member function `interface1`.

The advantages of the thread-safe interface are threefold.

1. A recursive call of a mutex is not possible. Recursive calls on a non-recursive mutex are undefined behavior in C++ and usually end in a deadlock.

2. The program uses minimal locking and, therefore, minimal synchronization. Using just a `std::recursive_mutex` in each public or private member function of the class `Critical` would end in more expensive synchronizations.

3. From the user perspective, `Critical` is straightforward to use because synchronization is only an implementation detail.

Each interface member function delegates its work to the corresponding implementation member function. The indirection overhead is a typical disadvantage of the thread-safe interface pattern.

The output of the program shows the interleaving of the three threads.

Thread-Safe Interface

Although the thread-safe interface seems easy to implement, there are two grave perils you have to keep in mind.

9.2.3.1 Perils

Using a static member in your class or having virtual interfaces requires special care.

9.2.3.1.1 Static members

When your class has a static member that is not constant, you have to synchronize all member function calls on the class instances.

The Thread-Safe Interface with a static member

```
1   class Critical{
2
3   public:
4       void interface1() const {
5           std::lock_guard<std::mutex> lockGuard(mut);
6           implementation1();
7       }
8       void interface2(){
9           std::lock_guard<std::mutex> lockGuard(mut);
10          implementation2();
```

```
11              implementation3();
12              implementation1();
13          }
14
15      private:
16          void implementation1() const {
17              std::cout << "implementation1: "
18                            << std::this_thread::get_id() << '\n';
19              ++called;
20          }
21          void implementation2(){
22              std::cout << "    implementation2: "
23                            << std::this_thread::get_id() << '\n';
24              ++called;
25          }
26          void implementation3(){
27              std::cout << "        implementation3: "
28                            << std::this_thread::get_id() << '\n';
29              ++called;
30          }
31
32          inline static int called{0};
33          inline static std::mutex mut;
34
35      };
```

The class `Critical` now has the static member `called` (line 32) to count how often the implementation functions were called. All instances of `Critical` use the same static member `called` and have, therefore, to be synchronized. The critical section contains, in this case, all instances of `Critical`.

Inline static data members

Since C++17, static data members can be declared inline. An inline static data member can be defined and initialized in the class definition.

```
struct X
{
    inline static int n = 1;
};
```

9.2.3.1.2 Virtuality

When you override a virtual interface function, the overriding function should have a lock even if the function is private.
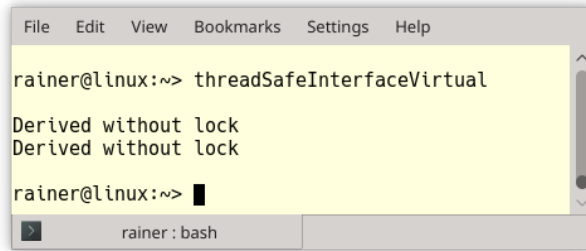
The Thread-Safe Interface with a virtual member function

```cpp
// threadSafeInterfaceVirtual.cpp

#include <iostream>
#include <mutex>
#include <thread>

class Base{

public:
    virtual void interface() {
        std::lock_guard<std::mutex> lockGuard(mut);
        std::cout << "Base with lock" << '\n';
    }
private:
    std::mutex mut;
};

class Derived: public Base{

    void interface() override {
        std::cout << "Derived without lock" << '\n';
    }

};

int main(){

    std::cout << '\n';

    Base* base1 = new Derived;
    base1->interface();

    Derived der;
    Base& base2 = der;
    base2.interface();

    std::cout << '\n';

}
```

In the calls `base1->interface` and `base2.interface` the static type of `base1` and `base2` is `Base` and, therefore, `interface` is a public member function. Because the interface member function is virtual, the call happens at run time using the dynamic type `Derived`. At last, the private member function `interface` of the class `Derived` is invoked.

```
File   Edit   View   Bookmarks   Settings   Help

rainer@linux:~> threadSafeInterfaceVirtual

Derived without lock
Derived without lock

rainer@linux:~> █

                rainer : bash
```

Thread-Safe Interface with virtual member function

There are two typical ways to overcome this issue.

1. Make the member function `interface` a non-virtual member function. This technique is called NVI (Non-Virtual Interfac)[26]. The non-virtual member functions guarantees that the `interface` function of the base class `Base` is used. Additionally, overriding the `interface` function using `override` causes a compile-time error because there is nothing to override.

2. Declare the member function `interface` as final: `virtual void interface() final`. Thanks to `final`, overriding an as `final` declared virtual member function causes a compile-time error.

Although I presented two ways to overcome this issue, I strongly suggest you prefer the NVI idiom. Use early binding if you don't need late binding (virtuality).

9.2.3.2 Further Information

- Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects[27]

9.2.4 Guarded Suspension

The guarded suspension basic variant combines a lock and a precondition that must be satisfied. If the precondition is not fulfilled, that calling thread puts itself to sleep. To avoid a race condition which may result in a data race or a deadlock, the checking thread uses a lock.

Various variants exist:

- The waiting thread can passively be notified about the state change or actively ask for the state change. In short, I call this push versus pull principle.

---

[26]https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Non-Virtual_Interface
[27]https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/

- The waiting can be done with or without a time boundary.

- The notification can be sent to one or all waiting threads.

I have already discussed in depth each of the three variants. What is, therefore, left for this section is to bundle all information in one place. For the details, please follow the links in this subsection to the guarded suspension.

9.2.4.1 Push versus Pull Principle

Let me start with the push principle.

9.2.4.1.1 Push Principle

Most of the time, you use a condition variable or a future/promise pair to synchronize threads. The condition variable or the promise send the notification to the waiting thread. A promise has no `notify_one` or `notify_all` member function. Instead, a valueless `set_value` call is typically used to signal a notification. The following program snippets show the thread sending the notification and the waiting thread.

- Condition variables

```
void waitingForWork(){
    std::cout << "Worker: Waiting for work." << '\n';
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
    doTheWork();
    std::cout << "Work done." << '\n';
}

void setDataReady(){
    {
      std::lock_guard<std::mutex> lck(mutex_);
      dataReady = true;
    }
    std::cout << "Sender: Data is ready."  << '\n';
    condVar.notify_one();
}
```

- Future/promise pair

```cpp
void waitingForWork(std::future<void>&& fut){

    std::cout << "Worker: Waiting for work." << '\n';
    fut.wait();
    doTheWork();
    std::cout << "Work done." << '\n';

}

void setDataReady(std::promise<void>&& prom){

    std::cout << "Sender: Data is ready."  << '\n';
    prom.set_value();

}
```

9.2.4.1.2 Pull Principle

Instead of passively waiting for the state change, you can actively ask for it. This pull principle is not natively supported in C++ but can be, for example, implemented with atomics.

```cpp
std::vector<int> mySharedWork;
std::atomic<bool> dataReady(false);

void waitingForWork(){
    std::cout << "Waiting " << '\n';
    while (!dataReady.load()){
        std::this_thread::sleep_for(std::chrono::milliseconds(5));
    }
    mySharedWork[1] = 2;
    std::cout << "Work done " << '\n';
}

void setDataReady(){
    mySharedWork = {1, 0, 3};
    dataReady = true;
    std::cout << "Data prepared" << '\n';
}
```

9.2.4.2 Waiting with and without time boundary

A condition variable and a future have three member functions for waiting: `wait`, `wait_for`, and `wait_until`. The `wait_for` variant requires a time duration and the `wait_until` variant a time point. I skip

waiting without time boundaries (`wait`) in this section not to bore you. The last section to the push principle already provides two examples.

The consumer thread in various waiting strategies waits for the time duration `steady_clock::now() +` `dur`. If the promise is ready it asks for the value; if not, it just displays its id: `this_thread::get_it()`.

```cpp
void producer(promise<int>&& prom){
    cout << "PRODUCING THE VALUE 2011\n\n";
    this_thread::sleep_for(seconds(5));
    prom.set_value(2011);
}

void consumer(shared_future<int> fut,
              steady_clock::duration dur){
    const auto start = steady_clock::now();
    future_status status= fut.wait_until(steady_clock::now() + dur);
    if ( status == future_status::ready ){
        lock_guard<mutex> lockCout(coutMutex);
        cout << this_thread::get_id() << " ready => Result: " << fut.get()
             << '\n';
    }
    else{
        lock_guard<mutex> lockCout(coutMutex);
        cout << this_thread::get_id() << " stopped waiting." << '\n';
    }
    const auto end= steady_clock::now();
    lock_guard<mutex> lockCout(coutMutex);
    cout << this_thread::get_id() << " waiting time: "
         << getDifference(start,end) << " ms" << '\n';
}
```

9.2.4.3 Notify one or all waiting threads

`notify_one` awakes one of the waiting threads, `notify_all` awakes all of the waiting threads. With `notify_one`, you have no guarantee which one will be awakened. The other threads do stay in the wait state. This could not happen with a `std::future`, because there is a one-to-one association between the future and the promise. If you want to simulate a one-to-many association, use a `std::shared_future` instead of a `std::future` because a `std::shared_future` can be copied.

The following program shows a simple workflow with one-to-one and one-to-many associations between promises and futures.

A boss/worker workflow

```cpp
// bossWorker.cpp

#include <future>
#include <chrono>
#include <iostream>
#include <random>
#include <string>
#include <thread>
#include <utility>

int getRandomTime(int start, int end){

  std::random_device seed;
  std::mt19937 engine(seed());
  std::uniform_int_distribution<int> dist(start,end);

  return dist(engine);
};

class Worker{
public:
  explicit Worker(const std::string& n):name(n){};

  void operator() (std::promise<void>&& preparedWork,
                   std::shared_future<void> boss2Worker){

    // prepare the work and notfiy the boss
    int prepareTime= getRandomTime(500, 2000);
    std::this_thread::sleep_for(std::chrono::milliseconds(prepareTime));
    preparedWork.set_value();
    std::cout << name << ": " << "Work prepared after "
              << prepareTime << " milliseconds." << '\n';

    // still waiting for permission to start working
    boss2Worker.wait();
  }
private:
  std::string name;
};

int main(){

  std::cout << '\n';
```
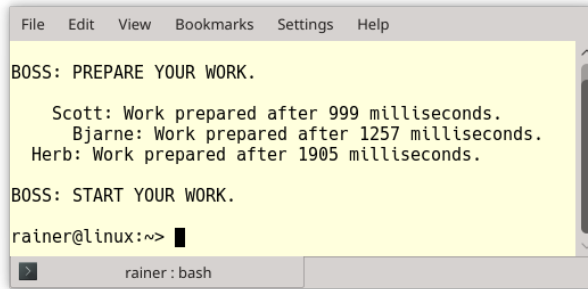
```
45     // define the std::promise => Instruction from the boss
46     std::promise<void> startWorkPromise;
47
48     // get the std::shared_future's from the std::promise
49     std::shared_future<void> startWorkFuture= startWorkPromise.get_future();
50
51     std::promise<void> herbPrepared;
52     std::future<void> waitForHerb = herbPrepared.get_future();
53     Worker herb("  Herb");
54     std::thread herbWork(herb, std::move(herbPrepared), startWorkFuture);
55
56     std::promise<void> scottPrepared;
57     std::future<void> waitForScott = scottPrepared.get_future();
58     Worker scott("    Scott");
59     std::thread scottWork(scott, std::move(scottPrepared), startWorkFuture);
60
61     std::promise<void> bjarnePrepared;
62     std::future<void> waitForBjarne = bjarnePrepared.get_future();
63     Worker bjarne("      Bjarne");
64     std::thread bjarneWork(bjarne, std::move(bjarnePrepared), startWorkFuture);
65
66     std::cout << "BOSS: PREPARE YOUR WORK.\n " << '\n';
67
68     // waiting for the worker
69     waitForHerb.wait(), waitForScott.wait(), waitForBjarne.wait();
70
71     // notify the workers that they should begin to work
72     std::cout << "\nBOSS: START YOUR WORK. \n" << '\n';
73     startWorkPromise.set_value();
74
75     herbWork.join();
76     scottWork.join();
77     bjarneWork.join();
78
79 }
```

The key idea of the program is that the boss (main-thread) has three workers: herb (line 53), scott (line 58), and bjarne (line 63). A thread represents each worker. In line 64, the boss waits until all workers are done with their work package preparation. This means each worker sends, after an arbitrary time, the notification to the boss that he is done. The worker-to-the-boss notification is a one-to-one relation (line 30) because it goes to a std::future. In contrast, the instruction to start the work is a one-to-many notification (line 73) from the boss to its workers. For this one-to-many notification, a std::shared_future is necessary.

A Boss/Worker Workflow

9.2.4.4 Further Information

- Concurrent Programming in Java: Design Principles and Patterns (Doug Lea)[28]. The difference between C++ and Java programming languages should not be an issue because the essence of concurrent programming is much deeper than syntax of a particular language.

Distilled Information

- A necessary prerequisite for a data race is shared mutable state. Synchronization patterns boil down to two concerns: dealing with sharing and dealing with mutation.
- The three patterns copied value, thread-specific storage, and future prevent the sharing of date.
- Thanks to the patterns scoped locking, strategized locking, thread-safe interface, or guarded suspension, you can safely share mutable data between threads.

---

[28]http://gee.cs.oswego.edu/dl/cpj/

10. Concurrent Architecture
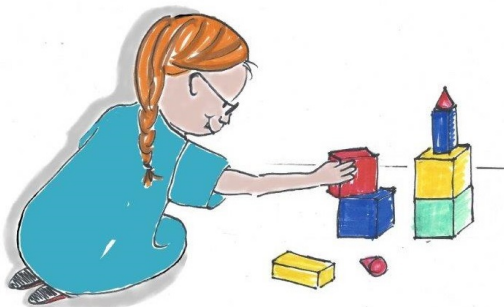
Cippi designs a building

The patterns presented in this chapter are classics. The presented patterns are very well explained in the invaluable book Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects[1]. My goal for this chapter is to give a concise overview of the Active Object, the Monitor Object. Additionally, I write about the Half-Sync/Half-Async pattern and the two strongly related patterns: Reactor, and Proactor. As in the last chapter on Synchronization Patterns, I'm wearing C++ glasses. Before I dive into the patterns, here are the patterns from a birds-eye perspective.

- The **Active Object** design pattern decouples member function execution from member function invocation for objects that each reside in their own thread of control. The goal is to introduce concurrency by using asynchronous member function invocation and a scheduler for handling requests. Wikipedia: Active object[2]

- The **Monitor Object** design pattern synchronizes concurrent member function execution to ensure that only one member function at a time runs within an object. It also allows object's member functions to schedule their execution sequences cooperatively. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects[3]

Both patterns synchronize and schedule member function invocation. The main difference is that the Active Object executes its member function in a different thread but the monitor in the same thread as the client. In contrast to the Active Object and the Monitor Object, which have a subsystem focus and are, therefore, typically called design patterns, the Half-Sync/Half-Async, the Reactor, and the Proactor pattern have system perspective and are consequently called architectural patterns.

---

[1]https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/
[2]https://en.wikipedia.org/wiki/Active_object
[3]https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/

- The **Half-Sync/Half-Async** architectural pattern decouples asynchronous and synchronous service processing in concurrent systems to simplify programming without unduly reducing performance. The pattern introduces two intercommunicating layers, one for asynchronous and one for synchronous service processing. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects[4]

- The **Reactor** pattern is an event-driven framework to demultiplex and dispatch service requests concurrently onto various service providers. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects[5]

- The **Proactor** pattern enables event-driven applications to demultiplex and dispatch service requests triggered by the completion of an asynchronous operation. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects[6]

10.1 Active Object

The Active Object design pattern decouples method invocation from method execution. The method invocation is performed on the client thread, but the method execution on the Active Object. The Active Object has its own thread and a list of method request objects (short request) to be executed. The client's method invocation enqueues the requests on the Active Object's list. The requests are dispatched on the servant. The Active Object pattern is also known as Concurrent Object.

10.1.1 Challenges

When many threads access a shared object synchronized, the following challenges have to be solved.

1. A thread invoking a processing-intensive member function should not block the other threads invoking the same object for too long.

2. It should be easy to synchronized access a shared object.

3. The concurrency characteristics of the executed requests should be adaptable to the concrete hardware and software.

10.1.2 Solution

The client's method invocation goes to a proxy, which represents the interface of the Active Object. The servant implements these member functions and runs in the Active Object's thread. At run time, the proxy transforms the invocation into a method invocation on the servant. This request is enqueued in an activation list by the scheduler. A scheduler's event loop runs in the same thread as the servant, dequeues the requests from the activation list, removes them, and dispatches them on the servant. The client obtains the result of the method invocation via a future from the proxy.

---

[4]https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/
[5]https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/
[6]https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/

10.1.3 Components

The Active Object pattern consists of six components:

1. The **proxy** provides an interface for the accessible member functions on the Active Object. The proxy triggers the construction of a request which goes into the activation list. The proxy runs in the client thread.

2. The **method request** class defines the interface for the method executing on an Active Object. This interface also contains guard methods, indicating if the job is ready to run. The request includes all context information to be executed later.

3. The **activation list** maintains the pending requests. The activation list decouples the client's thread from the Active Object thread. The proxy inserts the request object, and the scheduler removes them. Consequently, the access onto the activation list must be serialized.

4. The **scheduler** runs in the thread of the Active Object and decides which request from the activation list is executed next. The scheduler evaluates the guards of the request to determine if the request could run.

5. The **servant** implements the Active Object and runs in the active object's thread. The servant implements the interface of the method request, and the scheduler invokes its member functions.

6. The **future** is created by the proxy and is only necessary if the request returns a result. Therefore, the client receives the future and can obtain the result of the method invocation on the Active Object. The client can wait for the outcome or poll for it.

10.1.4 Dynamic Behavior

The dynamic behavior of the Active Object consists on three phases:

1. **Request construction and scheduling**: The client invokes a method on the proxy. The proxy creates a request and passes it to the scheduler. The scheduler enqueues the request on the activation list. Additionally, the proxy returns a future to the client if the request returns a result.

2. **Member function execution**: The scheduler determines which request becomes runnable by evaluating the guard method of the request. It removes the request from the activation list and dispatches it to the servant.

3. **Completion**: When the request returns something, it is stored in the future. The client can ask for the result. When the client has the result, the request and the future can be deleted.

The following picture shows the sequence of messages.
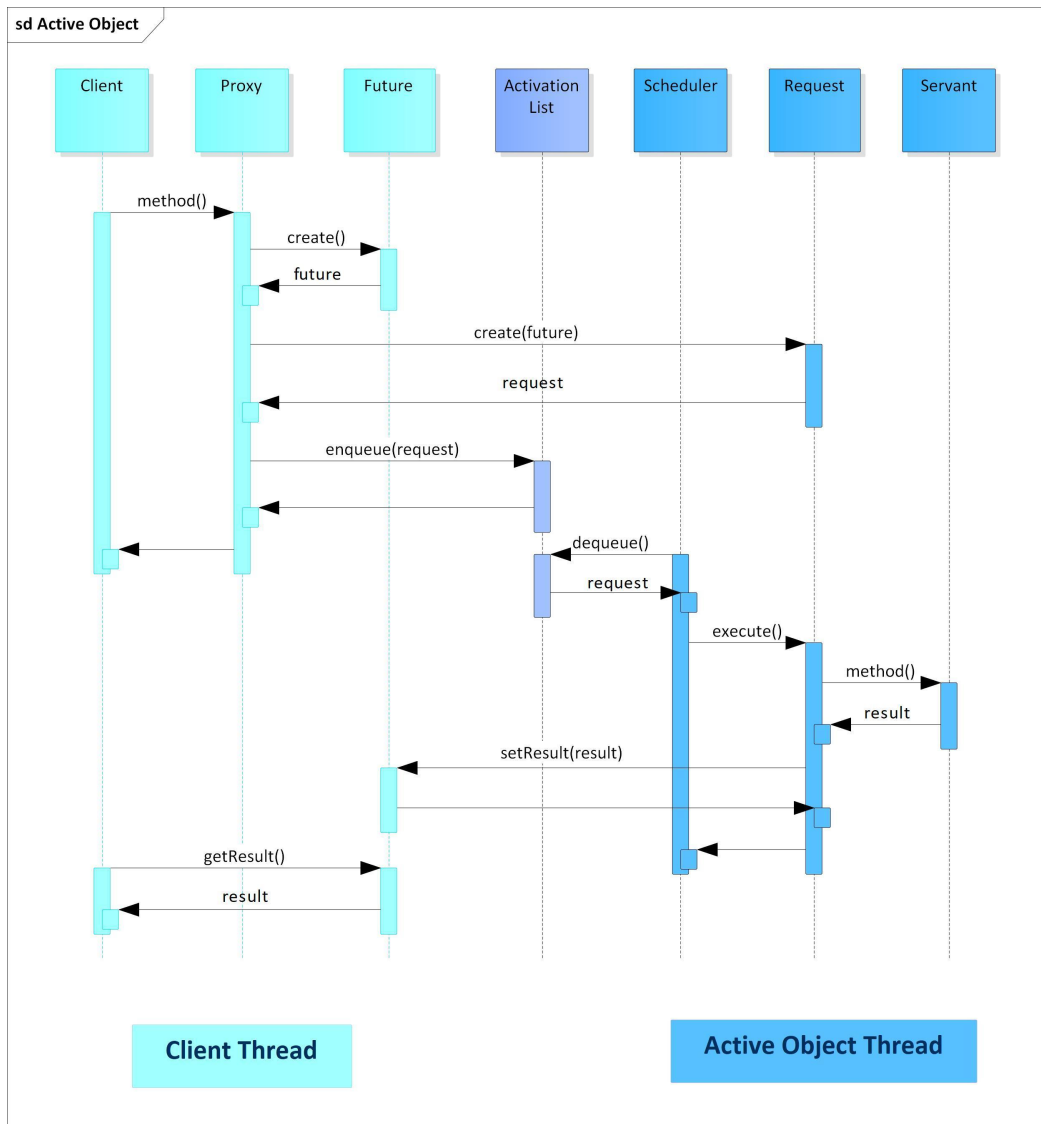
Active Object

Proxy

The proxy design pattern is one of the classics from the book Design Patterns: Elements of Reusable Object-Oriented Software[7]. A proxy is an object which stands for something else. A typical proxy could be a remote proxy CORBA[8], a security proxy, a virtual proxy which is created on-demand, or a smart pointer such as `std::shared_ptr`[9]. Each proxy adds additional functionality to the object it represents. A remote proxy stands for a remote object and gives the client the illusion of a local object. A security proxy turns an insecure connection into a secure connection by encrypting and decrypting data. A virtual proxy encapsulates the creation of a heavy-weight object in a lazy fashion, and a smart pointer manages the lifetime of the underlying memory.
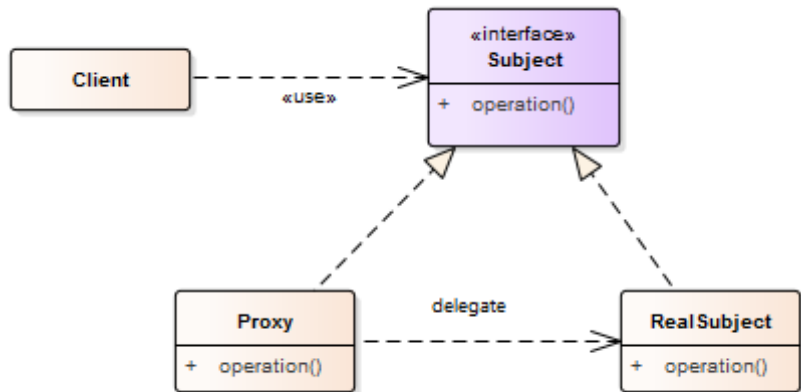
Proxy

- The **Proxy** has the same interface, such as the RealSubject, manages the reference, and often the subject's lifetime.
- The **Subject** has the same interface such as the proxy and the RealSubject.
- The **RealSubject** provides the functionality.

The Wikipedia article about the proxy pattern[10] gives you more details.

---

[7]https://en.wikipedia.org/wiki/Design_Patterns

[8]https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture

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

[10]https://en.wikipedia.org/wiki/Proxy_pattern

10.1.5 Advantages and Disadvantages

Before I present a minimal implementation of the Active Object pattern, here are its advantages and disadvantages.

- Advantages:
    - The synchronization is only necessary on the Active Object's thread but not on the client's threads.
    - Clear separation between the client (user) and the server (implementer). The synchronization challenges are on the implementer's side.
    - Enhanced throughput of the system because of the asynchronous execution of the requests. Invoking processing-intensive member functions do not block the entire system.
    - The scheduler can implement various strategies to execute the pending requests. If so, the jobs can be executed in a different order they are enqueued.

- Disadvantages:
    - If the requests are too fine-grained, the Active Object pattern's performance overhead such as the proxy, the activation list, and the scheduler may be excessive.
    - Due to the scheduler's scheduling strategy and the operating system's scheduling, debugging the Active Object pattern is often quite tricky. This holds, in particular, if the jobs are executed in a different order, they are enqueued.

10.1.6 Implementation

The following example presents a simplified implementation of the Active Object pattern. In particular, I don't define an interface for the method requests on the active object, which the proxy and the servant should implement. Further, the scheduler executes the next job when asked for it, and the `run` member function of the `Active Object` creates the threads.

The involved types `future<vector<future<pair<bool, int>>>>` are often quite verbose. To improve the readability, I heavily applied using declarations (lines 16 - 37).

Active Object

```cpp
// activeObject.cpp

#include <algorithm>
#include <deque>
#include <functional>
#include <future>
#include <iostream>
#include <memory>
#include <mutex>
#include <numeric>
```

```cpp
11   #include <random>
12   #include <thread>
13   #include <utility>
14   #include <vector>
15
16   using std::async;
17   using std::boolalpha;
18   using std::cout;
19   using std::deque;
20   using std::distance;
21   using std::for_each;
22   using std::find_if;
23   using std::future;
24   using std::lock_guard;
25   using std::make_move_iterator;
26   using std::make_pair;
27   using std::move;
28   using std::mt19937;
29   using std::mutex;
30   using std::packaged_task;
31   using std::pair;
32   using std::random_device;
33   using std::sort;
34   using std::jthread;
35   using std::uniform_int_distribution;
36   using std::vector;
37
38   class IsPrime {
39   public:
40       pair<bool, int> operator()(int i) {
41           for (int j = 2; j * j <= i; ++j){
42               if (i % j == 0) return make_pair(false, i);
43           }
44           return make_pair(true, i);
45       }
46   };
47
48   class ActiveObject {
49   public:
50
51       future<pair<bool, int>> enqueueTask(int i) {
52           IsPrime isPrime;
53           packaged_task<pair<bool, int>(int)> newJob(isPrime);
54           auto isPrimeFuture = newJob.get_future();
55           auto pair = make_pair(move(newJob), i);
```

```
56              {
57                  lock_guard<mutex> lockGuard(activationListMutex);
58                  activationList.push_back(move(pair));
59              }
60              return isPrimeFuture;
61          }
62
63          void run() {
64              vector<jthread> allServants;
65              // only one thread because the activation list is the bottleneck
66              for (int n = 0; n < 1; ++n) {
67                  allServants.emplace_back([this] {
68                      while ( !runNextTask() );
69                  });
70              }
71          }
72
73      private:
74
75          bool runNextTask() {
76              lock_guard<mutex> lockGuard(activationListMutex);
77              auto empty = activationList.empty();
78              if (!empty) {
79                  auto myTask= std::move(activationList.front());
80                  activationList.pop_front();
81                  myTask.first(myTask.second);
82              }
83              return empty;
84          }
85
86          deque<pair<packaged_task<pair<bool, int>(int)>, int >> activationList;
87          mutex activationListMutex;
88      };
89
90      vector<int> getRandNumbers(int number) {
91          random_device seed;
92          mt19937 engine(seed());
93          uniform_int_distribution<> dist(1000000, 1000000000);
94          vector<int> numbers;
95          for (long long i = 0 ; i < number; ++i) numbers.push_back(dist(engine));
96          return numbers;
97      }
98
99      future<vector<future<pair<bool, int>>>> getFutures(ActiveObject& activeObject,
100                                                          int numberPrimes) {
```

```
101        return async([&activeObject, numberPrimes] {
102            vector<future<pair<bool, int>>> futures;
103            auto randNumbers = getRandNumbers(numberPrimes);
104            for (auto numb: randNumbers){
105                futures.push_back(activeObject.enqueueTask(numb));
106            }
107            return futures;
108        });
109    }
110
111
112    int main() {
113
114        cout << boolalpha << '\n';
115
116         ActiveObject activeObject;
117
118        // a few clients enqueue work concurrently
119        auto client1 = getFutures(activeObject, 1998);
120        auto client2 = getFutures(activeObject, 2003);
121        auto client3 = getFutures(activeObject, 2011);
122        auto client4 = getFutures(activeObject, 2014);
123        auto client5 = getFutures(activeObject, 2017);
124
125        // give me the futures
126        auto futures = client1.get();
127        auto futures2 = client2.get();
128        auto futures3 = client3.get();
129        auto futures4 = client4.get();
130        auto futures5 = client5.get();
131
132        // put all futures together
133        futures.insert(futures.end(),make_move_iterator(futures2.begin()),
134                                     make_move_iterator(futures2.end()));
135
136        futures.insert(futures.end(),make_move_iterator(futures3.begin()),
137                                     make_move_iterator(futures3.end()));
138
139        futures.insert(futures.end(),make_move_iterator(futures4.begin()),
140                                     make_move_iterator(futures4.end()));
141
142        futures.insert(futures.end(),make_move_iterator(futures5.begin()),
143                                     make_move_iterator(futures5.end()));
144
145        // run the promises
```

```
146        activeObject.run();
147
148        // get the results from the futures
149        vector<pair<bool, int>> futResults;
150        futResults.reserve(futures.size());
151        for (auto& fut: futures) futResults.push_back(fut.get());
152
153        sort(futResults.begin(), futResults.end());
154
155        // separate the primes from the non-primes
156        auto prIt = find_if(futResults.begin(), futResults.end(),
157                            [](pair<bool, int> pa){ return pa.first == true; });
158
159        cout << "Number primes: " << distance(prIt, futResults.end()) << '\n';
160        cout << "Primes:" << '\n';
161        for_each(prIt, futResults.end(), [](auto p){ cout << p.second << " ";} );
162
163        cout << "\n\n";
164
165        cout << "Number no primes: " << distance(futResults.begin(), prIt) << '\n';
166        cout << "No primes:" << '\n';
167        for_each(futResults.begin(), prIt, [](auto p){ cout << p.second << " ";} );
168
169        cout << '\n';
170
171    }
```

First of all, the example's general idea is that clients can enqueue jobs concurrently on the activation list. The job of the servant is to determine which numbers are prime, and the activation list is part of the Active Object. The Active Object runs the jobs enqueued in the activation list on a separate thread, and the clients can ask for the results.

Here are the details. The five clients enqueue the work (lines 119 - 123) on the activeObject via the getFutures function. getFutures takes the activeObject and a number numberPrimes. numberPrimes random numbers are generated (line 103) between 1000000 and 1000000000 (line 93) and pushed on the return value: vector<future<pair<bool, int>>. future<pair<bool, int> holds a bool and an int. The bool indicates if the int is a prime. Let's have a closer look at line 105: futures.push_back(activeObject.enqueueTask(numb)). This call triggers that a new job is enqueued on the activation list (line 58). All calls on the activation list have to be protected. The activation list is a deque of promises (line 86): deque<pair<packaged_task<pair<bool, int>(int)>, int >>. Each promise performs the function object IsPrime (lines 38 - 46) when called. The return value is a pair of a bool and an int. The bool indicates if the number int is prime.

Now, the work packages are prepared. Let's start the calculation. All clients return in lines 126 - 130 their handles to the associated futures. Putting all futures together (lines 133 - 143) makes my job easier.

The call `activeObject.run()` in line 146 starts the execution. The member function `run` (line 63 - 71) creates the threads that are going to execute the member function `runNextTask` (line 67). `runNextTask` (lines 75 - 84) determines if the deque is not empty (line 78), and creates the new task. By calling `futResults.push_back(fut.get())` (line 151) on each future, all results are requested and pushed on `futResults`. Line 153 sorts the vector of pairs: `vector<pair<bool, int>>`. The remaining lines present the calculation. The iterator `prIt` in line 156 holds the first iterator to a pair that has a prime number. The screenshot shows the number of primes `distance(prIt, futResults.end())` (line 159) and the primes (line 113). Only the first non-primes are displayed.

Active Object

- [Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects](https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/)[11]
- [Prefer Using Active Object instead of Naked Thread (Herb Sutter)](http://www.drdobbs.com/parallel/prefer-using-active-objects-instead-of-n/225700095)[12]
- [Active Object implementation in C++11](https://github.com/lightful/syscpp/)[13]

---

[11]https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/
[12]http://www.drdobbs.com/parallel/prefer-using-active-objects-instead-of-n/225700095
[13]https://github.com/lightful/syscpp/

10.2 Monitor Object

The Monitor Object design pattern synchronizes concurrent member function execution to ensure that only one member function at a time runs within an object. It also allows an object's member functions to schedule their execution sequences cooperatively. This pattern is also known as Thread-Safe Passive Object.

10.2.1 Challenges

If many threads access a shared object concurrently, the following challenges exist.

1. Due to the concurrent access, the shared object must be protected from non-synchronized read and write operations to avoid data races.

2. The necessary synchronization should be part of the implementation and not part of the interface.

3. When a thread is done with the shared object, a notification should be triggered so that the next thread can use the shared object. This mechanism helps avoid deadlocks and improves the system's overall performance.

4. After the execution of a member function, the invariants of the shared object must hold.

10.2.2 Solution

A client (thread) can access the Monitor Object's synchronized member functions, and due to the monitor lock, only one synchronized member function can run at any given point in time. Each Monitor Object has a monitor condition that notifies the waiting clients.

10.2.3 Components

The Monitor Object consists of four components.
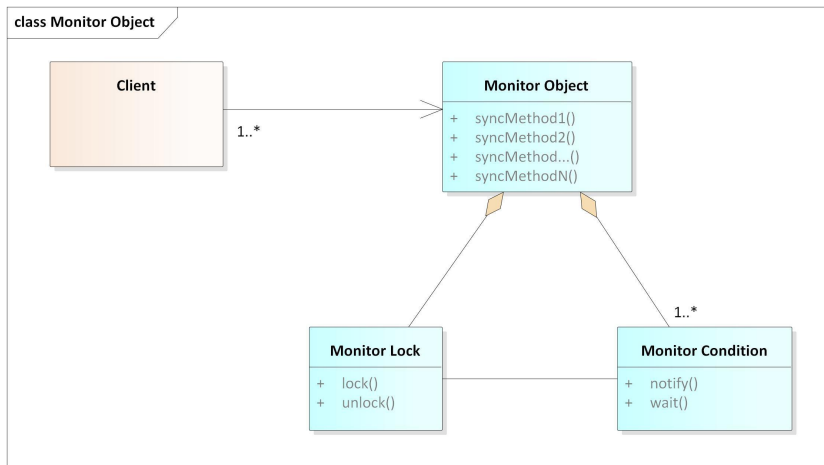
Monitor Object

1. **Monitor Object**: The Monitor Object supports one or more member functions. Each client must access the object through these member functions, and each member function runs in the client's thread.

2. **Synchronized member functions**: The synchronized member functions are the member functions supported by the Monitor Object. Only one member function can execute at any given point in time. The Thread-Safe Interface helps to distinguish between the interface member functions (synchronized member functions) and the implementation member functions of the Monitor Object.

3. **Monitor lock**: Each Monitor Object has one monitor lock, which ensures that at most one client can access the Monitor Object at any given point in time.

4. **Monitor condition**: The monitor condition allows separate threads to schedule their member function invocations on the Monitor Object. When the current client is done with its invocation of the synchronized member functions, the next waiting client is awakened to invoke the Monitor Object's synchronized member functions.

While the monitor lock ensures the synchronized member functions' exclusive access, the monitor condition guarantees minimal waiting for the clients. Essentially, the monitor lock protects from data races and the condition monitor from deadlocks.

10.2.4 Dynamic Behavior

The interaction between the Monitor Object and its components has different phases.

- When a client invokes a synchronized member function on a Monitor Object, it must first lock the global monitor lock. If the client is successful with its locking, it executes the synchronized member function and unlocks the monitor lock at the end. If the client is not successful, the client is blocked.

- When the client is blocked because it cannot progress, it waits until the monitor condition sends a notification. This notification happens when the monitor is unlocked. The notification can be sent only to one or to all of the waiting clients. Typically, waiting means resource-friendly sleeping in contrast to busy-waiting.

- When a client gets the notification to resume, it locks the monitor lock and executes the synchronized member function. At the end of the synchronized member function, the monitor lock is unlocked. The monitor condition sends a notification to signal that the next client can execute its synchronized member function.

10.2.5 Advantages and Disadvantages

What are the advantages and disadvantages of the Monitor Object?

- Advantages:
  - The client is not aware of the implicit synchronization of the Monitor Object, and the synchronization is fully encapsulated in the implementation.
  - The invoked synchronized member functions will be eventually automatically scheduled. The notification/waiting mechanism of the monitor condition behaves as a simple scheduler.

- Disadvantages:
  - It is often quite challenging to change the synchronization mechanism of the synchronization member functions because the functionality and the synchronization are strongly coupled.
  - When a synchronized member function invokes directly or indirectly the same Monitor Object, a deadlock may occur.

The following example defines a `ThreadSafeQueue`.

Monitor Object

```cpp
// monitorObject.cpp

#include <condition_variable>
#include <functional>
#include <queue>
#include <iostream>
#include <mutex>
#include <random>
#include <thread>

class Monitor {
public:
    void lock() const {
```

```
14              monitMutex.lock();
15          }
16
17          void unlock() const {
18              monitMutex.unlock();
19          }
20
21          void notify_one() const noexcept {
22              monitCond.notify_one();
23          }
24
25          template <typename Predicate>
26          void wait(Predicate pred) const {
27              std::unique_lock<std::mutex> monitLock(monitMutex);
28              monitCond.wait(monitLock, pred);
29          }
30
31      private:
32          mutable std::mutex monitMutex;
33          mutable std::condition_variable monitCond;
34      };
35
36      template <typename T>
37      class ThreadSafeQueue: public Monitor {
38       public:
39          void add(T val){
40              lock();
41              myQueue.push(val);
42              unlock();
43              notify_one();
44          }
45
46          T get(){
47              wait( [this] { return ! myQueue.empty(); } );
48              lock();
49              auto val = myQueue.front();
50              myQueue.pop();
51              unlock();
52              return val;
53          }
54
55      private:
56          std::queue<T> myQueue;
57      };
58
```

```
59
60    class Dice {
61    public:
62        int operator()(){ return rand(); }
63    private:
64        std::function<int()> rand = std::bind(std::uniform_int_distribution<>(1, 6),
65                                              std::default_random_engine());
66    };
67
68
69    int main(){
70
71        std::cout << '\n';
72
73        constexpr auto NumberThreads = 10;
74
75        ThreadSafeQueue<int> safeQueue;
76
77        auto addLambda = [&safeQueue](int val){ safeQueue.add(val);
78                                                std::cout << val << " "
79                                                << std::this_thread::get_id() << "; ";
80                                              };
81        auto getLambda = [&safeQueue]{ safeQueue.get(); };
82
83        std::vector<std::thread> addThreads(NumberThreads);
84        Dice dice;
85        for (auto& thr: addThreads) thr = std::thread(addLambda, dice() );
86
87        std::vector<std::thread> getThreads(NumberThreads);
88        for (auto& thr: getThreads) thr = std::thread(getLambda);
89
90        for (auto& thr: addThreads) thr.join();
91        for (auto& thr: getThreads) thr.join();
92
93        std::cout << "\n\n";
94
95    }
```
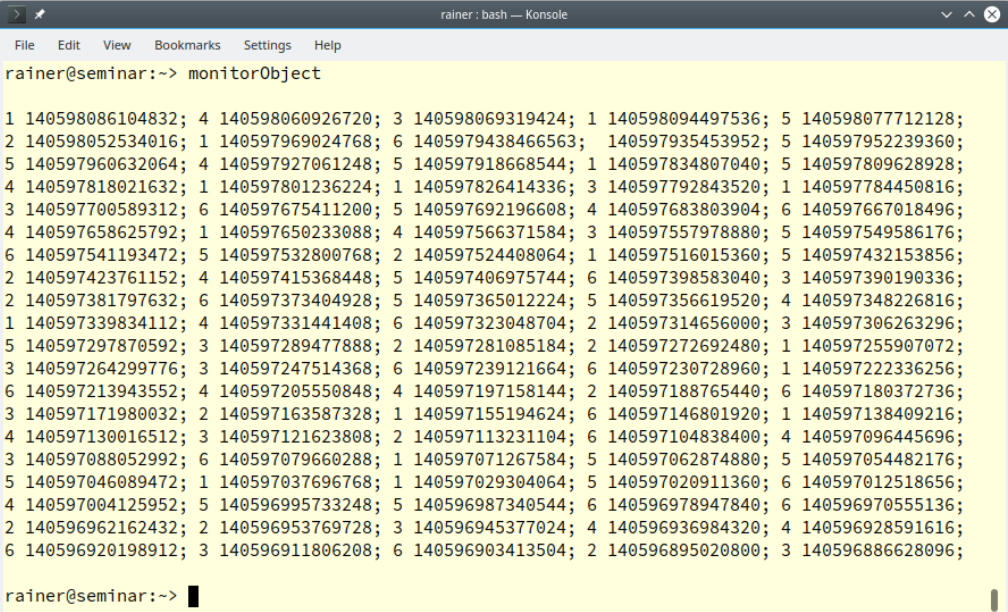
The central idea of the example is that the Monitor Object is encapsulated in a class and can, therefore, be reused. The class Monitor uses a std::mutex as monitor lock and std::condition_variable as monitor condition. The class Monitor provides the minimal interface that a Monitor Object should support.

ThreadSafeQueue in line 36 - 57 extends the std::queue in line 56 with a thread-safe interface. ThreadSafeQueue derives from the class Monitor and uses its member functions to support the

synchronized member functions `add` and `get`. The member functions `add` and `get` use the monitor lock to protect the Monitor Object, particularly the non-thread-safe `myQueue`. `add` notifies the waiting thread when a new item was added to `myQueue`. This notification is thread-safe. The member function `get` (lines 46 - 53) deserves more attention. First, the `wait` member function of the underlying condition variable is called. This `wait` call needs an additional predicate to protect against spurious and lost wakeups. The operations modifying the `myQueue` (lines 49 and 50) must also be protected because they can interleave with the call `myQueue.push(val)` (line 41). The Monitor Object `safeQueue` line 75 uses the lambda functions in lines 77 and 81 to add or remove a number from the synchronized `safeQueue`. `ThreadSafeQueue` itself is a class template and can hold values from an arbitrary type. One hundred clients add 100 random numbers between 1 - 6 to the `safeQueue` (line 75), while one hundred clients remove these 100 numbers concurrently from the `safeQueue`. The output of the program shows the numbers and the thread ids.

Monitor Object

With C++20, der program `monitorObject` can be further improved. First, include the header `<concepts>` and use the concept `std::predicate` as restricted typ parameter in the function template `wait` (lines 25 - 29). The concept `std::predicate` ensures that the function template `wait` can only be instantiated with a predicate.

Use of the concept `predicate`

```cpp
// monitorObjectCpp20.cpp
...

template <std::predicate Predicate>
void wait(Predicate pred) const {
    std::unique_lock<std::mutex> monitLock(monitMutex);
    monitCond.wait(monitLock, pred);
}
```

Second, use `std::jthread` instead of `std::thread`.

Use of `std::jthread`

```cpp
// monitorObjectCpp20.cpp
...

int main() {

    std::cout << '\n';

    constexpr auto NumberThreads = 100;

    ThreadSafeQueue<int> safeQueue;

    auto addLambda = [&safeQueue](int val){ safeQueue.add(val);
                                            std::cout << val << " "
                                            << std::this_thread::get_id() << "; ";
                                        };
    auto getLambda = [&safeQueue]{ safeQueue.get(); };

    std::vector<std::jthread> addThreads(NumberThreads);
    Dice dice;
    for (auto& thr: addThreads) thr = std::jthread(addLambda, dice());

    std::vector<std::jthread> getThreads(NumberThreads);
    for (auto& thr: getThreads) thr = std::jthread(getLambda);

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

}
```

The Active Object and the Monitor Object are similar but distinct in a few important points. Both architectural patterns synchronize the access to a shared object. The member functions of an Active

Object are executed in a different thread, but the Monitor Object member functions in the same thread. The Active Object decouples it's member function invocation better from its member function execution and is, therefore, easier to maintain.

10.3 Half-Sync/Half-Async

The Half-Sync/Half-Async architectural pattern decouples asynchronous and synchronous service processing in concurrent systems to simplify programming without unduly reducing performance. The pattern introduces two intercommunicating layers, one for asynchronous and one for synchronous service processing.

10.3.1 Challenges

Concurrent system often support asynchronous and synchronous services. Asynchronous services are typically faster but also more complex to program.

The architecture should support

- synchronous services for simplicity but enable also asynchronous services for performance reasons.
- communication between the asynchronous service and the synchronous services.

The Half-Sync/Half-Async pattern is often used in event-loops of servers or graphical user interfaces.

10.3.2 Solution

The event loop's typical workflow is to accept the client or user event, insert the request into a queue, and process the request synchronously in a separate thread. Accepting requests asynchronously ensures efficiency and the synchronous processing simplifies the processing of the request. The asynchronous and the synchronous service layers are decomposed into two layers, and the queue coordinates between both. The asynchronous layer consists of the lower-level system services such as interrupts, whereas the synchronous layer of high-level services such as database queries or file manipulations. The asynchronous and the synchronous layer can talk to each other via the queueing layer.

10.3.3 Components

Half-Sync/Half-Async
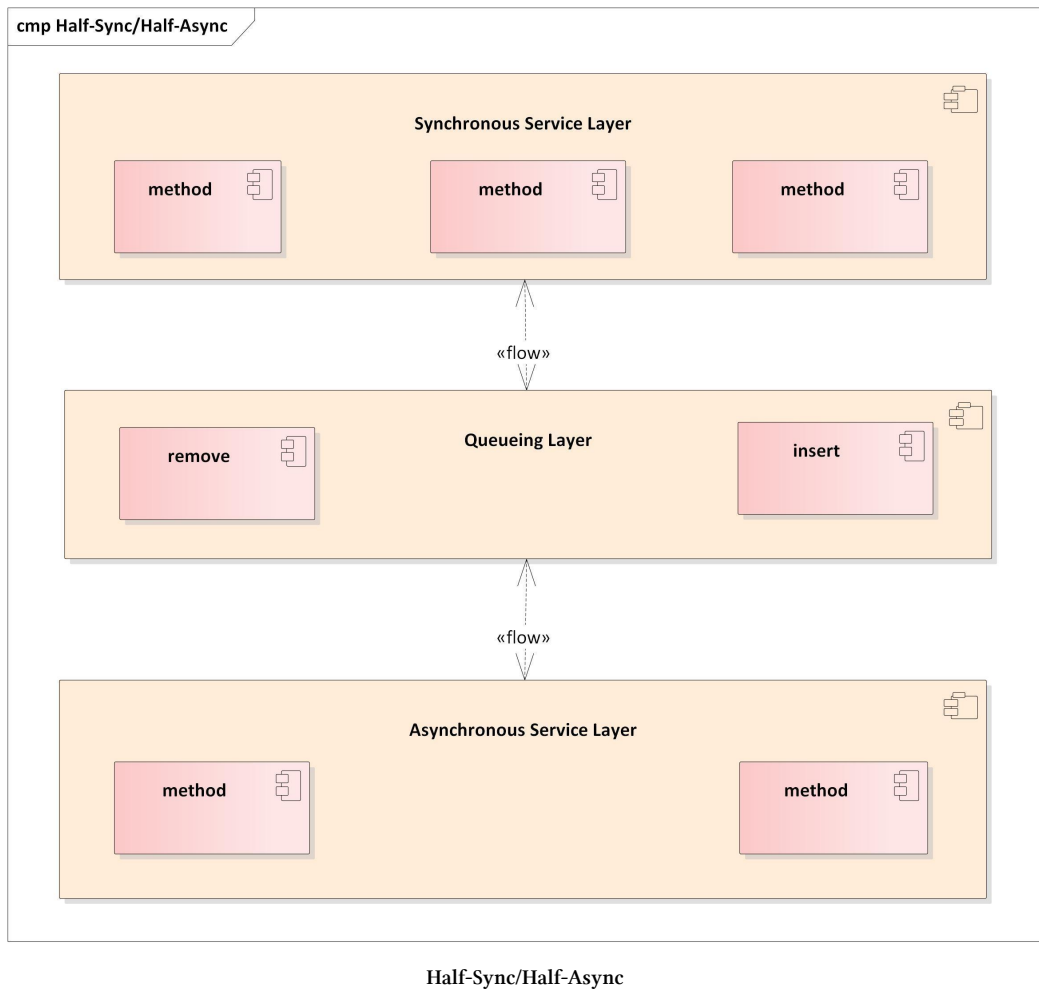
The Half-Sync/Half-Async pattern consists of four components.

- **Synchronous service layer**
  - The synchronous service layer performs high-level services. Typically, the high-level services run in a separate thread.
  - Services of this layer can block.
- **Asynchronous service layer**
  - The asynchronous service layer performs low-level services.

- Services in this layer cannot block.
- **Queueing layer**
  - The queueing layer serves as a communication channel between the synchronous and asynchronous layers.
  - It notifies a layer when messages are passed from the other layer.
- **External event source**
  - Creates external events for the asynchronous service layer.

10.3.4 Dynamic Behavior

- **Asynchronous phase**
  - External sources send notifications to the asynchronous layer.
  - The asynchronous service sends its result to the synchronous service using the queueing layer.
- **Queueing phase**
  - Buffers input from the asynchronous layer and notifies the synchronous layer.
- **Synchronous phase**
  - Process the input from the queueing layer provided by the asynchronous layer.

10.3.5 Advantages and Disadvantages

What are the advantages and disadvantages of the Half-Sync/Half-Async pattern?

- Advantages:
  - A clear separation of asynchronous and synchronous services. Low-level system services are handled in the asynchronous, and high-level services are handled in the synchronous layer.
  - The queueing layer ensures the loose coupling of the asynchronous and the synchronous layer.
  - The clear separation makes the software easier to understand, debug, maintain, and extend.
  - Blocking in the synchronous services does not affect the asynchronous services.
- Disadvantages:
  - The boundary-crossing between the asynchronous and synchronous layer may cause overhead. Often, the boundary-crossing involves a context switch between the kernel-space and the user-space because the asynchronous services run typically in the kernel-space and the synchronous services in the user-space.
  - The strict separation of the layers requires that the data is either copied or is immutable
  - Due to the inverted flow of control, debugging and testing is challenging.

10.3.6 Example

A defibrillator has to react to synchronous and asynchronous events. Synchronous events are for example, instructions for the operator to perform the resuscitation. On the contrary, pushing the power button by the operator is an asynchronous event. Two conditions must be met to apply the current to the patient using the power button: the heart rhythm is treatable, and the current must be synchronized with the insufficient heart muscle activity. Consequently, the push button event is stored on the queuing layer and has a higher priority than the instructions events for the operator.

The Half-Sync/Half-Async pattern is often used in an event demultiplexing and dispatching frameworks such as the Reactor or Proactor pattern.

10.4 Reactor

The Reactor pattern is an event-driven framework to demultiplex and dispatch service requests concurrently onto various service providers. The requests are processed synchronously.The Reactor pattern is also known as dispatcher or notifier.

10.4.1 Challenges

The server should handle more client requests concurrently. Each client request has a unique identifier, which enables the mapping to the specific service provider. The following points must hold. The Reactor should

- not block.
- support maximum throughput, avoid unnecessary context switches, and, therefore, the copying or synchronization of data.
- be easily expandable to support improved or new services.
- hide the application from multi-threading and synchronization challenges.

10.4.2 Solution

For each supported service type implement an event handler that fulfills the specific client request. Register this service handler within the Reactor. The Reactor uses an event demultiplexer to wait synchronously on all incoming events. When an event arrives, the Reactor is notified and dispatches it to the specific service.

10.4.3 Components

Reactor
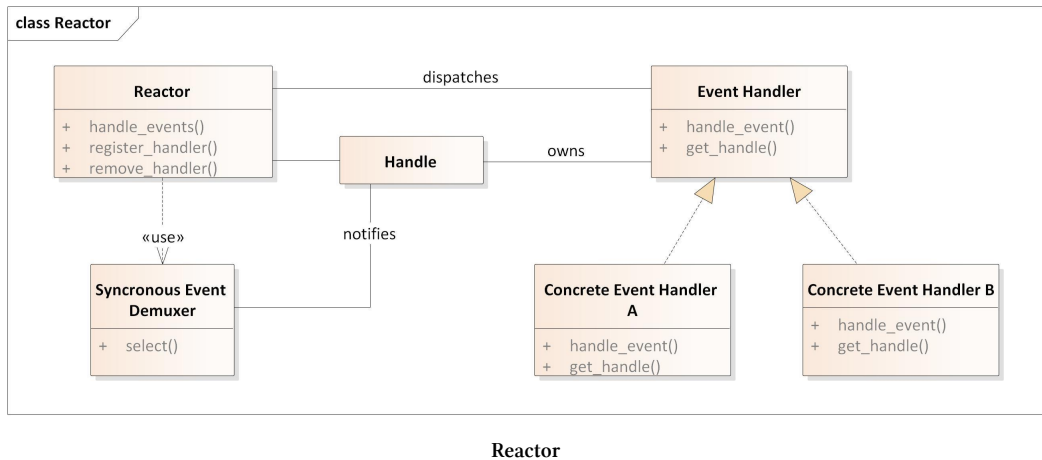
- **Handles**:
  - The handles identify different event sources such as network connections, open files, or GUI events.
  - The event source generates events such as connect, read, or write that are queued on the associated handle.

- **Synchronous event demultiplexer**:
  - The synchronous event demultiplexer waits for one or more indication events, and it blocks until the associated handle can process the event.

- **Event handler**:
  - The event handler defines the interface for processing the indication events.
  - The event handler defines the supported services of the application.

- **Concrete event handler**.
  - The concrete event handler implements the interface of the application defined by the event handler.

- **Reactor**:
  - The Reactor supports an interface to register and deregister the concrete event handler using file descriptors.
  - The Reactor uses a synchronous event demultiplexer to wait for indication events. An indication event can be read, write, or error event.
  - The Reactor maps the events to their concrete event handler.
  - The Reactor manages the lifetime of the event loop.

The Reactor (and not the application) waits for the indication events to demultiplex and dispatch the event. The concrete event handlers are registered within the Reactor. The Reactor inverts the flow of control. This inversion of control is often called Hollywood principle[14].

🔑 Synchronous Event Demultiplexer

The system calls `select`[15], `poll`[16], `epoll`[17], `kqueue`[18], or `WaitForMultipleObjects`[19] enables it to wait for indication events.

- `select` can only monitor 1024 file descriptors, and you should, therefore, only use it if other synchronous event demultiplexers are not available. `select` is supported on all Unix[20] and POSIX[21] operating systems.
- `poll` behaves similarly to `select` but overcomes its 1024 file descriptors limitation. Both system calls require that you specify the file descriptor with the highest number, and the system calls then scan all possible file descriptor numbers up to this highest set number. This strategy makes `select` and `poll` slower than `epoll`.
- `epoll` monitors only the specified file descriptors but is only available on Linux[22] operating systems.
- `kqueue` behaves similar to `epoll` and is available on the FreeBSD[23], and macOS[24] operating system.
- `WaitForMultipleObjects` is part of the Windows API[25].

10.4.4 Dynamic Behavior

The following points illustrate how the flow of control between the Reactor and and the evant handler goes.

- The application registers event handler for specific events in the Reactor.
- Each event handler provides it specific handler to the Reactor.
- The application starts the event loop. The event loop waits for indication events.

---

[14]https://en.wikipedia.org/wiki/Inversion_of_control
[15]https://en.wikipedia.org/wiki/Select_(Unix)
[16]https://man7.org/linux/man-pages/man2/poll.2.html
[17]https://en.wikipedia.org/wiki/Epoll
[18]https://en.wikipedia.org/wiki/Kqueue
[19]https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-waitformultipleobjects
[20]https://en.wikipedia.org/wiki/Unix
[21]https://en.wikipedia.org/wiki/POSIX
[22]https://en.wikipedia.org/wiki/Linux
[23]https://en.wikipedia.org/wiki/FreeBSD
[24]https://en.wikipedia.org/wiki/MacOS
[25]https://en.wikipedia.org/wiki/Windows_API

- The event demultiplexer returns to the Reactor when a event source becomes ready.
- The Reactor dispatches the handles to the corresponding event handler.
- The event handler processes the event.

10.4.5 Advantages and Disadvantages

What are the advantages and disadvantages of the reactor pattern?

- Advantages:
  - A clear separation of framework and application logic.
  - The modularity of various concrete event handlers.
  - The Reactor can be ported to various platforms, because the underlying event demultiplexing functions such as `select`[26], `epoll`[27], or `WaitForMultipleObjects`[28] are available on Unix (`select`, `epoll`), and Windows platforms (`WaitForMultipleObjects`).
  - The separation of interface and implementation enables easy adaption or extension of the services.
  - Overall structure supports the concurrent execution.

- Disadvantages:
  - Requires an event demultiplexing system call.
  - A long-running event handler can block the Reactor.
  - The inversion of control makes testing and debugging more difficult.

10.4.6 Example

The example uses the POCO framework[29]. *The POCO C++ Libraries are powerful cross-platform C++ libraries for building network- and internet-based applications that run on desktop, server, mobile, IoT, and embedded systems.*

---

[26]https://en.wikipedia.org/wiki/Select_(Unix)
[27]https://en.wikipedia.org/wiki/Epoll
[28]https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-waitformultipleobjects
[29]https://pocoproject.org/

The reactor pattern

```cpp
1   // reactor.cpp
2
3   #include <fstream>
4   #include <string>
5
6   #include "Poco/Net/SocketReactor.h"
7   #include "Poco/Net/SocketAcceptor.h"
8   #include "Poco/Net/SocketNotification.h"
9   #include "Poco/Net/StreamSocket.h"
10  #include "Poco/Net/ServerSocket.h"
11  #include "Poco/Observer.h"
12  #include "Poco/Thread.h"
13  #include "Poco/Util/ServerApplication.h"
14
15  using Poco::Observer;
16  using Poco::Thread;
17
18  using Poco::Net::ReadableNotification;
19  using Poco::Net::ServerSocket;
20  using Poco::Net::ShutdownNotification;
21  using Poco::Net::SocketAcceptor;
22  using Poco::Net::SocketReactor;
23  using Poco::Net::StreamSocket;
24
25  using Poco::Util::Application;
26
27  class EchoHandler {
28   public:
29    EchoHandler(const StreamSocket& s, SocketReactor& r): socket(s), reactor(r) {
30      reactor.addEventHandler(socket,
31        Observer<EchoHandler, ReadableNotification>(*this, &EchoHandler::socketReadable));
32    }
33
34    void socketReadable(ReadableNotification*) {
35      char buffer[8];
36      int n = socket.receiveBytes(buffer, sizeof(buffer));
37      if (n > 0) {
38        socket.sendBytes(buffer, n);
39      }
40      else {
41        reactor.removeEventHandler(socket,
42                Observer<EchoHandler, ReadableNotification>(*this, &EchoHandler::socketReadable));
43        delete this;
44      }
```

```
45      }
46
47    private:
48      StreamSocket socket;
49      SocketReactor& reactor;
50  };
51
52  class DataHandler {
53    public:
54
55      DataHandler(StreamSocket& s, SocketReactor& r): socket(s), reactor(r),
56                                                  outFile("reactorOutput.txt") {
57        reactor.addEventHandler(socket,
58          Observer<DataHandler, ReadableNotification>(*this, &DataHandler::socketReadable));
59        reactor.addEventHandler(socket,
60          Observer<DataHandler, ShutdownNotification>(*this, &DataHandler::socketShutdown));
61        socket.setBlocking(false);
62      }
63
64      ~DataHandler() {
65        reactor.removeEventHandler(socket,
66          Observer<DataHandler, ReadableNotification>(*this, &DataHandler::socketReadable));
67        reactor.removeEventHandler(socket,
68          Observer<DataHandler, ShutdownNotification>(*this, &DataHandler::socketShutdown));
69      }
70
71      void socketReadable(ReadableNotification*) {
72        char buffer[64];
73        int n = 0;
74        do {
75          n = socket.receiveBytes(&buffer[0], sizeof(buffer));
76          if (n > 0) {
77            std::string s(buffer, n);
78            outFile << s << std::flush;
79          }
80          else break;
81        } while (true);
82      }
83
84      void socketShutdown(ShutdownNotification*) {
85        delete this;
86      }
87
88    private:
89      StreamSocket socket;
```

```
90     SocketReactor& reactor;
91     std::ofstream outFile;
92   };
93
94   class Server: public Poco::Util::ServerApplication {
95
96    protected:
97     void initialize(Application& self) {
98       ServerApplication::initialize(self);
99     }
100
101    void uninitialize() {
102       ServerApplication::uninitialize();
103     }
104
105    int main(const std::vector<std::string>&) {
106
107       ServerSocket serverSocketEcho(4711);
108       ServerSocket serverSocketData(4712);
109       SocketReactor reactor;
110       SocketAcceptor<EchoHandler> acceptorEcho(serverSocketEcho, reactor);
111       SocketAcceptor<DataHandler> acceptorData(serverSocketData, reactor);
112       Thread thread;
113       thread.start(reactor);
114       waitForTerminationRequest();
115       reactor.stop();
116       thread.join();
117
118       return Application::EXIT_OK;
119
120     }
121
122   };
123
124   int main(int argc, char** argv) {
125
126     Server app;
127     return app.run(argc, argv);
128
129   }
```

Line 126 generates the TCP server. The server performs the main function (line 105) and is initialized in line 97 and uninitialized in line 102. The main function of the TCP server creates two server sockets, listening on port 4711 (line 107) and port 4712 (line 108). Line 110 and 111 create the server sockets

using the `EchoHandler` and the `DataHandler`. The `SocketAcceptor` models the Acceptor component of the Accepter-Connector design pattern. The `reactor` runs in a separate thread (line 113) until it gets its termination request (line 115). The `EchoHandler` registers its read handle in the constructor (line 30), and it unregisters its read handle in the member function `socketReadable` (line 41). The echo service send the clients message back (line 38). On the contrary, the `DataHandler` enables a client to transfer data to the server. The handler registers in its constructor its action for reading events (line 57) and shutdown events (line 59). Both handlers are unregistered in the destructor of `DataHandler` (line 64). The result of the data transfer is directly written to the file handle `outFile` (line 78).

🔑 The Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP)

Typically, either TCP[30] or the UDP[31] protocol is used for internet communication.

TCP is a connection-oriented protocol, and data is transmitted as a byte stream in both directions. TCP is reliable and guarantees the delivery of data in the order sent. Higher-order protocols such as Hypertext Transfer Protocol[32](HTTP), Hypertext Transfer Protocol Secure[33](HTTPS), Simple Mail Transfer Protocol[34](SMTP), or File Transfer Protocol[35](FTP) use TCP.

UDP is a connection-less protocol without the reliability of TCP. Data delivered in packets can be lost or be delivered out of order sent. UPD establishes no connection and is due to its lightweight structure faster than TCP. The protocols Domain Name System[36](DNS), Simple Network Management Protocol[37](SNMP), or Dynamic Host Configuration Protocol[38](DHCP) use UDP.

The following output shows on the left the server and on the right both clients. A telnet session serves as a client. The first client connects to port 4711: `telnet 127.0.0.1 4711`. This client connects with the echo server and displays, therefore, its request. The second client connects to port 4712: `telnet 127.0.0.1 4712`. The servers output shows that the client data is transferred to the server.

---

[30]https://en.wikipedia.org/wiki/Transmission_Control_Protocol
[31]https://en.wikipedia.org/wiki/User_Datagram_Protocol
[32]https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[33]https://en.wikipedia.org/wiki/HTTPS
[34]https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol
[35]https://en.wikipedia.org/wiki/File_Transfer_Protocol
[36]https://en.wikipedia.org/wiki/Domain_Name_System
[37]https://en.wikipedia.org/wiki/Simple_Network_Management_Protocol
[38]https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol
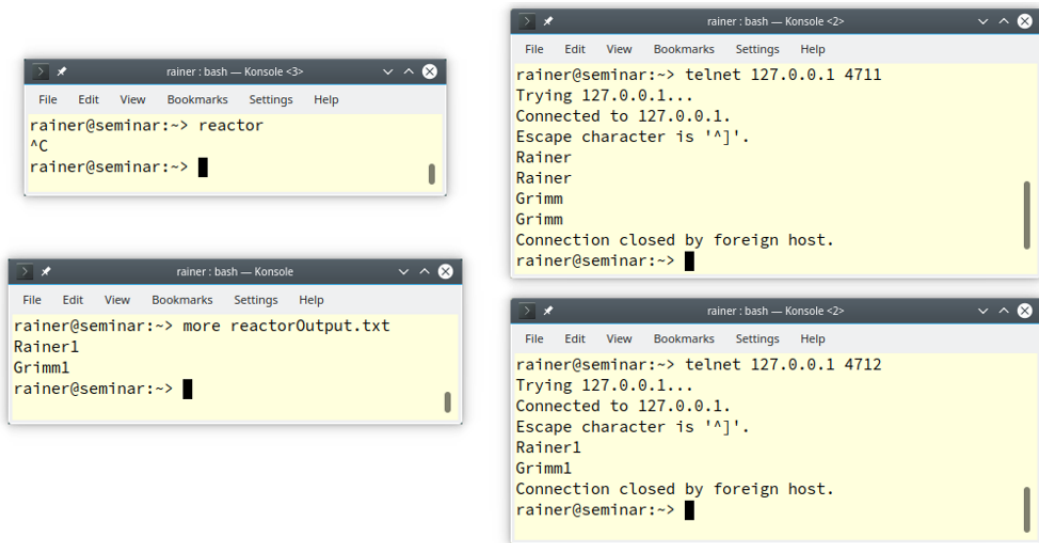
Reactor communicating with two clients

🔑 Acceptor-Connector

The Acceptor-Connector design pattern decouples the connection and initialization of services in a distributed system from the processing of the services after they are connected and initialized. It consists of three components: acceptor, connector, and service handler. The acceptor waits for a connection request from a remote connector and establishes an end-to-end service. The acceptor and the connector use their service handler to encapsulate the application-specific processing.

The Half-Sync/Half-Async pattern is typically used in the Reactor pattern to answer client requests in a separate thread.

The Proactor pattern is the asynchronous variant of the reactor pattern. The Reactor pattern demultiplexes and dispatches its event handler synchronously, but the Proactor pattern asynchronously.

10.5 Proactor

The Proactor pattern enables event-driven applications to demultiplex and dispatch service requests triggered by the completion of an asynchronous operation.

10.5.1 Challenges

Processing multiple service requests asynchronously can often improve the performance of event-driven applications such as servers. Event-driven applications must process multiple events synchronously to achieve this performance and avoid expensive data synchronization or context switching. Further, the new or improved services should be easily integrated, and the application should be shielded from the multi-threading and synchronization challenges.

10.5.2 Solution

Split the application services into two parts: long-duration operations, which should run asynchronously, and completion handlers, which process the long-duration asynchronous operations. The processing of the completion handler is quite similar to the processing of the event handler in the Reactor pattern. Still, the asynchronous operation is typically the job of the operating system. As the Reactor pattern, the Proactor pattern defines an event loop.

Here is the unique part of the Proactor pattern compared to the Reactor pattern. An asynchronous operation such as a connection request is initiated, and the operation is performed without blocking the caller's thread. When the long-duration operation is done it puts a completion event into the completion event queue. The Proactor waits on the queue by using the asynchronous event demultiplexer. The asynchronous event demultiplexer removes completion events from the queue, and the Proactor dispatches it to the specific completion handler. This completion handler processes the result of the asynchronous operation.

10.5.3 Components

The Proactor consists of nine components.
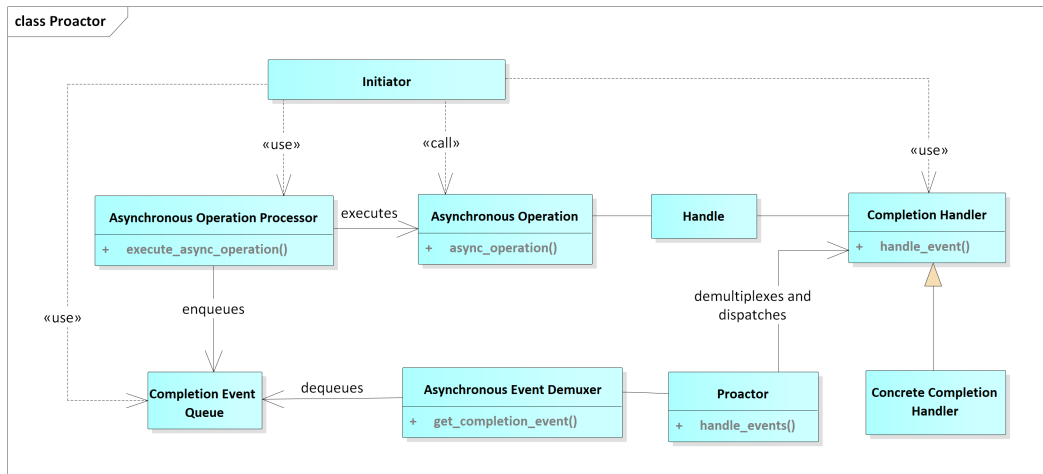
Proactor

- **Handle**:
  - stands for an entity of the operating system such as a socket that can generate a completion event
- **Asynchronous operation**:
  - is typically a long-duration operation that is executed asynchronously, and this can be a read or a write operation on a socket.
- **Asynchronous operation processor**:
  - executes an asynchronous operation and enqueues a completion event on the completion event queue when done
- **Completion handler**:
  - defines an interface for processing results of asynchronous operations
- **Concrete completion handler**:
  - processes the results of the asynchronous operations in an application-specific way
- **Completion event queue**:
  - buffers completion events until the asynchronous event demultiplexer dequeue them
- **Asynchronous event demultiplexer**:
  - can block while waiting for completion events to occur on a completion event queue
  - removes the completion event from the completion event queue
- **Proactor**:
  - calls the asynchronous event demultiplexer to dequeue a completion event
  - demultiplexes and dispatches completion events and invokes the concrete completion handler
- **Initiator**:
  - invokes the asynchronous operation
  - it interacts with the asynchronous operation processor

10.5.4 Advantages and Disadvantages

What are the advantages and disadvantages of the Proactor pattern?

- Advantages:
  - The application separates the application-independent asynchronous functionality from the application-specific functionality.
  - The Proactor can be used on various operating systems supporting different asynchronous event demultiplexer.
  - Applications do not need to start new threads because the long-duration asynchronous operations run in the caller's thread.
  - The Proactor pattern can avoid the cost of context switching.
  - The application logic doesn't start any threads, and, therefore, no synchronization is necessary.

- Disadvantages:
  - To apply the Proactor pattern most efficiently, the operating system should support asynchronous operations.
  - Due to the separation in time and space between the operation initiation and completion, debugging or testing the program is challenging.
  - The invocation of the asynchronous operation and maintaining of the completion event requires memory.

Asio

The Asio[39] library, which may become part of C++23 as networking library, enables you to implement the Proactor pattern in C++. Asio from Christopher Kohlhoff "is a cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach".

10.5.5 Example

The example uses the Asio[40] library that may become part of C++23 as a networking library. Asio enables it to implement the Proactor pattern in C++. Christopher Kohlhoff is the creator of Asio. He characterizes the library as *a cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach.*

---

[39]https://think-async.com/Asio/l
[40]https://think-async.com/Asio/l

The proactor pattern

```cpp
1   // proactor.cpp
2
3   #include <fstream>
4   #include <iostream>
5   #include <memory>
6   #include <string>
7   #include <utility>
8   #include <asio/ts/buffer.hpp>
9   #include <asio/ts/internet.hpp>
10
11  using asio::ip::tcp;
12
13  class EchoSession : public std::enable_shared_from_this<EchoSession> {
14   public:
15    EchoSession(tcp::socket sock): socket(std::move(sock)) { }
16
17    void start() {
18      do_read();
19    }
20
21   private:
22    void do_read() {
23      auto self(shared_from_this());
24      socket.async_read_some(asio::buffer(data, max_length),
25          [this, self](std::error_code, std::size_t length) {
26            do_write(length);
27          });
28    }
29
30    void do_write(std::size_t length) {
31      auto self(shared_from_this());
32      asio::async_write(socket, asio::buffer(data, length),
33          [this, self](std::error_code, std::size_t) {
34            do_read();
35          });
36    }
37
38    tcp::socket socket;
39    enum { max_length = 1024 };
40    char data[max_length];
41  };
42
43  class DataSession : public std::enable_shared_from_this<DataSession> {
44   public:
```

```
45    DataSession(tcp::socket sock, std::string fileName): socket(std::move(sock)), outFile(f\
46  ileName) {
47    }
48
49    void start() {
50      do_read();
51    }
52
53   private:
54    void do_read() {
55      auto self(shared_from_this());
56      socket.async_read_some(asio::buffer(data, max_length),
57          [this, self](std::error_code, std::size_t length) {
58            addData(length);
59          });
60    }
61    void addData(std::size_t length) {
62      std::string s(data, length);
63      outFile << s << std::flush;
64      do_read();
65    }
66
67    tcp::socket socket;
68    enum { max_length = 1024 };
69    char data[max_length];
70    std::ofstream outFile;
71  };
72
73  class EchoServer {
74   public:
75    EchoServer(asio::io_context& io_context, short port)
76      : acceptor(io_context, tcp::endpoint(tcp::v4(), port)), socket(io_context) {
77      do_accept();
78    }
79
80   private:
81    void do_accept() {
82      acceptor.async_accept(socket,
83          [this](std::error_code) {
84            std::make_shared<EchoSession>(std::move(socket))->start();
85            do_accept();
86          });
87    }
88
89    tcp::acceptor acceptor;
```

```
90     tcp::socket socket;
91   };
92
93   class DataServer {
94    public:
95     DataServer(asio::io_context& io_context, short port, const std::string& file)
96         : acceptor(io_context, tcp::endpoint(tcp::v4(), port)),
97           socket(io_context), fileName(file) {
98       do_accept();
99     }
100
101    private:
102     void do_accept() {
103       acceptor.async_accept(socket,
104           [this](std::error_code) {
105             std::make_shared<DataSession>(std::move(socket), std::move(fileName))->start();
106             do_accept();
107           });
108     }
109
110     tcp::acceptor acceptor;
111     tcp::socket socket;
112     std::string fileName;
113   };
114
115
116   int main() {
117
118     asio::io_context io_context;
119     EchoServer echoServer(io_context, 4711);
120     DataServer dataServer(io_context, 4712, "proactorOutput.txt");
121     io_context.run();;
122
123   }
```

The program uses an EchoServer (line 118) and a DataServer, listening to the specified Ports 4711 and 4712. The EchoServer writes the incoming data back, and the DataServer writes them into the file proactorOutput.txt.

The EchoServer accepts in the member function do_ accept (line 80) client's requests and handles them using EchoSession (line 13). do_accept is recursively called (line 84). When a client request is The crucial parts of the EchoSession are the member functions do_read (line 22) and do_write (line 30). do_read reads the data from the client connection asynchronously (line 24) and do_write writes them asynchronously (line 32) back. In the end, the member function do_write calls do_read (line 34) to read all incoming client data.

The workflow of the `DataServer` and its `DataSession` (line 43) is similar. The crucial difference is that the member function `do_read` writes the client data to the output file (line 57).

The following example uses two telnet[41] sessions to invoke the TCP-server. Both TCP clients connect to address 127.0.0.1. They use the port 4711 and 4712 respectively. Due to the `EchoServer`, data written to the port 4711 is immediately written back. The `DataServer` writes the client data to the output file `more proactorOutput.txt`.

Proactor communicating with two clients

10.6 Further Information

- Adaptive Communication Environment (ACE)[42]
- Boost.Asio[43]
- Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects[44]

[41]https://en.wikipedia.org/wiki/Telnet
[42]https://en.wikipedia.org/wiki/Adaptive_Communication_Environment
[43]https://www.boost.org/doc/libs/1_69_0/doc/html/boost_asio.html
[44]https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/

Distilled Information

- The two patterns active object and the monitor object synchronize and schedule member functions invocation. The member functions of an Active Object are executed in a different thread, but the Monitor Object member functions in the same thread.
- The Half-Sync/Half-Async pattern has an architectural focus and decouples asynchronous and synchronous service processing in concurrent systems. Both layers communicate using a queuing layer.
- The Reactor pattern is an event-driven framework to demultiplex and dispatch service requests concurrently onto various service providers.
- The Proactor pattern enables event-driven applications to demultiplex and dispatch service requests triggered by the completion of an asynchronous operation.

11. Best Practices

Cippi studies

This chapter provides you with a simple set of rules for writing well-defined and fast, concurrent programs in modern C++. Multithreading, particularly parallelism and concurrency, is quite a new topic in C++; therefore, more and more best practices are discovered in the coming years. Consider the rules in this chapter not as a complete list; but rather as a necessary starting point that evolves. This holds particularly true for the parallel STL. When updating this book (12/2018), the parallel algorithms of C++17 are only partially available; therefore, it is too early to formulate best practices for it.

11.1 General

Let's start with a few very general best practices that apply to atomics and threads.

11.1.1 Code Reviews

Code reviews should be part of each professional software development process. This holds especially true when you deal with concurrency. Concurrency is inherently complicated and requires a lot of thoughtful analysis and experience.

To make the review most effective, send the code you want to discuss to the reviewers before the review. Explicitly state which invariants should apply to your code. The reviewers should have enough time to analyze the code before the official review starts.

Not convinced? Let me give you an example. Do you remember the data races in the program `readerWriterLock.cpp` in the chapter `std::shared_lock`?

Reader–writer locks

```cpp
1    // readerWriterLock.cpp
2
3    #include <iostream>
4    #include <map>
5    #include <shared_mutex>
6    #include <string>
7    #include <thread>
8
9    std::map<std::string,int> teleBook{{"Dijkstra", 1972}, {"Scott", 1976},
10                                        {"Ritchie", 1983}};
11
12   std::shared_timed_mutex teleBookMutex;
13
14   void addToTeleBook(const std::string& na, int tele){
15     std::lock_guard<std::shared_timed_mutex> writerLock(teleBookMutex);
16     std::cout << "\nSTARTING UPDATE " << na;
17     std::this_thread::sleep_for(std::chrono::milliseconds(500));
18     teleBook[na]= tele;
19     std::cout << " ... ENDING UPDATE " << na << '\n';
20   }
21
22   void printNumber(const std::string& na){
23     std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
24     std::cout << na << ": " << teleBook[na];
25   }
26
27   int main(){
28
29     std::cout << '\n';
30
31     std::thread reader1([]{ printNumber("Scott"); });
32     std::thread reader2([]{ printNumber("Ritchie"); });
33     std::thread w1([]{ addToTeleBook("Scott",1968); });
34     std::thread reader3([]{ printNumber("Dijkstra"); });
35     std::thread reader4([]{ printNumber("Scott"); });
36     std::thread w2([]{ addToTeleBook("Bjarne",1965); });
37     std::thread reader5([]{ printNumber("Scott"); });
38     std::thread reader6([]{ printNumber("Ritchie"); });
39     std::thread reader7([]{ printNumber("Scott"); });
40     std::thread reader8([]{ printNumber("Bjarne"); });
41
42     reader1.join();
43     reader2.join();
44     reader3.join();
```

```
45     reader4.join();
46     reader5.join();
47     reader6.join();
48     reader7.join();
49     reader8.join();
50     w1.join();
51     w2.join();
52
53     std::cout << '\n';
54
55     std::cout << "\nThe new telephone book" << '\n';
56     for (auto teleIt: teleBook){
57       std::cout << teleIt.first << ": " << teleIt.second << '\n';
58     }
59
60     std::cout << '\n';
61
62   }
```

The issue is that the call teleBook[na] is line 24 can modify the telephone book. You can provoke the data race by putting the reading thread reader8 in front of the other readers. I use this program in my C++ seminars as a kind of exercise. The exercise is to spot the data race. About 10% of the participants find the data race within 5 minutes.

11.1.2 Minimize Sharing of Mutable Data

You should minimize data sharing of mutable data for two reasons: performance and safety. Safety is mainly about data races. Let me focus on performance in this paragraph. I deal with correctness in the following best practices section.

In the chapter Calculating the Sum of a Vector I made an exhaustive performance study. How fast can I sum up the values of a std::vector?

This was the critical part of the single-threaded summation.

Single threaded summation

```
...

constexpr long long size = 100000000;

std::cout << '\n';

std::vector<int> randValues;
randValues.reserve(size);

// random values
std::random_device seed;std::mt19937 engine(seed());
std::uniform_int_distribution<> uniformDist(1, 10);

const unsigned long long sum = std::accumulate(randValues.begin(),
                                               randValues.end(), 0);

...
```

Afterward, I performed the summation on four threads. I started naively with a shared summation variable.

Multi threaded summation with a shared variable

```
...

void sumUp(unsigned long long& sum, const std::vector<int>& val,
           unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        std::lock_guard<std::mutex> myLock(myMutex);
        sum += val[it];
    }
}

...
```

Optimized a little bit by using an atomic summation variable.

Multi threaded summation with an atomic

```
...

void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
          unsigned long long beg, unsigned long long end){
    for (auto it = beg; it < end; ++it){
        sum.fetch_add(val[it]);
    }
}

...
```

And I got my performance improvement by calculating the partial sums locally.

Multi threaded summation with local variables

```
...

void sumUp(unsigned long long& sum, const std::vector<int>& val,
          unsigned long long beg, unsigned long long end){
    unsigned long long tmpSum{};
    for (auto i = beg; i < end; ++i){
        tmpSum += val[i];
    }
    std::lock_guard<std::mutex> lockGuard(myMutex);
    sum += tmpSum;
}


...
```

The performance numbers are quite impressive and give a clear indication. The less you share state, the more you get out of your cores.

Performance of the various summations on Linux

Single threaded	std::lock_guard	Atomics	Local summation
0.07 sec	3.34 sec	1.34 sec	0.03 sec

11.1.3 Minimize Waiting

You may have heard of Amdahl's law[1]. It predicts the theoretical maximum speedup you can get using multiple processors. The law is quite simple. If p is the proportion of your code, that can run concurrently, you get a maximum speedup of $\frac{1}{1-p}$. So, if 90% of your code can run concurrently, you get at most a 10 times speedup: $\frac{1}{1-p} == \frac{1}{1-0.9} == \frac{1}{0.1} == 10$.

To see it from the opposite perspective, if 10% of your code has to run sequentially because you use a lock, you get at most a ten times speedup. Of course, I assumed that you have access to infinite processing resources.

The graphic shows a direct consequence of Amdahl's law explicitly.
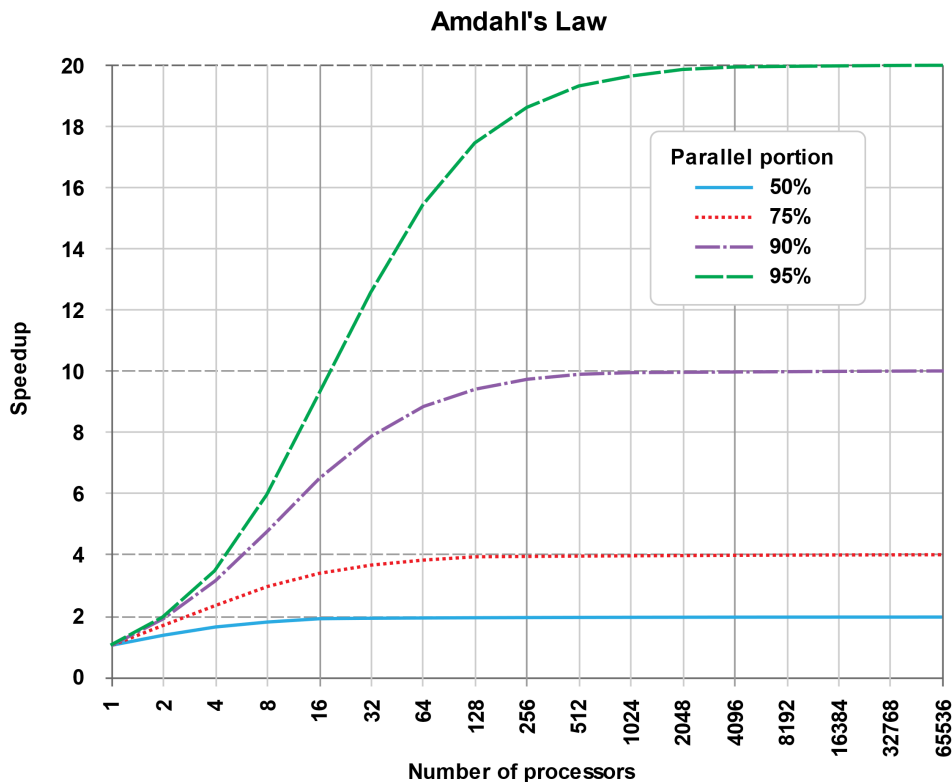
By Daniels220 at English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6678551

The optimum number of cores depends highly on the parallel portion of your code. For example: If you have 50% parallel code, you reach the peak performance with 16 cores. Using more cores makes

---

[1]https://en.wikipedia.org/wiki/Amdahl%27s_law

your program not faster. If you have 95% parallel code, you reach the peak performance with 2048 cores.

11.1.4 Prefer Immutable Data

A data race is a situation in which at least two threads access a shared variable simultaneously. At least one thread tries to modify the variable. The definition makes it quite obvious. A necessary condition for a data race is a mutable, shared state. The graphic makes my point clear.

Mutable and shared state

If you have immutable data, no data race can happen. You only have to guarantee that the immutable data are initialized in a thread-safe way. I presented in the chapter Thread-safe initialization four ways to guarantee this. Here are they:

- early initialization before a thread is created
- constant expressions
- the function `std::call_once` in combination with the flag `std::once_flag`
- a static variable with block scope

There are two typical ways to create immutable data in C++: `const` and `constexpr`. While `const` is a runtime technique, `constexpr` guarantees that the value is initialized at compile time and, therefore, thread-safe. Even user-defined types can be initialized at compile time.

11.1.4.1 A User-defined Type

There are a few restrictions for user-defined types which instances should be created at compile time.

The `constexpr` constructor

- can only be invoked with a constant expression.
- cannot use exception handling.

- has to be declared as `default` or `delete` or the function body must be empty (C++11).

The `constexpr` user-defined type

- cannot have virtual base classes.
- requires each base object and each non-static member to be initialized in the constructor's initialization list or directly in the class body. Consequently, it holds that each used constructor (e.g. of a base class) has to be `constexpr` constructor and that the applied initializers have to be constant expressions.

cppreference.com[2] provides additional information to `constexpr` user-defined types. To add praxis to the theory, I define the class `MyInt`. `MyInt` shows the just mentioned points. The class has also `constexpr` member functions.

Immutable user-defined types

```cpp
// userdefinedTypes.cpp

#include <iostream>
#include <ostream>

class MyInt{
public:
    constexpr MyInt()= default;
    constexpr MyInt(int fir, int sec): myVal1(fir), myVal2(sec){}
    MyInt(int i){
        myVal1= i - 2;
        myVal2= i + 3;
    }

    constexpr int getSum() const { return myVal1 + myVal2; }

    friend std::ostream& operator<< (std::ostream &out, const MyInt& myInt){
        out << "(" << myInt.myVal1 << "," << myInt.myVal2 << ")";
        return out;
    }

private:
    int myVal1= 1998;
    int myVal2= 2003;

};

int main(){

```

---

[2]https://en.cppreference.com/w/cpp/language/constexpr
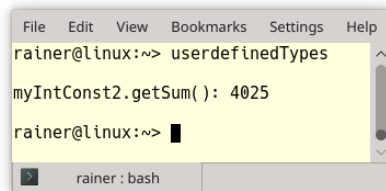
```
30      std::cout << '\n';
31
32      constexpr MyInt myIntConst1;
33
34      constexpr int sec = 2014;
35      constexpr MyInt myIntConst2(2011, sec);
36      std::cout << "myIntConst2.getSum(): " << myIntConst2.getSum() << '\n';
37
38      int arr[myIntConst2.getSum()];
39      static_assert( myIntConst2.getSum() == 4025, "2011 + 2014 should be 4025" );
40
41      std::cout << '\n';
42
43  }
```

The class `MyInt` has two `constexpr` constructors. A default constructor (line 8) and a constructor taking two arguments (line 9). Additionally, the class has one member function `getSum` that is a constant expression. I declared the member function `const` because a `constexpr` member function is in contrast to C++11 with C++14, not automatically `const`. The overloaded output operator `operator <<` is not a class member but can access its private and protected members. There are two ways to define the variables `myVal1` and `myVal2` (lines 23 and 24) if I use them in `constexpr` objects. First, I can initialize them in the constructor's initialization list (line 9); second, I can initialize them in the class body (lines 23 and 24). The initialization in the initialization list of the constructor has a higher priority.

Lines 38 and 39 shows that I can invoke the `constexpr` member function in a constant expression. This is the output of the program.

Using a `constexpr` object

I want to emphasize it once more explicitly: **A `constexpr` object can only use `constexpr` member functions**.

Functional programming languages such as Haskell, having no mutable data, are very suitable for concurrent programming.

11.1.5 Use pure functions

Haskell is called a pure functional language because it is based on pure functions. A pure function is a function which always produces the same results when given the same arguments. It has no side effect and can, therefore, not change the state of the program.

Pure functions have a significant advantage from the concurrency perspective. They can be reordered or automatically run on another thread.

Functions in C++ are per default impure. The following three functions are all pure, but each function has a different characteristic.

```cpp
int powFunc(int m, int n){
    if (n == 0) return 1;
    return m * powFunc(m, n-1);
}
```

powFunc is an ordinary function that runs at runtime.

```cpp
template<int m, int n>
struct PowMeta{
    static int const value = m * PowMeta<m, n-1>::value;
};

template<int m>
struct PowMeta<m, 0>{
    static int const value = 1;
};
```

PowMeta is a so-called meta-function because it runs at compile time.

```cpp
constexpr int powConst(int m, int n){
    int r = 1;
    for(int k = 1; k <= n; ++k) r *= m;
    return r;
}
```

The function powCont can run at runtime and at compile time. It is a constexpr function.

11.1.6 Look for the Right Abstraction

There are various ways to initialize a Singleton in a multithreading environment. You can rely on the standard library using a lock_guard or std::call_once, rely on the core language using a static

variable, or rely on atomics using acquire-release semantic. The acquire-release semantic is by far the most challenging one. It's a big challenge in various aspects. You have to implement it, maintain it, and explain it to your coworkers. In contrast to your effort, the well-known Meyers Singleton is a lot easier to implement and runs faster.

The story with the right abstractions goes on. Instead of implementing a parallel loop for summing up a container, use `std::reduce`. You can parametrize `std::reduce` with a binary callable and the parallel execution policy.

The more you go for the right abstraction, the less likely it becomes that you shoot yourself in the foot.

11.1.7 Use Static Code Analysis Tools

In the chapter on case studies, I introduced CppMem. CppMem[3] is an interactive tool for exploring the behavior of small code snippets using the C++ memory model. CppMem can help you in two aspects. First, you can verify the correctness of your code. Second, you get a deeper understanding of the memory model and, therefore, of the general's multithreading issues.

11.1.8 Use Dynamic Enforcement Tools

ThreadSanitizer[4] is a data race detector for C/C++. ThreadSanitizer is part of Clang 3.2 and GCC 4.8. To use ThreadSanitizer, you have to compile and link your program using the flag `-fsanitize=thread`.

The following program has a data race.

A data race on globalVar

```cpp
1  // dataRace.cpp
2
3  #include <thread>
4
5  int main(){
6
7    int globalVar{};
8
9    std::thread t1([&globalVar]{ ++globalVar; });
10   std::thread t2([&globalVar]{ ++globalVar; });
11
12   t1.join();
13   t2.join();
14
15 }
```

---

[3]http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/
[4]https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

t1 and t2 access globalVar at the same time. Both threads try to modify the globalVar. Let's compile and run the program.

```
g++ -std=c++11 dataRace.cpp -fsanitize=thread  -pthread -g -o dataRace
```

The output of the program is quite verbose.

A data race detected with ThreadSanitizer

I highlighted in red the critical line of the screenshot. There is a data race on line 10.

11.2 Multithreading

11.2.1 Threads

Threads are the basic building blocks for writing concurrent programs.

11.2.1.1 Minimize thread creation

How expensive is a thread? Quite expensive! This is the issue behind this best practice. Let me first talk about the usual size of a thread and then about the costs of its creation.

11.2.1.1.1 Size

A 'std::thread is a thin wrapper around the native thread. This means I'm interested in the size of a Windows thread and a POSIX thread[5] because most of the times, they are internally used.

- Windows systems: the post Thread Stack Size[6] gave me the answer: 1 MB.
- POSIX systems: the pthread_create[7] man-page provides me with the answer: 2MB. These is the sizes for the i386 and x86_64 architectures. If you want to know the sizes for other architectures that support POSIX, here are they:

Architecture	Default stack size
i386	2 MB
IA-64	32 MB
PowerPC	4 MB
S/390	2 MB
Sparc-32	2 MB
Sparc-64	4 MB
x86_64	2 MB

Stack size of an `std::thread`

11.2.1.1.2 Creation

I didn't find numbers how much time it takes to create a thread. To get a gut feeling, I made a simple performance test on Linux and Windows.

I used GCC 6.2.1 on a desktop and cl.exe on a laptop for my performance tests. The cl.exe is part of the Microsoft Visual Studio 2017. I compiled the programs with maximum optimization. This means on Linux the flag `O3` and on Windows `Ox`.

Here is my small test program.

---

[5]https://en.wikipedia.org/wiki/POSIX_Threads
[6]https://msdn.microsoft.com/en-us/library/windows/desktop/ms686774(v=vs.85).aspx
[7]http://man7.org/linux/man-pages/man3/pthread_create.3.html
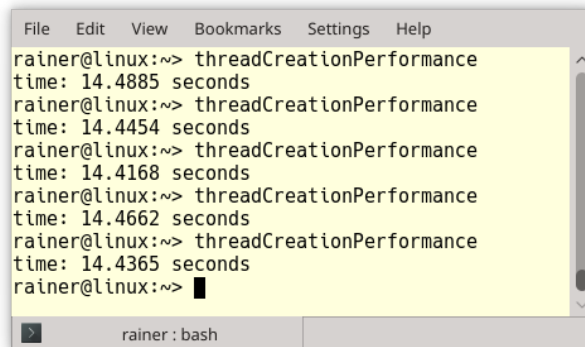
A small performance test for thread creation

```cpp
// threadCreationPerformance.cpp

#include <chrono>
#include <iostream>
#include <thread>

static const long long numThreads= 1'000'000;

int main(){

    auto start = std::chrono::system_clock::now();

    for (volatile int i = 0; i < numThreads; ++i) std::thread([]{}).detach();

    std::chrono::duration<double> dur= std::chrono::system_clock::now() - start;
    std::cout << "time: " << dur.count() << " seconds" << '\n';

}
```

The program creates 1 million threads that execute the empty lambda function in line 13. These are the numbers for Linux and Windows:

11.2.1.1.3 Linux

Thread creation on Linux

This means that creating a thread took about 14.5 sec / 1000000 = **14.5 microseconds on Linux**.

11.2.1.1.4 Windows

Thread creation on Windows

The thread creation took about 44 sec / 1000000 = **44 microseconds on Windows**.

To put it the other way around. You can create about **69 thousand threads on Linux and 23 thousand threads on Windows in one second**.

11.2.1.2 Use tasks instead of threads

std:async versus threads

```cpp
// asyncVersusThread.cpp

#include <future>
#include <thread>
#include <iostream>

int main(){

  std::cout << '\n';

  int res;
  std::thread t([&]{ res = 2000 + 11; });
  t.join();
  std::cout << "res: " << res << '\n';

  auto fut= std::async([]{ return 2000 + 11; });
  std::cout << "fut.get(): " << fut.get() << '\n';
```

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

Based on the program, there are many reasons for preferring tasks over threads. The main reasons are:

- you can use a secure communication channel for returning the result of the communication. If you use a shared variable, you have to synchronize the access to it.
- you can quite easily return values, notifications, and exceptions to the caller.

With extended futures, we get the possibility to compose futures and build highly sophisticated workflows. These workflows are based on the continuation then, and the combinations when_any and when_all.

11.2.1.3 Be extremely careful if you detach a thread

The following code snippet requires our full attention.

```
std::string s{"C++11"}

std::thread t([&s]{ std::cout << s << '\n'; });
t.detach();
```

Because thread t is detached from the lifetime of its creator, two race conditions can cause undefined behavior.

1. Thread t may outlive the lifetime of its creator. The consequence is that t refers to a non-existing std::string.
2. The program shuts down before thread t can do its work because the lifetime of the output stream std::cout is bound to the main thread's lifetime.

11.2.1.4 Consider using an automatic joining thread

A thread t with a callable unit is called joinable if neither a t.join() nor a t.detach() call happened. The destructor of a joinable thread throws the std::terminate exception. To not forget the t.join(), you can create your wrapper around std::thread. This wrapper checks in the constructor if the given thread is still joinable and joins the destructor's given thread.

You don't have to build this wrapper on your own. Use the std::jthread, scoped_thread from Anthony Williams, or the gsl::joining_thread from the guideline support library[8].

---

[8]https://github.com/Microsoft/GSL

11.2.2 Data Sharing

With data sharing of mutable data the challenges in multithreading programming start.

11.2.2.1 Pass data per default by copy

```
std::string s{"C++11"}

std::thread t1([s]{ ... }); // do something with s
t1.join();

std::thread t2([&s]{ ... }); // do something with s
t2.join();

// do something with s
```

If you pass data such as the `std::string s` to a thread `t1` by copy, the creator thread and the created thread `t1` use independent data. This is in contrast to the thread `t2`. It gets its `std::string s` by reference. This means you have to synchronize the access to `s` in the creator thread and the created thread `t2` preventively. This is error-prone and expensive.

11.2.2.2 Use `std::shared_ptr` to share ownership between unrelated threads

Assume you have an object which you want to share between unrelated threads. The critical question is, who is the object's owner and, therefore, responsible for releasing the memory? Now you can choose between a memory leak if you don't deallocate the memory or undefined behavior because you invoked delete more than once. Most of the time, the undefined behavior ends in a runtime crash.

The following program shows this non-solvable issue.

Unrelated threads share ownership

```
1   // threadSharesOwnership.cpp
2
3   #include <iostream>
4   #include <thread>
5
6   using namespace std::literals::chrono_literals;
7
8   struct MyInt{
9     int val{2017};
10    ~MyInt(){
11      std::cout << "Good Bye" << '\n';
12    }
13  };
```

```
14
15   void showNumber(MyInt* myInt){
16       std::cout << myInt->val << '\n';
17   }
18
19   void threadCreator(){
20       MyInt* tmpInt= new MyInt;
21
22       std::thread t1(showNumber, tmpInt);
23       std::thread t2(showNumber, tmpInt);
24
25       t1.detach();
26       t2.detach();
27   }
28
29   int main(){
30
31       std::cout << '\n';
32
33       threadCreator();
34       std::this_thread::sleep_for(1s);
35
36       std::cout << '\n';
37
38   }
```
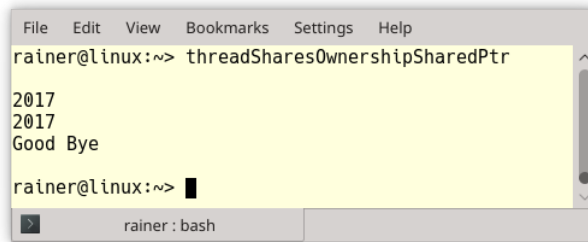
This example is intentionally easy. I let the main thread sleep for one second (line 34) to be sure that it outlives the lifetime of the child thread t1 and t2. This is, of course, no appropriate synchronization, but it helps me to make my point. The program's vital issue is: Who is responsible for the deletion of tmpIntin line 20? Thread t1 (line 22), thread t2 (line 23), or the function (main thread) itself. Because I can not forecast how long each thread runs, I decided to go with a memory leak. Consequentially, the destructor of MyInt in line 10 is never called:

Unrelated threads share ownership

The lifetime issues are pretty easy to handle if I use a std::shared_ptr.

Unrelated threads share ownership via `std::shared_ptr`

Unrelated threads share ownership via `std::shared_ptr`

```cpp
1    // threadSharesOwnershipSharedPtr.cpp
2
3    #include <iostream>
4    #include <memory>
5    #include <thread>
6
7    using namespace std::literals::chrono_literals;
8
9    struct MyInt{
10     int val{2017};
11     ~MyInt(){
12       std::cout << "Good Bye" << '\n';
13     }
14   };
15
16   void showNumber(std::shared_ptr<MyInt> myInt){
17       std::cout << myInt->val << '\n';
18   }
19
20   void threadCreator(){
21       auto sharedPtr = std::make_shared<MyInt>();
22
23       std::thread t1(showNumber, sharedPtr);
24       std::thread t2(showNumber, sharedPtr);
25
26       t1.detach();
27       t2.detach();
28   }
29
30   int main(){
31
```

```
32        std::cout << '\n';
33
34        threadCreator();
35        std::this_thread::sleep_for(1s);
36
37        std::cout << '\n';
38
39    }
```

Two minor changes to the source code were necessary. First, the pointer in line 21 became a `std::shared_ptr`, and second, the function `showNumber` in line 16 takes a smart pointer instead of a plain pointer.

11.2.2.3 Minimize the time holding a lock

If you hold a lock, only one thread can enter the critical section and make progress.

```
void setDataReadyBad(){
    std::lock_guard<std::mutex> lck(mutex_);
    mySharedWork = {1, 0, 3};
    dataReady = true;
    std::cout << "Data prepared" << '\n';
    condVar.notify_one();
}                                    // unlock the mutex

void setDataReadyGood(){
    mySharedWork = {1, 0, 3};
    {
        std::lock_guard<std::mutex> lck(mutex_);
        dataReady = true;
    }                                // unlock the mutex
    std::cout << "Data prepared" << '\n';
    condVar.notify_one();
}
```

The functions `setDataReadyBad` and `setDataReadyGood` are the notification components of a condition variable. The variable `dataReady` is necessary to protect against spurious wakeups and lost wakeups. Because `dataReady` is a non-atomic variable, it has to be synchronized using the lock `lck`. To make the lifetime of the lock as short as possible, use an artificial scope ({ ... }) such as in the function `setDataReadyGood`.

11.2.2.4 Put a mutex into a lock

You should not use a mutex without a lock.

```
std::mutex m;
m.lock();
// critical section
m.unlock();
```

Something unexpected may happen in the critical section, or you forget to unlock the mutex: the result is the same. If you don't unlock a mutex, another thread requiring the mutex is blocked, and you end with a deadlock.

Thanks to locks that automatically take care of the underlying mutex, your risk of getting a deadlock is considerably reduced. According to the RAII idiom, a lock automatically binds its mutex in the constructor and releases it in the destructor.

```
std::mutex m;
...
{
  std::lock_guard<std::mutex> lockGuard(m);
  // critical section
}           // unlock the mutex
```

The artificial scope ({ ... }) ensures that the lock's lifetime automatically ends; therefore, the underlying mutex is unlocked.

11.2.2.5 Try to lock at most one mutex at one point in time

Of course, sometimes you need more than one mutex at one point in time. In this case, you may become the victim of a race condition which causes a deadlock such as in the following chapter; therefore, you should try to avoid holding more than one mutex at one point in time if possible.

11.2.2.6 Give your locks a name

If you use a lock such as std::lock_guard without a name, it will be immediately destroyed.

```
std::mutex m;
...
{
    std::lock_guard<std::mutex>{m};
    // critical section
}
```
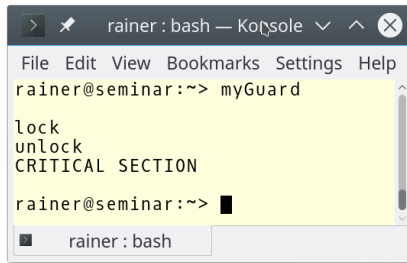
In this innocent-looking code snippet, the std::lock_guard is immediately destroyed. Therefore, the following critical section is executed without synchronization. The locks from the C++ standard follow all the same pattern. They lock its mutex and its constructor and unlock it in its destructor. This pattern is called RAII.

The following example shows the surprising behavior:

```cpp
1    // myGuard.cpp
2
3    #include <mutex>
4    #include <iostream>
5
6    template <typename T>
7    class MyGuard{
8      T& myMutex;
9      public:
10       MyGuard(T& m):myMutex(m){
11         myMutex.lock();
12             std::cout << "lock" << '\n';
13       }
14       ~MyGuard(){
15             myMutex.unlock();
16         std::cout << "unlock" << '\n';
17       }
18   };
19
20   int main(){
21
22     std::cout << '\n';
23
24     std::mutex m;
25     MyGuard<std::mutex> {m};
26     std::cout << "CRITICAL SECTION" << '\n';
27
28     std::cout << '\n';
29
30   }
```

The `MyGuard` calls lock and unlock in its constructor and its destructor. Because of the temporary, the call to the constructor and destructor happens in line 25. In particular, this means that the destructor's call happens at line 25 and not, as usual, in line 31. As a consequence, the critical section in line 26 is executed without synchronization.

This screenshot of the program shows that the output of *unlock* happens before the output of *CRITICAL SECTION*.

Locks having no name

11.2.2.7 Use `std::lock` or `std::scoped_lock` for locking more mutexes atomically

If a thread needs more than one mutex, you must be extremely careful that you lock the mutex always in the same sequence. If not, a bad interleaving of threads may cause a deadlock.

```cpp
void deadLock(CriticalData& a, CriticalData& b){
  std::lock_guard<std::mutex> guard1(a.mut);
  // some time passes
  std::lock_guard<std::mutex> guard2(b.mut);
  // do something with a and b
}

...

std::thread t1([&]{deadLock(c1,c2);});
std::thread t2([&]{deadLock(c2,c1);});

...
```

Thread `t1` and `t2` need two resources, `CriticalData`, to perform their job. `CriticalData` has its own mutex `mut` to synchronize the access. Unfortunately, both invoke the function `deadlock` with the arguments `c1` and `c2` in a different sequence. Now we have a race condition. If thread `t1` can lock the first mutex `a.mut` but not the second one, `b.mut` because in the meantime thread `t2` locks the second one, we get a deadlock.

Thanks to `std::unique_lock` you can defer the locking of its mutex. The function `std::lock`, which can lock an arbitrary number of mutexes atomically, does the locking.

```
void deadLock(CriticalData& a, CriticalData& b){
  unique_lock<mutex> guard1(a.mut,defer_lock);
  // some time passes
  unique_lock<mutex> guard2(b.mut,defer_lock);
  std::lock(guard1,guard2);
  // do something with a and b
}

...

std::thread t1([&]{deadLock(c1,c2);});
std::thread t2([&]{deadLock(c2,c1);});

...
```

C++17 has a new lock, `std::scoped_lock`, which can get an arbitrary number of mutexes and locks them atomically. Now, the workflow becomes even more straightforward.

```
void deadLock(CriticalData& a, CriticalData& b){
   std::scoped_lock(a.mut, b.mut);
   // do something with a and b
}

...

std::thread t1([&]{deadLock(c1,c2);});
std::thread t2([&]{deadLock(c2,c1);});

...
```

11.2.2.8 Never call unknown code while holding a lock

Calling an `unknownFunction` while holding a mutex is a recipe for undefined behavior.

```
std::mutex m;
{
  std::lock_guard<std::mutex> lockGuard(m);
  sharedVariable= unknownFunction();
}
```

I can only speculate about the `unknownFunction`. If `unknownFunction`

- tries to lock the mutex `m`, which is undefined behavior. Most of the times, you get a deadlock.

- starts a new thread that tries to lock the mutex m, you get a deadlock.
- locks another mutex m2 you may get a deadlock because you lock the two mutexes m and m2 at the same time.
- does not directly or indirectly try to lock the mutex m; all seems to be okay. "Seems" because your coworker can modify the function or the function is dynamically linked, and you get a different version. All bets are open what may happen.
- work as expected you may have a performance problem because you don't know how long the function unknownFunction would take.

To solve this issue, use a local variable.

```
auto tempVar = unknownFunction();
std::mutex m,
{
  std::lock_guard<std::mutex> lockGuard(m);
  sharedVariable = tempVar;
}
```

This additional indirection solves all issues. tempVar is a local variable and can, therefore, not be the victim of a data race. This means that you can invoke unknownFunction without a synchronization mechanism. Additionally, the time for holding a lock is reduced to its bare minimum: assigning the value of tempVar to sharedVariable.

11.2.3 Condition Variables

Synchronizing threads via notifications is a simple concept, but condition variables make this task very challenging. The main reason is that a condition variable has no state.

- If a condition variable gets a notification, it may be the wrong one (spurious wakeup).
- If a condition variable gets its notification before it was ready, the notification is lost (lost wakeup).

11.2.3.1 Don't use condition variables without a predicate

Using a condition variable without a predicate is often a race condition.

Condition variables without a predicate

```cpp
// conditionVariableLostWakeup.cpp

#include <condition_variable>
#include <mutex>
#include <thread>

std::mutex mutex_;
std::condition_variable condVar;

void waitingForWork(){
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck);
    // do the work
}

void setDataReady(){
    condVar.notify_one();
}

int main(){

  std::thread t1(setDataReady);
  std::thread t2(waitingForWork);

  t1.join();
  t2.join();

}
```

If the thread t1 runs before the thread t2, you get a deadlock. t1 sends its notification before t2 can accept it. The notification is lost. This happens very often because thread t1 starts before thread t2, and thread t1 has less work to perform.

Adding a bool variable dataReady to the workflow solves this issue. dataReady also protects against a spurious wakeup because the waiting thread checks at first if the notification was from the right thread.

Condition variables with a predicate

```cpp
1   // conditionVarialbleLostWakeupSolved.cpp
2
3   #include <condition_variable>
4   #include <mutex>
5   #include <thread>
6
7   std::mutex mutex_;
8   std::condition_variable condVar;
9
10  bool dataReady{false};
11
12  void waitingForWork(){
13      std::unique_lock<std::mutex> lck(mutex_);
14      condVar.wait(lck, []{ return dataReady; });
15      // do the work
16  }
17
18  void setDataReady(){
19      {
20          std::lock_guard<std::mutex> lck(mutex_);
21          dataReady = true;
22      }
23      condVar.notify_one();
24  }
25
26  int main(){
27
28    std::thread t1(waitingForWork);
29    std::thread t2(setDataReady);
30
31    t1.join();
32    t2.join();
33
34  }
```

11.2.3.2 Use Promises and Futures instead of Condition Variables

For one-time notifications, promises and futures are the better choice. The workflow of previous program conditioVarialbleLostWakeupSolved.cpp can directly be implemented with a promise and a future.

Notification with promise and future

```
1    // notificationWithPromiseAndFuture.cpp
2
3    #include <future>
4    #include <utility>
5
6    void waitingForWork(std::future<void>&& fut){
7        fut.wait();
8        // do the work
9    }
10
11   void setDataReady(std::promise<void>&& prom){
12       prom.set_value();
13   }
14
15   int main(){
16
17     std::promise<void> sendReady;
18     auto fut = sendReady.get_future();
19
20     std::thread t1(waitingForWork, std::move(fut));
21     std::thread t2(setDataReady, std::move(sendReady));
22
23     t1.join();
24     t2.join();
25
26   }
```

The workflow is reduced to its bare minimum. The promise `prom.set_value()` sends the notification the future `fut.wait()` is waiting for. The program needs no mutexes and locks because there is no critical section. Because no lost wakeup or spurious wakeup can happen, a predicate is also not necessary.

If your workflow requires that you use a condition variable many times, then a promise and future pair is no alternative.

11.2.4 Promises and Futures

Promises and futures an often used as an easy-to-use replacement for threads or condition variables.

11.2.4.1 If possible, go for `std::async`

If possible, you should go for `std::async` to execute an asynchronous task.

```
auto fut = std::async([]{ return 2000 + 11; });
// some time passes
std::cout << "fut.get(): " << fut.get() << '\n';
```

By invoking `auto fut = std::async([]{ return 2000 + 11; })` you say to the C++ runtime: "Run my job". I don't care if it is executed immediately, if it runs on the same thread, if it runs on a thread pool, if it runs on a GPU[9]. You are only interested in picking up the future result: `fut.get()`.

From a conceptional view, a thread is just an implementation detail for running your job. You only specify *what* should be done and not *how* it should be done.

11.3 Memory Model

The foundation of multithreading is a *well-defined* memory model. Having a basic understanding of the memory helps a lot to get a deeper insight into the multithreading challenges.

11.3.1 Don't use volatile for synchronization

In C++ `volatile` has no multithreading semantic in contrast to C# or Java. In C# or Java, `volatile` declares an atomic such as `std::atomic` declares an atomic in C++ and is typically used for objects which can change independently of the regular program flow. Due to this characteristic, no optimized storing in caches takes place.

11.3.2 Don't program Lock Free

This advice sounds ridiculous after writing a book about concurrency and having an entire chapter dedicated to the memory model. The reason for this advice is quite simple. Lock-free programming is very error-prone and requires an expert level in this unique domain. In particular, if you want to implement a lock-free data structure, be aware of the ABA problem.

11.3.3 If you program Lock-Free, use well-established patterns

If you have identified a bottleneck that could benefit from a lock-free solution, apply established patterns.

1. Sharing an atomic boolean or an atomic counter is straightforward.
2. Use a thread-safe or even lock-free container to support consumer/producer scenario. If your container is thread-safe, you can put and get values from the container without worrying about synchronization. You shift the application challenges to the infrastructure.

---

[9]https://en.wikipedia.org/wiki/Graphics_processing_unit

11.3.4 Don't build your abstraction, use guarantees of the language

Thread-safe initialization of a shared variable can be done in various ways. You can rely on guarantees of the C++ runtime such as constant expressions, static variables with block scope, or use the function `std::call_once` in combination with the flag `std::once_flag`. We program in C++; therefore, you can build your abstraction based on atomics using even the highly sophisticated acquire-release semantic. Don't do this in the first place unless you have to do it. This means if you have identified a bottleneck by measuring the performance of a critical code path, only make the change if you know that your handcrafted version outperforms the default guarantees of the language.

11.3.5 Don't reinvent the wheel

Writing thread-safe data structures is quite a challenging endeavor. Writing lock-free data-structures is way harder; therefore, use existing libraries such as Boost.Lockfree[10] or CDS[11].

11.3.5.1 Boost.Lockfree

Boost.Lockfree supports three different data structures:

Queue
> a lock-free multi-produced/multi-consumer queue

Stack
> a lock-free multi-produced/multi-consumer stack

spsc_queue
> a wait-free single-producer/single-consumer queue (commonly known as ring buffer)

11.3.5.2 CDS

CDS stands for Concurrent Data Structures and contains many intrusive (non-owning) and non-intrusive (owning) containers. The containers of the Standard Template Library are non-intrusive because they automatically manage their elements.

- **Stacks** (lock-free)
- **Queues** and **priority-queues** (lock-free)
- **Ordered lists**
- **Ordered sets** and **maps** (lock-free and lock-based)
- **Unordered sets** and **maps** (lock-free and lock-based)

---

[10]http://www.boost.org/doc/libs/1_66_0/doc/html/lockfree.html
[11]http://libcds.sourceforge.net/

Distilled Information

- Concurrent programming is inherently complicated, therefore having in general, but also for multithreading, and the memory model makes a lot of sense.
- A general rule for concurrent programming is, program as constant and as local as possible. Both principles avoid data races by design.
- When possible, prefer tasks about threads and also condition variables. Tasks can deliver values and exceptions and can also send notifications.
- In general, lock-free programming is very challenging and should be the domain of the experts. There are a few exceptions to this rule, such as atomic counters.

Data Structures

12. General Considerations

Cippi analyzes the challenge

12.1 Concurrent Stack

Before I start to write about concurrent data structures, I want to emphasize it explicitly. I could never have written a book about concurrency without the help of previous authors. This statement holds, in particular, true for the chapters about concurrent data structures. My book is, therefore, heavily influenced by the book The Art of Multiprocessing Programmming[1] by Maurice Herlihy and Nir Shavit and by the book C++ Concurrency in Action[2] by Anthony Williams.

When threads share a data structure, and the data structure is mutable, you have to protect the data structure from concurrent access. Conceptually, protection can be done from the outside or from the

---

[1]https://www.oreilly.com/library/view/the-art-of/9780123705914/
[2]https://www.manning.com/books/c-plus-plus-concurrency-in-action-second-edition

inside. From the outside means that it is in the responsibility of the caller (application) to protect the data. This outside perspective is the perspective I mainly used in this book until now. Form inside means that the data structure is responsible for protecting itself. A data structure that protects itself so that no data race can appear is called thread-safe. This inside perspective is the perspective I write about in this and the next chapter.

First of all, what are the general consideration you have to keep in mind to design a concurrent data structure?

Implementing thread-safe data structures is special. Before I dive into each of these unique concerns, here is a concise overview, including the answers you have to give.

- **Locking Strategy**: Should the data structure support coarse-grained, fine-grained locking, or be lock-free? Coarse-grained locking might be easier to implement but introduces contention. A fine-grained implementation or a lock-free one is way more challenging.
- **The Granularity of the Interface**: The bigger the thread-safe data structure's interface, the more difficult it becomes to reason about the concurrent usage of the data structure.
- **Typical Usage Pattern**: When readers use your data structure mainly, you should not optimize for writers.
- **Avoidance of Loopholes**: Don't pass internals of your data structure to clients.
- **Contention**: Do concurrent client requests seldom or often use your data structure?
- **Scalability**: How is your data structure's performance characteristic when the number of concurrent clients increases or the data structure is bounded?
- **Invariants**: Which invariant must hold for your data structure when used?
- **Exceptions**: What should happen if an exception occurs?

Of course, these considerations are dependent on each other. For example, using a coarse-grained locking strategy lets you think about the granularity of the interface and the invariants. In contrast, it may increase the contention on the data structure and breaks scalability.

Although the general considerations in this chapter apply to lock-based data structures and the lock-free data structures, most examples in this chapter use locks. I delay all detailed discussions about lock-free data structures to the dedicated chapter.

12.2 Locking Strategy

Should the data structure support coarse-grained, fine-grained locking or be lock-free? First of all, what do I mean by coarse-grained locking? Coarse-grained locking means that only one thread uses the data structure at one point in time. The thread-safe interface pattern when using one lock is a typical way to implement coarse-grained locking. Here is the straightforward idea of the thread-safe interface pattern.

- All interface member functions (`public`) should use a lock.

- All implementation member functions (`protected` and `private`) must not use a lock.
- The interface member functions call only `protected` or `private` member functions but no `public` member functions.

The thread-safe interface pattern has two nice properties: all public member functions are thread-safe per design and deadlock-free per design. The thread-safe interface is thread-safe because each `public` member function uses a lock and the thread-safe interface is deadlock-free because a `public` member function can not invoke another `public` member function of the class. I assume the following implementation makes my point:

The Thread-Safe Interface

```cpp
1   // threadSafeInterface.cpp
2
3   #include <iostream>
4   #include <mutex>
5   #include <thread>
6
7   class Critical{
8
9   public:
10      void interface1() const {
11          std::lock_guard<std::mutex> lockGuard(mut);
12          implementation1();
13      }
14      void interface2(){
15          std::lock_guard<std::mutex> lockGuard(mut);
16          implementation2();
17          implementation3();
18          implementation1();
19      }
20   private:
21      void implementation1() const {
22          std::cout << "implementation1: "
23                    << std::this_thread::get_id() << '\n';
24      }
25      void implementation2(){
26          std::cout << "    implementation2: "
27                    << std::this_thread::get_id() << '\n';
28      }
29      void implementation3(){
30          std::cout << "        implementation3: "
31                    << std::this_thread::get_id() << '\n';
32      }
33
34
```

```
35    mutable std::mutex mut;
36
37    };
38
39    int main(){
40
41        std::cout << '\n';
42
43        std::thread t1([]{
44            const Critical crit;
45            crit.interface1();
46        });
47
48        std::thread t2([]{
49            Critical crit;
50            crit.interface2();
51            crit.interface1();
52        });
53
54        Critical crit;
55        crit.interface1();
56        crit.interface2();
57
58        t1.join();
59        t2.join();
60
61        std::cout << '\n';
62
63    }
```

The thread-safe interface pattern sounds promising but also has an obvious drawback. The data structure implementing the thread-safe interface is a bottleneck because only one thread can use the data structure at one point in time. This characteristic means if you have many threads working concurrently on the data structure, you should look for more fine-grained locking. For example, instead of protecting the entire interface to a singly-linked list with one lock, you can use a lock on individual nodes of the singly-linked list.

Of course, the data structure can also be lock-free. I discuss the special challenges of lock-free data structures in the same chapter.

12.3 Granularity of the Interface

Assume you want to implement a lock-based wrapper ThreadSafeQueue for a std::deque. The following code snippet gives a rough idea of the ThreadSafeQueue.

```cpp
class ThreadSafeQueue{
    ...
public:
    bool empty() const;
    std::shared_ptr<int> pop();
    ...
private:
    std::deque<int> data;
    ...
};
```

For simplicity reasons, I only display the member functions `empty` and `pop`. `empty` returns if the `ThreadSafeQueue` is empty and `pop` returns and removes the head of the `ThreadSafeQueue`. The interface has the wrong granularity! Why? Assume that two threads want to perform the following function on the same `threadSafeQueue`.

```cpp
ThreadSafeQueue threadSafeQueue;

std::shared_ptr<int> getHead(){
    if (!threadSafeQueue.empty()){
        auto head = threadSafeQueue.pop();
        return head;
    }
    return std::shared_ptr<int>();
}
...
std::thread t1([&]{ auto res = getHead();
                    ...
                });
std::thread t2([&]{ auto res = getHead();
                    ...
                });
```

This code has a race condition which can cause undefined behavior. Between the check `!threadSafeQueue.empty()` that the queue is not empty and the removing of the head-element via `threadSafeQueue.pop()`, there is a time window. For example, the following interleaving can happen.

<div align="center">

Undefined behavior with `ThreadSafeQueue`

</div>

thread t1	thread t2
(1) !threadSafeQueue.empty() == true	
	(2) !threadSafeQueue.empty() == true
(3) auto head = threadSafeQueue.pop();	
	(4) auto head = threadSafeQueue.pop();

If `threadSafeQueue` has only one element, the second call `threadSafeQueue.pop()` from thread `t2` has undefined behavior. Although both member functions are thread-safe, the combination of both member functions has undefined behavior. The interface puts the burden to synchronize the access to `threadSafeQueue` on the client's shoulder. The is far from ideal.

Changing the granularity of the member functions on `ThreadSafeQueue` solves this issue quite elegantly. Just combine the two calls `empty` and `pop` into one member function.

```cpp
class ThreadSafeQueue {
    ...
public:
    std::shared_ptr<int> tryPop(){
        std::lock_guard<std::mutex> queLock(queMute);
        if (!data.empty()){
            auto head = data.pop();
            return head;
        }
        return std::shared_ptr<int>();
    }
    ...
private:
    std::deque<int> data;
    mutable std::lock_mutex queMutex;
    ...
};
```

12.4 Typical Usage Pattern

The typical usage pattern for a data structure is the read access. Reader-writer locks allow you to optimize for the read access. When you put a `std::shared_timed_mutex` into a `std::shared_lock`, the lock becomes a shared lock but when you put a `std::shared_timed_mutex` into a `std::lock_guard` or into a `std::unique_lock`, you get a exclusive lock.

A telephone book is a data structure that more often read than modified and is, therefore, the ideal candidate for a reader-writer lock. Let me start with an exclusive lock to have an initial performance number. I have a telephone book with roughly 89,000 entries. Ten threads read all 89,000 names in an arbitrary order, and one thread adds 1 to a telephone number of each tenth family name. Of course, all threads run at the same time.

The following image shows you a part of the telephone book. You can see the name/number pairs separated by a colon and the name separated from the number by a comma.
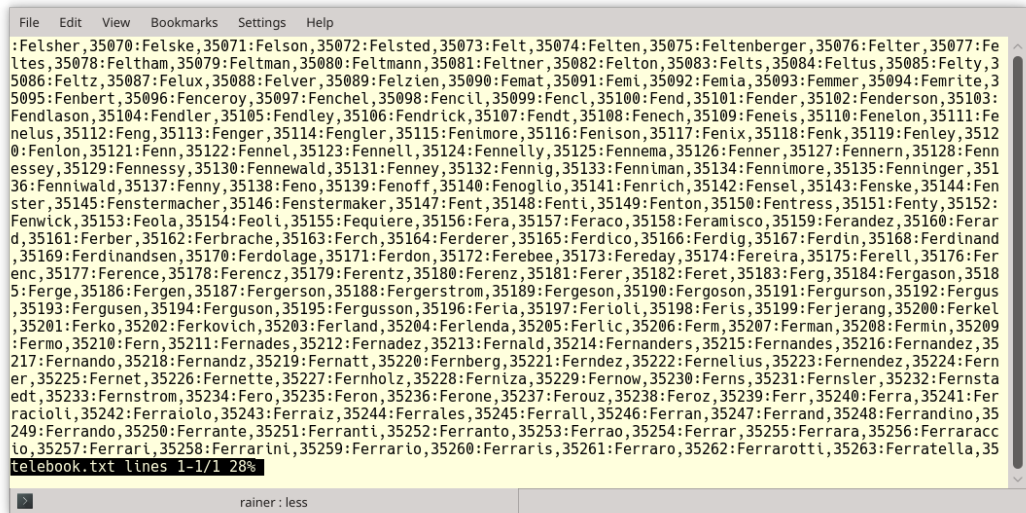
The initial telephone book

The following program crates a `std::unordered_map<std::string, int>` from the file.

Exclusive locking on a telephone book

```cpp
// exclusiveLockingTelebook.cpp

#include <chrono>
#include <fstream>
#include <future>
#include <iostream>
#include <mutex>
#include <random>
#include <regex>
#include <shared_mutex>
#include <sstream>
#include <string>
#include <unordered_map>
#include <vector>

using map = std::unordered_map<std::string, int>;

class TeleBook{

    mutable std::mutex teleBookMutex;
    mutable map teleBook;
    const std::string teleBookFile;
```

```
23
24   public:
25       TeleBook(const std::string& teleBookFile_): teleBookFile(teleBookFile_){
26           auto fileStream = openFile(teleBookFile);
27           auto fileContent = readFile(std::move(fileStream));
28           teleBook = createTeleBook(fileContent);
29           std::cout << "teleBook.size(): " <<  teleBook.size() << '\n';
30       }
31
32       map get() const {
33           std::lock_guard<std::mutex> lockTele(teleBookMutex);
34           return teleBook;
35       }
36
37       int getNumber(const std::string& name) const {
38           std::lock_guard<std::mutex> lockTele(teleBookMutex);
39           return teleBook[name];
40       }
41
42       void setNewNumber(const std::string& name) {
43           std::lock_guard<std::mutex> lockTele(teleBookMutex);
44           teleBook[name]++;
45       }
46
47   private:
48
49       std::ifstream openFile(const std::string& myFile){
50           std::ifstream file(myFile, std::ios::in);
51           if ( !file ){
52               std::cerr << "Can't open file "+ myFile + "!" << '\n';
53               exit(EXIT_FAILURE);
54           }
55       return file;
56   }
57
58       std::string readFile(std::ifstream file){
59           std::stringstream buffer;
60           buffer << file.rdbuf();
61           return buffer.str();
62   }
63
64       map createTeleBook(const std::string& fileCont){
65           map teleBook;
66
67           std::regex regColon(":");
```

```
68          std::sregex_token_iterator fileContIt(fileCont.begin(), fileCont.end(),
69                                       regColon, -1);
70          const std::sregex_token_iterator fileContEndIt;
71
72          std::string entry;
73          std::string key;
74          int value;
75          while (fileContIt != fileContEndIt){
76              entry = *fileContIt++;
77              auto comma = entry.find(",");
78              key = entry.substr(0, comma);
79              value = std::stoi(entry.substr(comma + 1, entry.length() -1));
80              teleBook[key] = value;
81          }
82          return teleBook;
83      }
84  };
85
86  std::vector<std::string> getRandomNames(const map& teleBook){
87
88      std::vector<std::string> allNames;
89      for (const auto& pair: teleBook) allNames.push_back(pair.first);
90
91      std::random_device randDev;
92      std::mt19937 generator(randDev());
93
94      std::shuffle(allNames.begin(), allNames.end(), generator);
95
96      return allNames;
97  }
98
99  void getNumbers(const std::vector<std::string>& randomNames, TeleBook& teleBook){
100     for (const auto& name: randomNames) teleBook.getNumber(name);
101 }
102
103 int main(){
104
105     std::cout << '\n';
106
107     // get the filename
108     const std::string myFileName = "tele.txt";
109     TeleBook teleBook(myFileName);
110
111     std::vector<std::string> allNames = getRandomNames(teleBook.get());
112     std::vector<std::string> tenthOfAllNames(allNames.begin(),
```

```
113                                              allNames.begin() + allNames.size()/10);
114
115      auto start = std::chrono::steady_clock::now();
116
117      auto futReader0 = std::async(std::launch::async,
118                                      [&]{ getNumbers(allNames, teleBook); });
119      auto futReader1 = std::async(std::launch::async,
120                                      [&]{ getNumbers(allNames, teleBook); });
121      auto futReader2 = std::async(std::launch::async,
122                                      [&]{ getNumbers(allNames, teleBook); });
123      auto futReader3 = std::async(std::launch::async,
124                                      [&]{ getNumbers(allNames, teleBook); });
125      auto futReader4 = std::async(std::launch::async,
126                                      [&]{ getNumbers(allNames, teleBook); });
127      auto futReader5 = std::async(std::launch::async,
128                                      [&]{ getNumbers(allNames, teleBook); });
129      auto futReader6 = std::async(std::launch::async,
130                                      [&]{ getNumbers(allNames, teleBook); });
131      auto futReader7 = std::async(std::launch::async,
132                                      [&]{ getNumbers(allNames, teleBook); });
133      auto futReader8 = std::async(std::launch::async,
134                                      [&]{ getNumbers(allNames, teleBook); });
135      auto futReader9 = std::async(std::launch::async,
136                                      [&]{ getNumbers(allNames, teleBook); });
137
138      auto futWriter = std::async(std::launch::async, [&]{
139          for (const auto& name: tenthOfAllNames) teleBook.setNewNumber(name);
140      });
141
142      futReader0.get(), futReader1.get(), futReader2.get(), futReader3.get(),
143      futReader4.get(), futReader5.get(), futReader6.get(), futReader7.get(),
144      futReader8.get(), futReader9.get(), futWriter.get();
145
146      std::chrono::duration<double> duration = std::chrono::steady_clock::now()
147                                              - start;
148
149      std::cout << "Process time: " << duration.count() << " seconds" << '\n';
150
151      std::cout << '\n';
152
153  }
```

Let me start with the constructor of the class TeleBook (lines 25 - 30). It opens the file, reads the content, and creates a telephone book. The function getRandomNames (lines 86 - 97) generates an arbitrary

permutation of the family names. Each tenth family name goes to the std::vector tenthOfAllNames. These are the family names for which the telephone number is modified. Now, to the most interesting lines 117 - 149. futReader0 to futReader9 are the futures, representing ten threads. Each thread reads all family names using the function getNumbers in lines 99 - 101. The updating Future futWriter performs it job directly in the lambda function (lines 138 - 140). When all futures are done, line 149 displays the overall process time. I want to mention explicitly, that the interface functions (get, getNumber, and setNewNumber) of TeleBook use the std::mutex teleBookMutex (line 20) for the synchronization. teleBookMuex is mutable and can, therefore, be used in a const member function.

Now, I come to the optimization step. The member functions get (lines 32 - 35) and getNumber (lines 37 - 40) don't modify the telebook and can, therefore, use a reader-writer lock. Of course, this optimization can not be applied to the member function setNewNumber (lines 42 - 45). I only display the interface of the new class TeleBook in the optimized program for simplicity. The other parts of the program are identical.

Shared locking on a telephone book

```cpp
// sharedLockingTelebook.cpp

...

class TeleBook{

    mutable std::shared_timed_mutex teleBookMutex;
    mutable map teleBook;
    const std::string teleBookFile;

public:
    TeleBook(const std::string& teleBookFile_): teleBookFile(teleBookFile_){
        auto fileStream = openFile(teleBookFile);
        auto fileContent = readFile(std::move(fileStream));
        teleBook = createTeleBook(fileContent);
        std::cout << "teleBook.size(): " << teleBook.size() << '\n';
    }

    map get() const {
        std::shared_lock<std::shared_timed_mutex> lockTele(teleBookMutex);
        return teleBook;
    }

    int getNumber(const std::string& name) const {
        std::shared_lock<std::shared_timed_mutex> lockTele(teleBookMutex);
        return teleBook[name];
    }

    void setNewNumber(const std::string& name) {
```

```
30              std::lock_guard<std::shared_timed_mutex> lockTele(teleBookMutex);
31              teleBook[name]++;
32          }
33
34      private:
35          ...
```

You should not compare the performance of Linux(GCC) with the performance of Windows(cl.exe) because the underlying computers are not comparable. Instead, you should compare the relative performance of exclusive versus shared locking on both platforms. The numbers are pretty astonishing.

12.4.1 Linux (GCC)
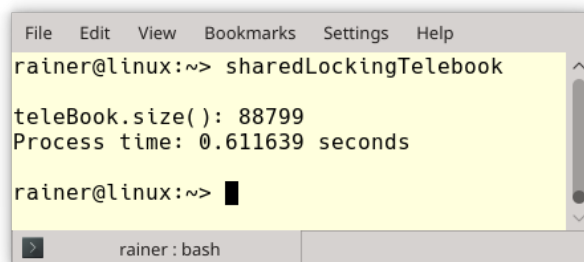
12.4.1.1 Exclusive Locking

The initial telephone book

12.4.1.2 Shared Locking

The initial telephone book

Exclusive locking is on Linux about 15 % slower than shared locking. This difference is less than I expected because the read/write ratio is 100 to 1.

12.4.2 Windows (cl.exe)

12.4.2.1 Exclusive Locking
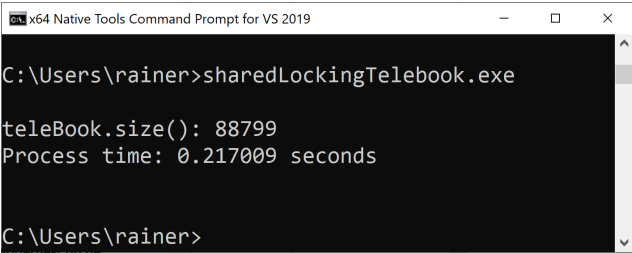
The initial telephone book

12.4.2.2 Shared Locking

The initial telephone book

Honestly, the Windows numbers surprised me because shared locking is on Windows two times slower than exclusive locking. It seems that the usage of a `std::shared_lock` and a `std::lock_guard` together with a `std::shared_time_mutex` is a heavyweight operation so that sharing does not pay off.

12.5 Avoidance of Loopholes

Don't pass the internals of your data structure to clients. Internals can be passed by reference or pointer to the outside world. Passing an arbitrary callable to the data structure opens a loophole, which is difficult challenging to spot.

Loophole in the interface

```
1    // lockDouble.cpp
2
3    #include <future>
4    #include <iostream>
5    #include <mutex>
6
7    class LockDouble {
8        public:
9            double get() const {
10               std::lock_guard<std::mutex> lockDoubGuard(lockDoubMutex);
11               return lockDoub;
12           }
13
14           void set(double val) {
15               std::lock_guard<std::mutex> lockDoubGuard(lockDoubMutex);
16               lockDoub = val;
17           }
18
19           template <typename Func>
20           void apply(Func func){
21               std::lock_guard<std::mutex> lockDoubGuard(lockDoubMutex);
22               func(lockDoub);
23           }
24
25
26       private:
27           double lockDoub{};
28           mutable std::mutex lockDoubMutex;
29
30   };
31
32   int main(){
33
34       LockDouble lck1;
35
36       auto fut1 = std::async([&lck1]{ lck1.set(20.11); });
37       auto fut2 = std::async([&lck1]{ std::cout << lck1.get() << '\n'; });
38
39       double* loophole = nullptr;
40       lck1.apply([&loophole](double& d) mutable { loophole = &d; });
41       *loophole = 11.22;
42
43       auto fut3 = std::async([&lck1]{ std::cout << lck1.get() << '\n'; });
```
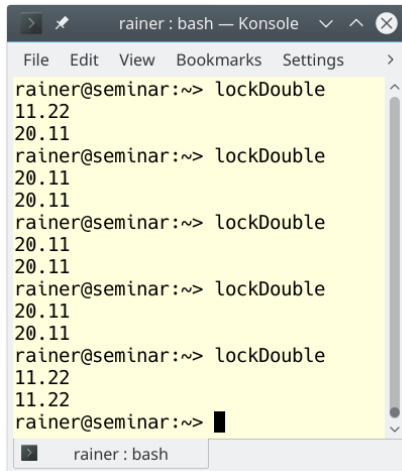
```
44
45   }
```

The class `LockDouble` has a clean interface. Each access to the variable `lockDouble` is protected by the same mutex `lockDoubMutex` put into a `std::lock_guard`. The member function `get` returns a copy and not a non-const reference to `lockDouble`. If the member function `get` would return a non-const reference to `lockDoub`, a client could quite easily produce a data race.

```
...
double& get() {
    std::lock_guard<std::mutex> lockDoubGuard(lockDoubMutex);
    return lockDoub;
}
...
LockDouble lck;
lck.set(22.11);
double& d = lck.get();
d = 11.22;
```

Of course, the issue is that the reference `d` can change `lockDoub`, which should be protected by the mutex `lockDoubMutex`. I assume this issue was quite easy to detect.
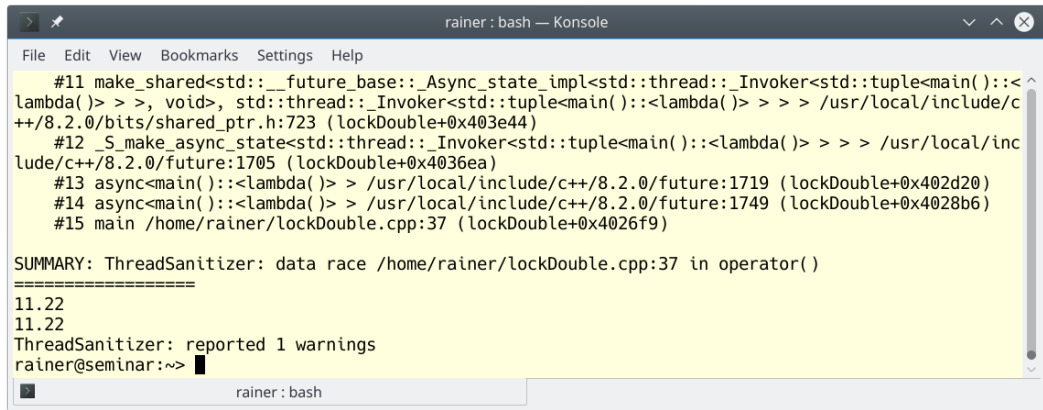
The member function `apply` in lines 19 - 23 opens the loophole. I create in line 40 a lambda function that returns a reference to `lockDoub`. The expression `*loophole = 11.22` changes the value of `lockDoub` without synchronization. Of course, this is a data race. The screenshot shows the effects of the non-synchronized access to `lockDoub`.

Sychronized and non-synchronized access to `lockDoub`

ThreadSanitizer[3] shows the data race explicitly.

Data race detection with the Threadsanitizer

12.6 Contention

Do concurrent client requests seldom or often use your data structure? When the contention is low, straightforward synchronization primitives such as locks are usually fast enough. Using sophisticated and challenging solutions such as atomics could be overkill. Before you go for the advanced solutions, you should measure. To get an idea, how expensive locks are, let me make a straightforward test.

Now, I can make it short. I already did this test in the chapter Calculating the Sum of a Vector. I filled a `std::vector` with one hundred million arbitrary but uniformly distributed[4] numbers between 1 and 10. Then I calculated the sum of the numbers in various ways. Two ways are, in particular, interesting for this section.

12.6.1 Single-Threaded Summation without Synchronization

First, I show the straightforward range-based for loop to calculate the sum.

Summation of a vector in a range-based for loop

```cpp
// calculateWithLoop.cpp

#include <chrono>
#include <iostream>
#include <random>
#include <vector>

constexpr long long size = 100000000;

int main(){

    std::cout << '\n';

    std::vector<int> randValues;
    randValues.reserve(size);

    // random values
    std::random_device seed;
    std::mt19937 engine(seed());
    std::uniform_int_distribution<> uniformDist(1, 10);
    for (long long i = 0 ; i < size ; ++i)
        randValues.push_back(uniformDist(engine));

    const auto sta = std::chrono::steady_clock::now();

    unsigned long long sum = {};
    for (auto n: randValues) sum += n;

    const std::chrono::duration<double> dur =
          std::chrono::steady_clock::now() - sta;

    std::cout << "Time for addition " << dur.count()
              << " seconds" << '\n';
    std::cout << "Result: " << sum << '\n';

    std::cout << '\n';

}
```

The reference number for Linux

Explicit summation on Linux

and for Windows.

Explicit summation on Windows

12.6.2 Single-Threaded Summation with Synchronization (lock)

In contrast, the range-based for-loop with additional synchronization. I show only the source code difference to the non-synchronized version.

Summation of a vector by using a lock for the summation variable
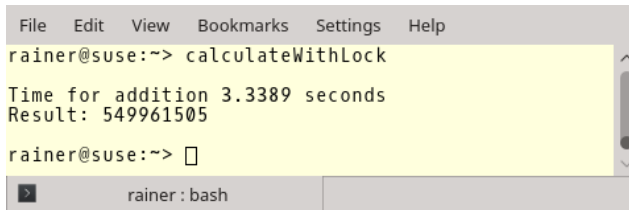
```cpp
// calculateWithLock.cpp

...

std::mutex myMutex;

for (auto i: randValues){
  std::lock_guard<std::mutex> myLockGuard(myMutex);
  sum += i;
}

...
```

Respectively, the performance numbers for Linux
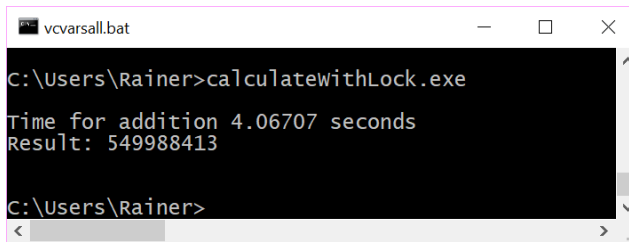
Single-threaded summation on Linux using a lock

and Windows.

Single-threaded summation on Windows using a lock

12.6.3 Single-Threaded Summation with Synchronization (atomic)

The following range-based for-loop uses and atomic for synchronization. Here are the essential lines showing the difference to the non-synchronized version.

Summation of a vector by using an atomic for the summation variable
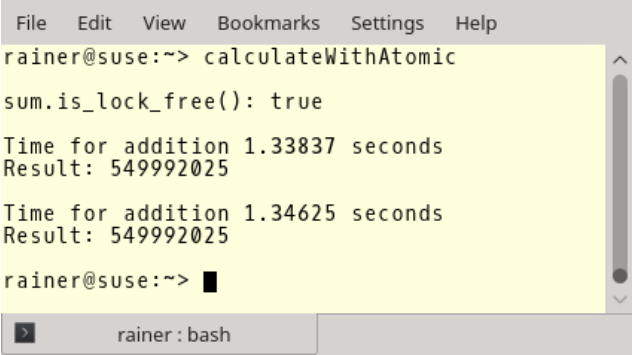
```cpp
// calculateWithAtomic.cpp

...

std::atomic<unsigned long long> sum = {};

for (auto i: randValues) sum += i;

...
```

Respectively, the performance numbers for Linux
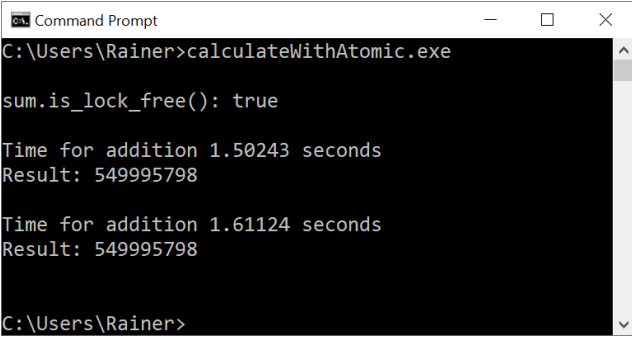
Single-threaded summation on Linux using an atomic

and Windows.

Single-threaded summation on Windows using an atomic

12.6.4 The Comparison

The non-synchronized version is about 18 times faster than the version using lock and about 50 - 150 times faster than the version using locks. The synchronized version using atomics is about 2.5 times faster than the version using locks. This difference holds for Linux and Windows. The performance numbers seem to speak a clear statement against synchronization with locks or atomics, but you have to consider that this was a synchronization-heavy job. Additionally, when you have to synchronize only a few times, a lock or a atomic may be your best and sufficiently performant version.

12.7 Scalability

How is your data structure's performance characteristic when the number of concurrent clients increases or the data structure is bounded? These are two questions which you should have to answer. Scalability means a 1 to 1 relation between clients of the data structure and the throughput.

- For example, when you have a thread-safe queue with one producer/consumer, each additional producer or consumer must be blocked until the previous one is done. This blocking restriction does not hold for many producers/many consumers queue. For simplicity reasons, I call the second scenario an n/m relation. n can also be 1. The second scenario is if the producer can satisfy all consumers or the other way around. If not, producers or consumers have to wait, which hinters scalability.

- When your data structure is a bounded thread-safe queue, you can not expect perfect scalability because the producer/consumer fails at one point out of lockstep. An internally used puffer between the producers and consumers may decouple them but will not solve the original issue.

Let me answer the two questions for the concrete `ThreadSafeQueue` from chapter Concurrent Architecture.

The Monitor Object

```
template <typename T>
class Monitor{
public:
    void lock() const {
        monitMutex.lock();
    }
    void unlock() const {
        monitMutex.unlock();
    }

    void notify_one() const noexcept {
        monitCond.notify_one();
    }
    void wait() const {
        std::unique_lock<std::recursive_mutex> monitLock(monitMutex);
        monitCond.wait(monitLock);
    }

private:
    mutable std::recursive_mutex monitMutex;
    mutable std::condition_variable_any monitCond;
};

template <typename T>
class ThreadSafeQueue: public Monitor<ThreadSafeQueue<T>>{
public:
    void add(T val){
        derived.lock();
        myQueue.push(val);
        derived.unlock();
        derived.notify_one();
```

```
32          }
33
34      T get(){
35          derived.lock();
36          while (myQueue.empty()) derived.wait();
37          auto val = myQueue.front();
38          myQueue.pop();
39          derived.unlock();
40          return val;
41      }
42  private:
43      std::queue<T> myQueue;
44      ThreadSafeQueue<T>& derived = static_cast<ThreadSafeQueue<T>&>(*this);
45  };
```

The member function `add` adds (line 27) an element of type `T` to the `std::queue`, and the member function `get` (line 34) removes an element from the `std::queue`. The class `ThreadSafeQueue` implements the Monitor Object. The Monitor Object design pattern synchronizes concurrent member function execution to ensure that only one member function at a time runs within an object. When a producer is done, one consumer will be notified by the `std::condition_variable_any` (line 21). A `std::recursive_-mutex` (line 20) protects modification on an internal queue. This description should be sufficient to answer our two questions. If you want to know more about this architecture pattern, read the previous chapter to the Monitor Object.

- It's pretty evident that the consumer can be blocked in line 36, if no values are available.
- The second question is straightforward to answer because the thread-safe queue is not bounded.

12.8 Invariants

An invariant is a condition or relation that is always true. For example, the sum of all credit and debts for all accounts should be zero at any time. Should, because, for the following program, the invariant will not hold.

Broken Invariants

```
1  // invariant.cpp
2
3  #include <functional>
4  #include <iostream>
5  #include <mutex>
6  #include <numeric>
7  #include <random>
8  #include <thread>
```

```
 9    #include <vector>
10
11    class Accounts{
12    public:
13        void deposit(int account){
14            std::lock_guard<std::mutex> lockAcc(mutAcc);
15            accounts[account] += 10;
16        }
17
18        void takeOff(int account){
19            std::lock_guard<std::mutex> lockAcc(mutAcc);
20            accounts[account] -= 10;
21        }
22
23        int getSum() const {
24            std::lock_guard<std::mutex> lockAcc(mutAcc);
25            return std::accumulate(accounts.begin(), accounts.end(), 0);
26        }
27
28    private:
29        std::vector<int> accounts = std::vector<int>(100, 0);
30        mutable std::mutex mutAcc;
31    };
32
33    class Dice{
34    public:
35        int operator()(){ return rand(); }
36    private:
37        std::function<int()> rand = std::bind(std::uniform_int_distribution<>(0, 99),
38                                              std::default_random_engine());
39    };
40
41    using namespace std::chrono_literals;
42
43    int main(){
44
45        constexpr auto TRANS = 1000;
46        constexpr auto OBS = 10;
47        Accounts acc;
48        Dice dice;
49
50        std::vector<std::thread> transactions(TRANS);
51        for (auto& thr: transactions) thr = std::thread([&acc, &dice]{
52            acc.deposit(dice());
53            std::this_thread::sleep_for(10ns);
```
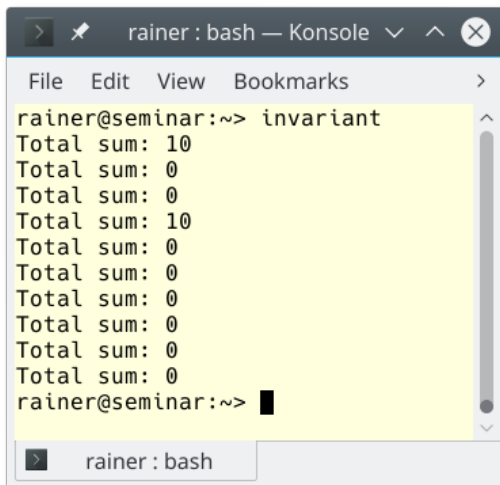
```
54          acc.takeOff(dice())); }
55      );
56
57      std::mutex coutMutex;
58
59      std::vector<std::thread> observers(OBS);
60      for (auto& thr: observers) thr = std::thread([&acc, &coutMutex]{
61          std::lock_guard<std::mutex> coutLock(coutMutex);
62          std::this_thread::sleep_for(1ms);
63          std::cout << "Total sum: " << acc.getSum() << '\n'; }
64      );
65
66      for (auto& thr: transactions) thr.join();
67      for (auto& thr: observers) thr.join();
68  }
```

The class Accounts has 100 accounts, all initialized to zero (line 24). You can deposit 10 unit (lines 13 - 16) from an account given by the index. The member function takeOff (lines 18 - 21) allows it to withdraw 10 units from a given account. The member function getSum (lines 23 - 26) helps to check if all invariants hold. All three member functions are synchronized via the std::mutex mutAcc. Now, to the provocation of the invariants. I create 1000 threads (lines 50 - 56) which adds (line 52) and remove (54) 10 units to and from an arbitrary account which is given by the dice. In contrast, 10 threads observe the sum of all accounts (lines 59 - 64). The longer the sleeping between the deposit and the takeOff call is (line 53), the more probable is it to observe the invariant such as in the following screenshot.

Broken invariants caught in action

Of course, putting the `deposit` and the `takeOff` call into one critical section would guarantee the invariant.

12.9 Exceptions

What should happen if an exception occurs? The answer to this question depends on the data structure you use to create the thread. Here are a few options?

- `std::thread`: When the created thread throws an exception, `std::terminate` is called. This means that `std::terminate` passes the `main`-thread.
- Task: Tasks such as `std::async`, `std::packaged_task`, and `std::promise` can throw an exception which has to be handled by the associated `std::future` .
- Parallel algorithms of the STL: If an exception occurs during the usage of an algorithm with an execution policy, `std::terminate` is called.

`std::terminate` calls the installed `std::terminate_handler`[5]. The consequence is that per default `std::abort`[6] is called, which causes abnormal program termination.

Now, it's time to consider these concerns in practice. Typically concurrent data structures are stacks and queues.

Distilled Information

- There many question you have to answer before you design lock-based or lock-free data structures.
-
- Should data data structure support coarse-grained, fine-grained locking, or even be lock-free?
- What are the typical usage patterns of your concurrent data structure? Do you want to optimize the data structure for read or write operations or for high contention?

---

[5]https://en.cppreference.com/w/cpp/error/terminate_handler
[6]https://en.cppreference.com/w/cpp/utility/program/abort

13. Lock-Based Data Structures
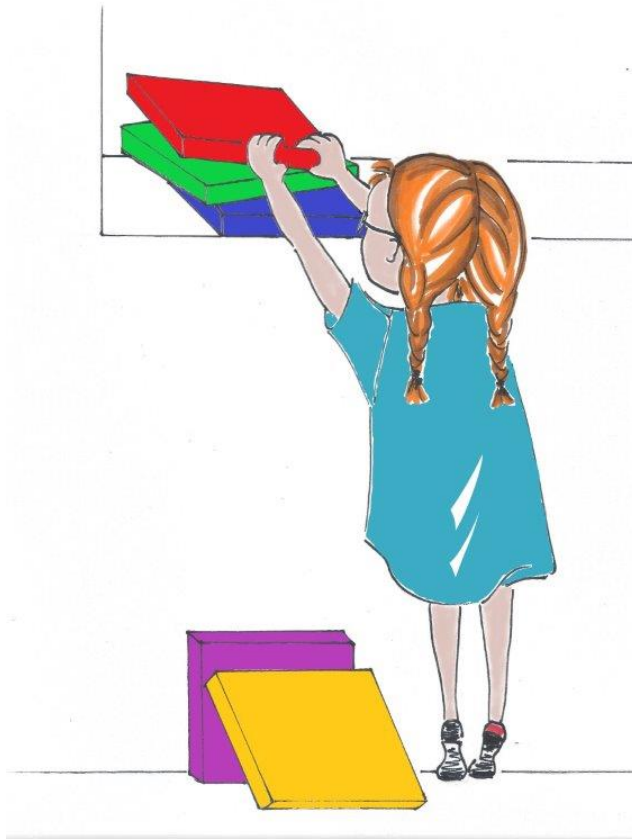
Cippi builds a stack

First of all: What is a stack?

13.0.1 A Stack
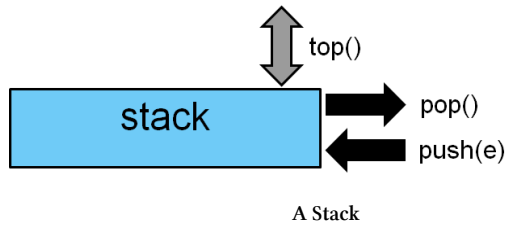
A Stack

A `std::stack`[1] follows the LIFO principle (**L**ast **I**n **F**irst **O**ut). A stack `sta`, which needs the header `<stack>`, has three member functions.

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.

```cpp
#include <stack>
...
std::stack<int> myStack;

std::cout << myStack.empty() << '\n';    // true
std::cout << myStack.size() << '\n';     // 0

myStack.push(1);
myStack.push(2);
myStack.push(3);
std::cout << myStack.top() << '\n';      // 3

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

std::cout << myStack.empty() << '\n';    // true
std::cout << myStack.size() << '\n';     // 0
```

Let me implement the concurrent stack successively.

13.0.1.1 A Simplified Implementation

My first implementation only supports the `push` member function. The class `ConcurrentStackPush` is a thin wrapper around a `std::stack`.

---

[1]http://en.cppreference.com/w/cpp/container/stack

A concurrent Stack supporting push

```cpp
1   // concurrentStackPush.cpp
2
3   #include <list>
4   #include <mutex>
5   #include <stack>
6   #include <string>
7   #include <utility>
8   #include <vector>
9
10  template<typename T, template <typename, typename> class Cont = std::deque>
11  class ConcurrentStackPush{
12  public:
13      void push(T val){
14          std::lock_guard<std::mutex> lockStack(mutexStack);
15          myStack.push(std::move(val));
16      }
17      ConcurrentStackPush() = default;
18      ConcurrentStackPush(const ConcurrentStackPush&) = delete;
19      ConcurrentStackPush& operator= (const ConcurrentStackPush&) = delete;
20  private:
21      mutable std::mutex mutexStack;
22      std::stack<T, Cont<T, std::allocator<T>>> myStack;
23  };
24
25  int main(){
26
27      ConcurrentStackPush<int> conStack;
28      conStack.push(5);
29
30      ConcurrentStackPush<double, std::vector> conStack2;
31      conStack2.push(5.5);
32
33      ConcurrentStackPush<std::string, std::list> conStack3;
34      conStack3.push("hello");
35
36  }
```

From the concurrency perspective, the type ConcurrentStackPush supports the push member function (lines 13 - 16), which copies a new element to the internal myStack (line 22). Thanks to the mutexStack in line 21, the copy-operation is thread-safe. I assume, you are irritated by the second template parameter: template <typename, typename> class Cont = std::deque. The second template parameter is a so-called template-template parameter. It's default is std::deque. Cont is the container for holding the elements. Cont needs the type and the allocator of its arguments. In line 22, you see the usage of the

parameter `Cont`. `Cont` is the second argument of the internal `std::stack`. The lines 27, 30, and 33 show how you can instantiate a `ConcurrentStackPush` with various containers. `conStack` (line 27) uses a `std::deque`. `contStack2` (line 30) uses a `std::vector`, and `contStack2` (line 33) uses a `std::list`.

`std::stack`[2] and the following `std::queue`[3] are so-called container adapter because they use an existing container and support a stack-like or queue-like interface.

You may ask yourself why I use the `ConcurrentStackPush` in a single-threaded environment and why I don't show any output. Here is my answer: just having one member function, `push`, is too restricting. `ConcurrentStackPush` should only serve as a starting point for a complete Implementation.

13.0.1.2 A Complete Implementation

According to the info box stack, my concurrent stack should at least support the member functions `push`, `pop`, and `top`.

You may assume that the straightforward extension of the previous type `ConcurrentStackPush` would do the job. This assumption is wrong.

A broken Concurrent Stack

```cpp
template<typename T, template <typename, typename> class Cont = std::deque>
class ConcurrentStackBroken {
public:
    void push(T val) {
        std::lock_guard<std::mutex> lockStack(mutexStack);
        myStack.push(std::move(val));
    }
    void pop() {
        std::lock_guard<std::mutex> lockStack(mutexStack);
        myStack.pop();
    }
    T& top(){
        std::lock_guard<std::mutex> lockStack(mutexStack);
        return myStack.top();
    }
    ConcurrentStackBroken() = default;
    ConcurrentStackBroken(const ConcurrentStackBroken&) = delete;
    ConcurrentStackBroken& operator = (const ConcurrentStackBroken&) = delete;
private:
    mutable std::mutex mutexStack;
    std::stack<T, Cont<T, std::allocator<T>>> myStack;
};
```

---

[2]http://en.cppreference.com/w/cpp/container/stack
[3]http://en.cppreference.com/w/cpp/container/queue

The `ConcurrentStackBroken` has the member function `push` the two member functions `pop`, and `top`. Just using the same mutex `mutexStack` for all three member functions is not correct. This native implementation has at least two issues. One is obvious; the second one is more involved.

First, the member function `top` returns a reference. The reference can be used to modify an element which is a data race, such as in the following code snippet.

Returning a reference

```cpp
int main() {

    ConcurrentStackBroken<int> conStack;
    conStack.push(5);

    auto fut = std::async(std::launch::async, [&conStack]{ conStack.top() += 5; });
    auto fut2 = std::async(std::launch::async,
                            [&conStack]{ std::cout << conStack.top() << '\n'; });

}
```

The issue is that the write operation of the first `std::async` is not synchronized with the read operation of the second `std::async`.

The second issue is that the `top` member function followed by the `pop` member function is not an atomic operation. Let's see what I mean.

`top` and `pop` are not atomic

```cpp
int main(){

    ConcurrentStackBroken<int> conStack;
    constexpr auto SENTINEL = std::numeric_limits<int>::min();
    conStack.push(SENTINEL);
    conStack.push(5);

    auto saveRemove = [&conStack]{ if (conStack.top() != SENTINEL) conStack.pop(); };

    auto fut = std::async(std::launch::async, saveRemove);
    auto fut2 = std::async(std::launch::async, saveRemove);
    auto fut3 = std::async(std::launch::async, saveRemove);

}
```

To ensure that the `ConcurrentStackBroken` always has a valid element before I `pop` it, I pushed a `SENTINEL` on it. `SENTINEL` is the invariant that should hold for the concurrent data structure. The lambda-function `saveRemove` applies my protocol. The issue in `saveRemove` is that more than one `pop` operation

could happen in sequence, based on the wrong assumption that `conStack` has a valid element. Of course, the three `std::async` all interleave, and, therefore, the following sequence of operations can happen. The operations `top` and `pop` do not interleave because they use the same mutex.

A possible interleaving

```
conStack.top() // associated promise to `fut`
conStack.top() // associated promise to `fut2`
conStack.top() // associated promise to `fut3`
conStack.pop() // associated promise to `fut`
conStack.pop() // associated promise to `fut2`     // (2)
conStack.pop() // associated promise to `fut3`     // (3)
```

The constructed sequence of operations is fatal. With the call (2), the SENTINEL is removed, and, therefore, the invariant is broken. The call (3) has undefined behavior because the `pop` call on the `std::stack` triggers a `pop_back` call on the internal container. Calling `pop_back` on an empty container is undefined behavior.

Changing the interface of a concurrent data structure is often a viable way to overcome concurrency issues. In this case, I change the interface's granularity and combine the member functions `top` and `pop` into on member function `topAndPop`. Of course, this combination of member functions `top` and `pop` violates the Single responsibility principle[4].

A concurrent Stack

```cpp
1   // concurrentStack.cpp
2
3   #include <future>
4   #include <limits>
5   #include <iostream>
6   #include <mutex>
7   #include <stack>
8   #include <stdexcept>
9   #include <utility>
10
11  template<typename T, template <typename, typename> class Cont = std::deque>
12  class ConcurrentStack {
13  public:
14      void push(T val) {
15          std::lock_guard<std::mutex> lockStack(mutexStack);
16          myStack.push(std::move(val));
17      }
18      T topAndPop() {
19          std::lock_guard<std::mutex> lockStack(mutexStack);
20          if ( myStack.empty() ) throw std::out_of_range("The stack is empty!");
```

---

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

```
21              auto val = myStack.top();
22              myStack.pop();
23              return val;
24          }
25      ConcurrentStack() = default;
26      ConcurrentStack(const ConcurrentStack&) = delete;
27      ConcurrentStack& operator = (const ConcurrentStack&) = delete;
28  private:
29      mutable std::mutex mutexStack;
30      std::stack<T, Cont<T, std::allocator<T>>> myStack;
31  };
32
33  int main() {
34
35      ConcurrentStack<int> conStack;
36
37      auto fut = std::async([&conStack]{ conStack.push(2011); });
38      auto fut1 = std::async([&conStack]{ conStack.push(2014); });
39      auto fut2 = std::async([&conStack]{ conStack.push(2017); });
40
41      auto fut3 = std::async([&conStack]{ return conStack.topAndPop(); });
42      auto fut4 = std::async([&conStack]{ return conStack.topAndPop(); });
43      auto fut5 = std::async([&conStack]{ return conStack.topAndPop(); });
44
45      fut.get(), fut1.get(), fut2.get();
46
47      std::cout << fut3.get() << '\n';
48      std::cout << fut4.get() << '\n';
49      std::cout << fut5.get() << '\n';
50
51  }
```

The member function `topAndPop` (lines 18 - 24) returns a copy instead of a reference such as the member function `top` before. Calling `pop` on an empty container is undefined behavior. I prefer to throw a `std::out_of_range` exception (line 20) if the stack is empty. Returning a special non-value or returning a `std::optional`[5] is also a valid option. Copying the value has a downside. If the copy constructor of the value throws an exception such as `std::bad_alloc`[6], the value is lost.

The calls `fut.get()`, `fut1.get()`, `fut2.get()` (line 45) ensure that the associated promise runs. If you don't specify the launch policy, the promise may run lazily in the caller's thread. Lazily means that the promise will execute if and only if the future asks for its result with `get` or `wait`. Launching the promises in a separate thread is also a valid option:

---

[5]https://en.cppreference.com/w/cpp/utility/optional
[6]https://en.cppreference.com/w/cpp/memory/new/bad_alloc

Launching each promise in a separate thread

```
auto fut = std::async(std::launch::asnyc, [&conStack]{ conStack.push(2011); });
auto fut1 = std::async(std::launch::asnyc, [&conStack]{ conStack.push(2014); });
auto fut2 = std::async(std::launch::asnyc, [&conStack]{ conStack.push(2017); });
```

Finally, the output of the program:

A concurrent stack

13.1 Concurrent Queue

Accordingly to the chapter to the Concurrent Stack I want to answer the question: What is a queue?

13.1.1 A Queue
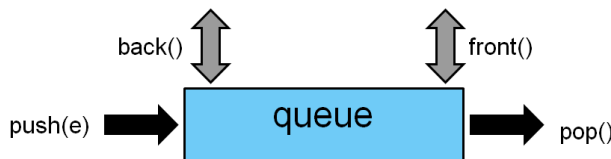
A Queue

A `std::queue`[7] follows the FIFO principle (**F**irst **I**n **F**irst **O**ut). A queue `que`, which needs the header `<queue>`, has four member functions.

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.

---

[7]http://en.cppreference.com/w/cpp/container/queue

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

std::cout << myQueue.empty() << '\n';     // true
std::cout << myQueue.size() << '\n';      // 0

myQueue.push(1);
myQueue.push(2);
myQueue.push(3);
std::cout << myQueue.back() << '\n';      // 3
std::cout << myQueue.front() << '\n';     // 1

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

std::cout << myQueue.empty() << '\n';     // true
std::cout << myQueue.size() << '\n';      // 0
```

Based on my discussion of the concurrent stack, the concurrent queue's first implementation is quite similar.

13.1.2 Coarse-Grained Locking

Let's start straightforward. My first implementation combines the `front` and `pop` member function into a member function `frontAndPop`. In contrast, the member function `push` adds the elements to the end of the queue. The member function `back`, which is optional for a queue[8], returns the last element of the queue. Honestly, I'm biased if I should support the member function `back`. Here are my thoughts.

1. Not supporting `back` limits the interface because you can never ask for the last element.

2. Combine `back` and `push` into one member function `backAndPush`. `backAndPush` should, in this case, return to the `push` operation previous value. This combination seems to be promising but has two serious issues. First, the previous value has to be copied because a reference or a pointer would break the synchronization. This copy operation is a performance penalty. Second, the copy constructor could throw an exception.

3. Supporting both member functions `back` and `push` separately, introduces a data race. Assume a user makes an assumption based on the value of the last element. We have the same issue, such as supporting `front` and `pop` as two different member functions. The argumentation to the member functions `top` and `pop` in concurrent stack also apply here. Honestly, this usage of the concurrent queue seems quite untypical to me, but caution counts.

---

[8]https://en.wikipedia.org/wiki/Queue_(abstract_data_type)

Implementing the first variant is straightforward because it is quite similar to the concurrent stack. Here it is:

A concurrent Queue with Coarse-Grained Locking

```cpp
// concurrentQueueCoarseLocking.cpp

#include <future>
#include <limits>
#include <iostream>
#include <mutex>
#include <queue>
#include <stdexcept>
#include <utility>

template<typename T, template <typename, typename> class Cont = std::deque>
class ConcurrentQueue {
public:
    void push(T val) {
        std::lock_guard<std::mutex> lockQueue(mutexQueue);
        myQueue.push(std::move(val));
    }
    T frontAndPop() {
        std::lock_guard<std::mutex> lockQueue(mutexQueue);
        if ( myQueue.empty() ) throw std::out_of_range("The queue is empty!");
        auto val = myQueue.front();
        myQueue.pop();
        return val;
    }
    ConcurrentQueue() = default;
    ConcurrentQueue(const ConcurrentQueue&) = delete;
    ConcurrentQueue& operator = (const ConcurrentQueue&) = delete;
private:
    mutable std::mutex mutexQueue;
    std::queue<T, Cont<T, std::allocator<T>>> myQueue;
};

int main() {

    ConcurrentQueue<int> conQueue;

    auto fut = std::async([&conQueue]{ conQueue.push(2011); });
    auto fut1 = std::async([&conQueue]{ conQueue.push(2014); });
    auto fut2 = std::async([&conQueue]{ conQueue.push(2017); });

    auto fut3 = std::async([&conQueue]{ return conQueue.frontAndPop(); });
```
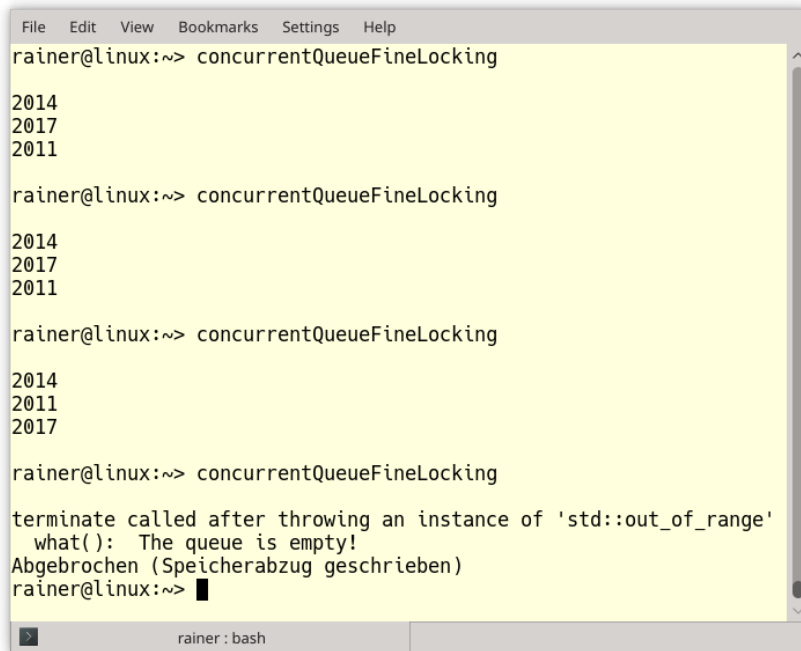
```
42        auto fut4 = std::async([&conQueue]{ return conQueue.frontAndPop(); });
43        auto fut5 = std::async([&conQueue]{ return conQueue.frontAndPop(); });
44
45        fut.get(), fut1.get(), fut2.get();
46
47        std::cout << fut3.get() << '\n';
48        std::cout << fut4.get() << '\n';
49        std::cout << fut5.get() << '\n';
50
51    }
```

Without further ado, the output of the program.

A Concurrent Stack

Done? No, I ignored an optimization possibility.

13.1.3 Fine-Grained Locking

In contrast to a stack, the push and pop operations on the queue happen separately on different ends.

13.1.3.1 A Broken Implementation

Instead of a coarse-grained locking with one mutex, a more fine-grained locking with two mutexes should reduce the synchronization overhead.

A broken implementation of fine-grained concurrent queue

```
template<typename T, template <typename, typename> class Cont = std::deque>
class ConcurrentQueue{
public:

    void push(T val){
        std::lock_guard<std::mutex> lockQueue(mutexBackQueue);
        myQueue.push(std::move(val));
    }
    T frontAndPop(){
        std::lock_guard<std::mutex> lockQueue(mutexFrontQueue);
```

```
        if (myQueue.empty()) throw std::out_of_range("The queue is empty!");
        auto val = myQueue.front();
        myQueue.pop();
        return val;
    }
    ConcurrentQueue() = default;
    ConcurrentQueue(const ConcurrentQueue&) = delete;
    ConcurrentQueue& operator= (const ConcurrentQueue&) = delete;
private:
    mutable std::mutex mutexFrontQueue;
    mutable std::mutex mutexBackQueue;
    std::queue<T, Cont<T, std::allocator<T>>> myQueue;
};
```

There is an obvious issue that makes the implementation incorrect. When the queue is empty, push and pop operate on the same element, which is a data race. Adding an element to separate the push from the pop operations solves this issue.

I could not achieve the fine-grained concurrent queue, based on the abstraction std::queue provides. Now, I have to do it by myself. First of all, how could a queue be implemented?

13.1.3.2 A Simple Queue

The straightforward way to implement a queue is it by using a singly-linked list. Singly-linked means that one node points to the next node but not the other way around. Additionally, a head pointer points to the head and the tail pointer to the tail of the data structure. Items can be removed (pop) from and added (push) to the queue. Elements are removed from the queue by putting the head pointer one position further. The remove operation will also return the old head node. Elements are added to the queue by pointing the previous tail node to the new node. The remove and the add operation adjust the head and the tail pointer.

A Simple Queue

Here is the implementation of the simple queue.

A Simple Queue

```cpp
1    // simpleQueue.cpp
2
3    #include <iostream>
4    #include <memory>
5    #include <utility>
6
7    template <typename T>
8    class Queue{
9    private:
10       struct Node{
11           T data;
12           std::unique_ptr<Node> next;
13           Node(T data_): data(std::move(data_)){}
14       };
15       std::unique_ptr<Node> head;
16       Node* tail;
17   public:
18       Queue(): tail(nullptr){};
19       std::unique_ptr<T> pop(){
20           if (!head) throw std::out_of_range("The queue is empty!");
21           std::unique_ptr<T> res = std::make_unique<T>(std::move(head->data));
22           std::unique_ptr<Node> oldHead = std::move(head);
23           head = std::move(oldHead->next);
24           if (!head) tail = nullptr;
25           return res;
26       }
27       void push(T val){
28           std::unique_ptr<Node> newNode = std::make_unique<Node>(Node(std::move(val)));
29           Node* newTail = newNode.get();
30           if (tail) tail->next= std::move(newNode);
31           else head = std::move(newNode);
32           tail = newTail;
33       }
34       Queue(const Queue& other) = delete;
35       Queue& operator=(const Queue& other) = delete;
36   };
37
38   int main(){
39
40       std::cout << '\n';
41
42       Queue<int> myQueue;
43       myQueue.push(1998);
44       myQueue.push(2003);
```
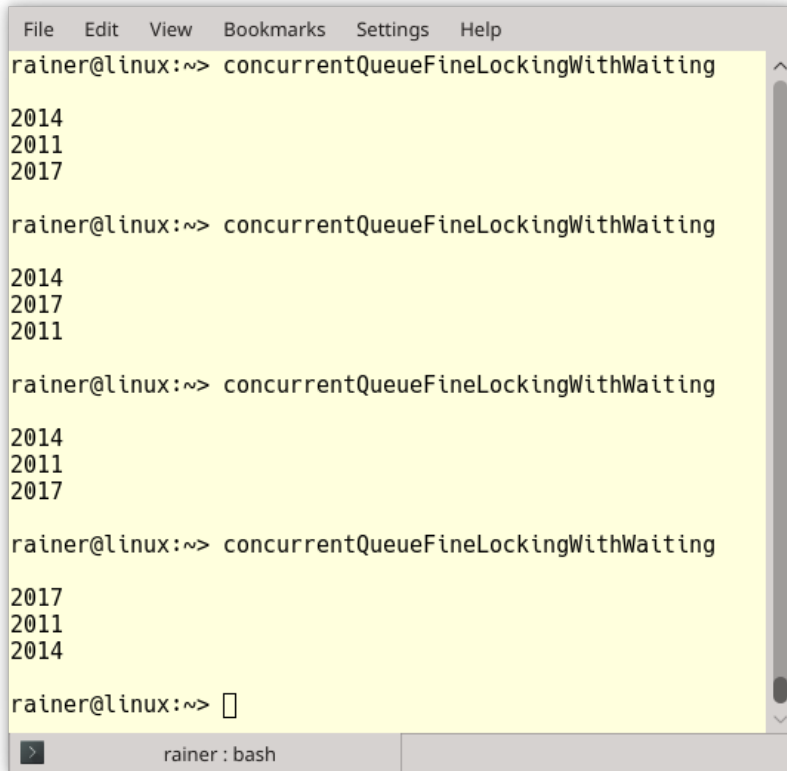
```
45        std::cout << *myQueue.pop() << '\n';
46        std::cout << *myQueue.pop() << '\n';
47        myQueue.push(2011);
48        myQueue.push(2014);
49        std::cout << *myQueue.pop() << '\n';
50        myQueue.push(2017);
51        myQueue.push(2020);
52        std::cout << *myQueue.pop() << '\n';
53        std::cout << *myQueue.pop() << '\n';
54        std::cout << *myQueue.pop() << '\n';
55
56        std::cout << '\n';
57
58    }
```

I use `std::unique_ptrs` to automatically manage the lifetime of the nodes. Only the `tail` pointer is a raw pointer (line 16) because the pointed-to node already has an owner. The member function `push(T val)` (lines 27 - 33) adds a new value to the `Queue`. First, a new node `newNode` (line 28) is created. This new node becomes the new tail (line 29). If `tail` points to an existing `Node` (line 30), `tail`'s pointer is adjusted to the `newNode`; if not, `head` becomes the `newNode` (line 31). Finally, `tail` becomes the `newTail` (line 32). The member function `pop` (lines 19 - 26) removes a node and returns it (line 25). When the queue is empty, the member function throws an exception (line 20). Line 21 creates the return value and move the `head` into the `oldHead` (line 22). `oldHead` is an `std::unique_ptr` and is , therefore, automatically be destroyed when it goes out of scope. `oldHead->next` becomes the new head (line 23). When the `pop` memberfunction returns the last node and the becomes empty , `tail` is set to a `nullptr`.

The screenshot shows the output of the program.

Usage of the Simple Queue

You may ask why I introduced my queue implementation because we have the same issue, such as with the broken implementation before: `head` and `tail` can operate on the same node if the queue has

one node. Subsequent interleaving `pop` and `push` calls could race. For example, the `tail->next` call (line 30) might interleave with the `oldHead->next` call (line 23) in the end. This interleaving means, at the end, that I need one mutex to protect the entire data structure. Let me first answer your question. This queue implementation could not be made thread-safe by using two mutexes, but this implementation is the base for a thread-safe queue with fine-grained locking. I have to apply a trick to separate the head from the tail.

13.1.3.3 A Simple Queue with a Dummy Node

A dummy node does the trick. It separates the head from the tail. Now, the calls next on the head and the tail could not interleave. Of course, the implementation becomes more difficult because the dummy node has to be dealt with.

A Simple Queue with a Dummy Node

```cpp
1   // simpleQueueWithDummy.cpp
2
3   #include <iostream>
4   #include <memory>
5   #include <utility>
6
7   template <typename T>
8   class Queue{
9   private:
10      struct Node{
11          T data;
12          std::unique_ptr<Node> next;
13          Node(T data_): data(std::move(data_)){}
14      };
15      std::unique_ptr<Node> head;
16      Node* tail;
17  public:
18      Queue(): head(new Node(T{})), tail(head.get()){};
19      std::unique_ptr<T> pop(){
20          if (head.get() == tail) throw std::out_of_range("The queue is empty!");
21          std::unique_ptr<T> res = std::make_unique<T>(std::move(head->data));
22          std::unique_ptr<Node> oldHead = std::move(head);
23          head = std::move(oldHead->next);
24          if (!head) tail = nullptr;
25          return res;
26      }
27      void push(T val){
28          std::unique_ptr<Node> dummyNode = std::make_unique<Node>(Node(T{}));
29          Node* newTail = dummyNode.get();
30          tail->next= std::move(dummyNode);
```

```
31              tail->data = val;
32              tail = newTail;
33          }
34      Queue(const Queue& other) = delete;
35      Queue& operator=(const Queue& other) = delete;
36  };
37
38  int main(){
39
40      std::cout << '\n';
41
42      Queue<int> myQueue;
43      myQueue.push(1998);
44      myQueue.push(2003);
45      std::cout << *myQueue.pop() << '\n';
46      std::cout << *myQueue.pop() << '\n';
47      myQueue.push(2011);
48      myQueue.push(2014);
49      std::cout << *myQueue.pop() << '\n';
50      myQueue.push(2017);
51      myQueue.push(2020);
52      std::cout << *myQueue.pop() << '\n';
53      std::cout << *myQueue.pop() << '\n';
54      std::cout << *myQueue.pop() << '\n';
55
56      std::cout << '\n';
57
58  }
```

The differences between the new simple queue to the previous one without a dummy node are not that big. First, `head` and `tail` are initialized to point to the dummy node (line 18). Let's first analyze the `pop` member function. Line 20 checks consequently if the queue is logically empty; meaning, that the queue has only the dummy node. The member function `push` changes more than the member function `pop`. First, a new dummy node is created (line 28), which becomes at the end the node `tail` point's to (line 29 and line 32). The old dummy points to the new dummy node (line 30) and gets the value `val` (line 31).

As expected, the program's output using a queue with a dummy node is the same as the previous one without a dummy node.

Usage of the Simple Queue with a Dummy Node

Now, I'm done with my refactorization. The crucial observation for the queue with a dummy node is that the `pop` member function and the `push` member function work almost entirely on different ends. This separation allows it to use a mutex for each end. Only the check (`head.get() == tail`) (line 20) in the `pop` member function requires both mutexes. This is not so bad because this critical region is short-lived.

With this knowledge in mind, let me put the pieces together and finally come to the concurrent queue with fine-grained locking.

13.1.3.4 The Implementation

The synchronization of the queue is based on two mutexes. One mutex protects the head, and one mutex protects the tail of the queue. The final question is: Where to put the two mutexes? To get maximum performance, the critical regions should be as short-lived as possible. This means for the member function `pop` that the entire member function has to be protected, but for the member function `push` only the instructions that use the queue's tail. The other operations are local and, therefore, thread-safe.

A Fine-Grained Concurrent Queue

```cpp
1   // concurrentQueueFineLocking.cpp
2
3   #include <future>
4   #include <iostream>
5   #include <memory>
6   #include <mutex>
7   #include <utility>
8
9   template <typename T>
10  class ConcurrentQueue{
11  private:
12      struct Node{
```

```
13              T data;
14              std::unique_ptr<Node> next;
15              Node(T data_): data(std::move(data_)){}
16          };
17          std::unique_ptr<Node> head;
18          Node* tail;
19          std::mutex headMutex;
20          std::mutex tailMutex;
21      public:
22          ConcurrentQueue(): head(new Node(T{})), tail(head.get()){};
23          std::unique_ptr<T> pop(){
24              std::lock_guard<std::mutex> headLock(headMutex);
25              {
26                  std::lock_guard<std::mutex> tailLock(tailMutex);
27                  if (head.get() == tail) throw std::out_of_range("The queue is empty!");
28              }
29              std::unique_ptr<T> res = std::make_unique<T>(std::move(head->data));
30              std::unique_ptr<Node> oldHead = std::move(head);
31              head = std::move(oldHead->next);
32              if (!head) tail = nullptr;
33              return res;
34          }
35          void push(T val){
36              std::unique_ptr<Node> dummyNode = std::make_unique<Node>(Node(T{}));
37              Node* newTail = dummyNode.get();
38              std::lock_guard<std::mutex> tailLock(tailMutex);
39              tail->next= std::move(dummyNode);
40              tail->data = val;
41              tail = newTail;
42          }
43          ConcurrentQueue(const ConcurrentQueue& other) = delete;
44          ConcurrentQueue& operator=(const ConcurrentQueue& other) = delete;
45      };
46
47      int main(){
48
49          std::cout << '\n';
50
51          ConcurrentQueue<int> conQueue;
52
53          auto fut = std::async([&conQueue]{ conQueue.push(2011); });
54          auto fut1 = std::async([&conQueue]{ conQueue.push(2014); });
55          auto fut2 = std::async([&conQueue]{ conQueue.push(2017); });
56
57          auto fut3 = std::async([&conQueue]{ return *conQueue.pop(); });
```

```
58      auto fut4 = std::async([&conQueue]{ return *conQueue.pop(); });
59      auto fut5 = std::async([&conQueue]{ return *conQueue.pop(); });
60
61      fut.get(), fut1.get(), fut2.get();
62
63      std::cout << fut3.get() << '\n';
64      std::cout << fut4.get() << '\n';
65      std::cout << fut5.get() << '\n';
66
67      std::cout << '\n';
68
69   }
```

First of all: Is the implementation thread-safe? The `ConcurrentQueue` has only two member functions. Two mutexes protect operations on the shared singly-linked list of `Nodes`. The `headMutex` (line 19) is responsible for protecting the head of the data structure, and the `tailMutex` (line 20) is responsible for protecting the tail of the data structure. The only operation, which could work on the `head` and the `tail` of the singly-linked list is protected by both mutexes (line 27). Consequently, the `ConcurrentQueue` is data race free. You should avoid acquiring more than one mutex at one point in time because this can provoke a deadlock if both mutexes are acquired in a different order. Although the member function `pop` acquires first the `headMutex` (line 24) and then the `tailMutex` (line 26), there is no potential for a deadlock because the mutexes are always acquired in the same order.

Executing the fine-grained concurrent queue works as expected.

A concurrent Queue with Fine-Grained Locking

Sometimes a program run throws an exception. This exception is acceptable due to my design decision. When the queue is empty, I throw a `std::out_of_range` exception in the `pop` member function. Instead of throwing an exception, a more appropriate strategy is to wait.

13.1.3.5 Waiting for the Value

Using a `std::condition_variable` enables the `pop` call to wait until a value is available.

A Concurrent Queue with Fine-Grained Locking and Waiting

```
1   // concurrentQueueFineLockingWithWaiting.cpp
2
3   #include <condition_variable>
4   #include <future>
5   #include <iostream>
6   #include <memory>
7   #include <mutex>
8   #include <utility>
9
10  template <typename T>
11  class Queue{
```

```
12    private:
13        struct Node{
14            T data;
15            std::unique_ptr<Node> next;
16            Node(T data_): data(std::move(data_)){}
17        };
18        std::unique_ptr<Node> head;
19        Node* tail;
20        std::mutex headMutex;
21        std::mutex tailMutex;
22        std::condition_variable condVar;
23    public:
24        Queue(): head(new Node(T{})), tail(head.get()){};
25        std::unique_ptr<T> pop(){
26            std::lock_guard<std::mutex> headLock(headMutex);
27            {
28                std::unique_lock<std::mutex> tailLock(tailMutex);
29                if (head.get() == tail) condVar.wait(tailLock);
30            }
31            std::unique_ptr<T> res = std::make_unique<T>(std::move(head->data));
32            std::unique_ptr<Node> oldHead = std::move(head);
33            head = std::move(oldHead->next);
34            if (!head) tail = nullptr;
35            return res;
36        }
37        void push(T val){
38            std::unique_ptr<Node> dummyNode = std::make_unique<Node>(Node(T{}));
39            Node* newTail = dummyNode.get();
40            {
41                std::unique_lock<std::mutex> tailLock(tailMutex);
42                tail->next= std::move(dummyNode);
43                tail->data = val;
44                tail = newTail;
45            }
46            condVar.notify_one();
47        }
48        Queue(const Queue& other) = delete;
49        Queue& operator=(const Queue& other) = delete;
50    };
51
52    int main(){
53
54        std::cout << '\n';
55
56        Queue<int> conQueue;
```

```
57
58      auto fut = std::async([&conQueue]{ conQueue.push(2011); });
59      auto fut1 = std::async([&conQueue]{ conQueue.push(2014); });
60      auto fut2 = std::async([&conQueue]{ conQueue.push(2017); });
61
62      auto fut3 = std::async([&conQueue]{ return *conQueue.pop(); });
63      auto fut4 = std::async([&conQueue]{ return *conQueue.pop(); });
64      auto fut5 = std::async([&conQueue]{ return *conQueue.pop(); });
65
66      fut.get(), fut1.get(), fut2.get();
67
68      std::cout << fut3.get() << '\n';
69      std::cout << fut4.get() << '\n';
70      std::cout << fut5.get() << '\n';
71
72      std::cout << '\n';
73
74  }
```

The modification to the previous concurrent queue with fine-grained locking is minimal. The `pop` call waits in line 29 if no value is available. `std::condition_variable` requires a `std::unique_lock` instead of a `std::lock_guard`. The `push` call notifies its waiter with `condVar.notify_one` that a new value is available (line 46). `condVar.notify_one` is thread-safe and needs, therefore, no synchronization. Maybe, you miss the predicate in the `wait` call (line 29). This predicate protect against spurious wakeups and lost wakeups. This protection is exactly the job of the condition `head.get() == tail`.

The screenshot ends my story to the lock-based concurrent queue.

A concurrent Queue with Fine-Grained Locking and Waiting

Distilled Information

- Implementing a lock-based data structure is challenging and should be done by experts in this domain.
- Stack and queue are typical lock-based data structures.
- The protection of the lock-based data structure can be coarse-grained or fine-grained. Coarse-grained means that you protect the entire data structure. Fine-grained that you protect only specific elements of the data structure.

14. Lock-Free Data Structures

Cippi plays with a snake

The general thoughts about lock-based data structure and the particular thoughts about a concurrent stack or concurrent queue, also apply to this chapter. Consequentially, I will mention or refer to previous thoughts if necessary. The main difference between lock-based and lock-free data structures is in short that your are faced with higher challenges when you design a lock-free data structure.

Design a Lock-Free Data Structure is Very Challenging

I want to explicitly emphasize that the implementation of lock-free data structures is very challenging. It is quite easy to overlook an issue and end with deadlock or data race[1]. It would be best to consider my examples in this chapter only as a straightforward introduction to lock-free data structures. Don't invent your lock-free structure. Use existing libraries lock-free data structures such as Boost.Lockfree[2] or CDS[3].

---

[1] chapterXXXDataSSSRaces
[2] https://www.boost.org/doc/libs/1_66_0/doc/html/lockfree.html
[3] http://libcds.sourceforge.net/doc/cds-api/index.html

14.1 General Considerations

14.1.1 The Next Evolutionary Step

When you design a thread-safe data structure, you often start with a lock-based one. This means, your first implementation is lock-based and you use coarse-grained locking. Due to the increase of concurrent execution, you switch probably from a coarse-grained implementation to a fine-grained implementation. I presented this route in my previous section about a concurrent queue. My coarse-grained implementation protected the entire queue and my fine-grained only the addressed nodes. The next step in this evolution is obvious: implement the concurrent queue lock-free. Before you make this very challenging step you should answer a few questions?

- Do the performance requirements justify the implementation of the lock-free data structure?
- Is there no existing lock-free data structure available? It is very likely that find your lock-free data structure in Boost[4] or CDS[5].
- Do we have the necessary expertise to implement a lock-based data structure?
- Does your platform support lock-free atomics? The C++ standard guarantees that only `std::atomic_flag` is lock-free.

Only if you answer all questions with yes, you may attack the challenge to implement a lock-based data structure.

14.1.2 Sequential Consistency

I use in my examples to lock-free data structures the default memory ordering: sequential consistency. The reason is simple. Sequential consistency provides the strongest guarantees of all memory ordering and is, therefore, easier to use than the other memory orders. Sequential consistency is an ideal starting point when designing lock-free data structures. In further optimization steps, you can weaken the memory ordering and apply acquire-release semantic, or relaxed semantic.

Depending on the architecture, weakening the memory ordering may not pay off. For example, the x86[6] memory model is one of the strongest memory models of all modern architectures. Consequentially, breaking the sequential consistency and applying a weaker memory ordering may not give the performance improvements you hoped for. On the contrary, ARMv8[7], PowerPC[8], Itanium[9], and, in particular, DEC alpha[10] may pay off when breaking the sequential consistency.

---

[4] https://www.boost.org/doc/libs/1_66_0/doc/html/lockfree.html
[5] http://libcds.sourceforge.net/doc/cds-api/index.html
[6] https://en.wikipedia.org/wiki/X86
[7] https://en.wikipedia.org/wiki/ARM_architecture
[8] https://en.wikipedia.org/wiki/PowerPC
[9] https://en.wikipedia.org/wiki/Itanium
[10] https://en.wikipedia.org/wiki/DEC_Alpha

14.2 Concurrent Stack

First, I want to start with a short reminder. What is a stack?
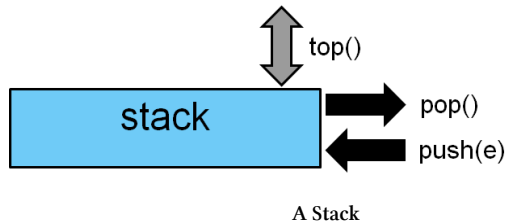
A Stack

A `std::stack`[11] follows the LIFO principle (**L**ast **I**n **F**irst **O**ut). A stack `sta`, which needs the header `<stack>`, has three member functions.

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.

```cpp
#include <stack>
...
std::stack<int> myStack;

std::cout << myStack.empty() << '\n';    // true
std::cout << myStack.size() << '\n';     // 0

myStack.push(1);
myStack.push(2);
myStack.push(3);
std::cout << myStack.top() << '\n';      // 3

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

std::cout << myStack.empty() << '\n';    // true
std::cout << myStack.size() << '\n';     // 0
```

Now, let me start with the implemenation of a lock-free stack.

---

[11]http://en.cppreference.com/w/cpp/container/stack

14.2.1 A Simplified Implementation

Following my strategy in the previous chapter about a lock-based concurrent stack, I implement in this chapter a lock-free concurrent stack. Beside the many similarities of the lock-based and lock-free implementation of the concurrent stack, there is one big difference: The lock-free concurrent stack is based on a singly-linked list and not on a STL container such as the lock-based concurrent stack.

In my simplified implementation, I start with the `push` member function. Let me first visualize, how a new node is added to a singly-linked list. `head` is the pointer to the first node in the singly-linked list.

A singly-linked list

Create a new node

The new node becomes the `head`
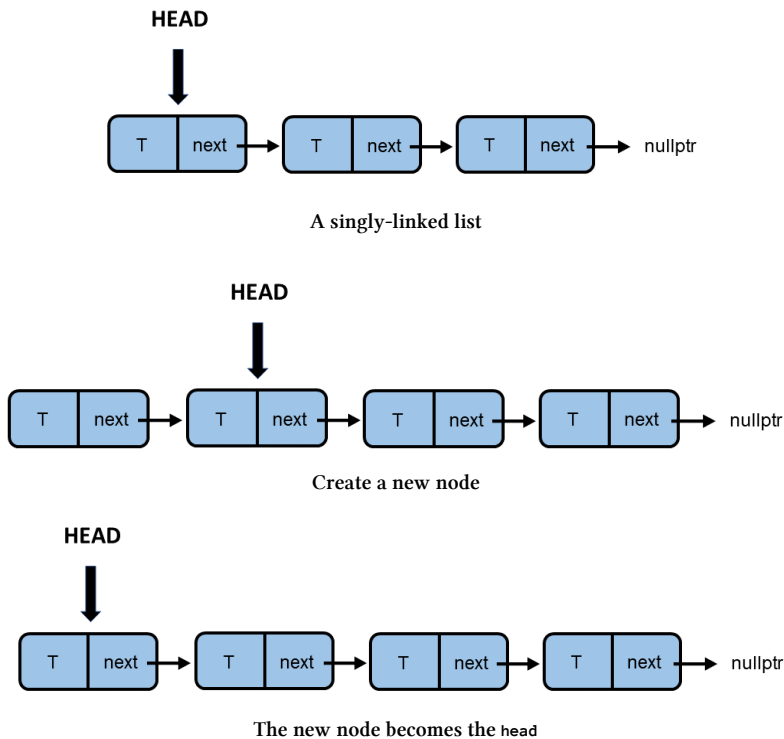
Each node in the singly-linked list has two attributes. Its value `T` and the `next`. `next` points to the next element in the singly-linked list. Only the node points to the `nullptr`. Adding a new node to the data is straightforward. Create a new node and let `next` pointer point to the previous head. So far, the new node is not accessible. Finally, the new node becomes the new head, and the `push` operation is completed.

The following code snippets show the lock-free implementation of a concurrent stack.

A lock-free stack supporting push

```cpp
// lockFreeStackPush.cpp

#include <atomic>
#include <iostream>

template<typename T>
class LockFreeStackPush {
 private:
    struct Node {
        T data;
        Node* next;
        Node(T d): data(d), next(nullptr) {}
    };
    std::atomic<Node*> head;
 public:
    LockFreeStackPush() = default;
    LockFreeStackPush(const LockFreeStackPush&) = delete;
    LockFreeStackPush& operator= (const LockFreeStackPush&) = delete;

    void push(T val) {
        Node* const newNode = new Node(val);
        newNode->next = head.load();
        while( !head.compare_exchange_strong(newNode->next, newNode) );
    }
};

int main(){

    LockFreeStackPush<int> lockFreeStack;
    lockFreeStack.push(5);

    LockFreeStackPush<double> lockFreeStack2;
    lockFreeStack2.push(5.5);

    LockFreeStackPush<std::string> lockFreeStack3;
    lockFreeStack3.push("hello");

}
```

Let me analyze the crucial member function push (line 20). It creates the new node (line 21), adjusts its next pointer to the old head, and makes the new node in a so-called CAS operation the new head (line 22). A CAS operation provides in an atomic step a compare and swap operation.

The call `newNode->next = head.load()` loads the old value of `head`. If the loaded value `newNode->next` is still the some such as `head` in line 23, `head` is updated to the `newNode` and the call `head.compare_-exchange_strong` returns `true`. If not, the call returns `false` and the `while` loop is executed until the call returns `true`. `head.compare_exchange_strong` returns `false` if another thread added in the meantime a new node to the stack. When you apply `compare_exchange_strong` in a loop, you can also use `compare_exchange_weak`. `compare_exchange_weak` can spuriously fail and return `false` if `head` is equal to `newNode->next`. Due to the recursion, this would cause only an additional iteration.

Lines 22 and 23 build a kind of atomic transaction. First, you make a snapshot of the data structure (line 22), then you try to publish the transaction (line 23). If the snapshot is not valid anymore, you make a rollback and try it once more.

> ### `push` **is lock-free but not wait-free**
>
> The previous implementation of the member function `push` is lock-free but not wait-free. When many threads call `compare_exchange_strong` concurrently, only one threads can make progress. All other threads have to wait.

The simplified version has two issues. First, it does not have a `pull` operation, and second, it releases no memory. I address both issues in the next implementation.

14.2.2 A Complete Implementation

Typically, a stack supports the member functions `push`, `pop`, and `top`. Implementing the `pop` and `top` member functions thread-safe, does not guarantee that the invocation of `top` followed by `pop` is thread-safe. It may happen that one thread `t1` called `stack.top()` and is interleaved by another thread `t2` that called `stack.top()` and then `stack.pop()`. Now, the final `t1.pop()` call is based on the wrong stack size. Read more about this issue in the previous chapter about the concurrent stack.

14.2.2.1 No Memory Reclamation

Consequentially, the following implementation combines both member functions `top` and `pop` into one: `topAndPop`.

A lock-free stack supporting `push` and `topAndPop`

```cpp
// lockFreeStackWithLeaks.cpp

#include <atomic>
#include <future>
#include <iostream>
#include <stdexcept>

template<typename T>
class LockFreeStack {
 private:
    struct Node {
        T data;
        Node* next;
        Node(T d): data(d), next(nullptr){ }
    };
    std::atomic<Node*> head;
 public:
    LockFreeStack() = default;
    LockFreeStack(const LockFreeStack&) = delete;
    LockFreeStack& operator= (const LockFreeStack&) = delete;

    void push(T val) {
        Node* const newNode = new Node(val);
        newNode->next = head.load();
        while( !head.compare_exchange_strong(newNode->next, newNode) );
    }

    T topAndPop() {
        Node* oldHead = head.load();
        while( oldHead && !head.compare_exchange_strong(oldHead, oldHead->next) ) {
            if ( !oldHead ) throw std::out_of_range("The stack is empty!");
        }
        return oldHead->data;
    }
};

int main(){

    LockFreeStack<int> lockFreeStack;

    auto fut = std::async([&lockFreeStack]{ lockFreeStack.push(2011); });
    auto fut1 = std::async([&lockFreeStack]{ lockFreeStack.push(2014); });
    auto fut2 = std::async([&lockFreeStack]{ lockFreeStack.push(2017); });

```

```
45        auto fut3 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
46        auto fut4 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
47        auto fut5 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
48
49        fut.get(), fut1.get(), fut2.get();
50
51        std::cout << fut3.get() << '\n';
52        std::cout << fut4.get() << '\n';
53        std::cout << fut5.get() << '\n';
54
55   }
```
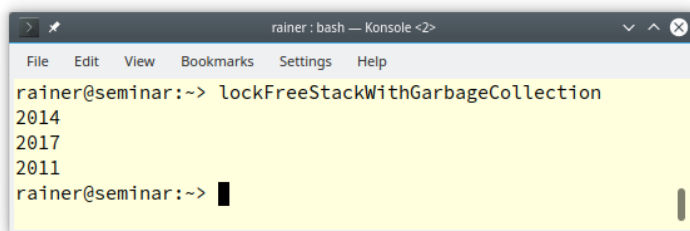
The member function `topAndPop` returns the top element of the stack. It reads the head element of the stack (line 29) and make the next node the new head if `oldHead` is not a `nullptr` (line 30). `oldhead` is a `nullptr` if the stack is empty. I decided to throw an exception if the stack is empty (line 31). Returning a special non-value or returning a `std::optional`[12] is also a valid option. Copying the value has a downside. If the copy constructor of the value throws an exception such as std::bad_alloc[13], the value is lost. Finally, the member functions returns the head element (line 33).

The calls `fut.get()`, `fut1.get()`, `fut2.get()` (line 49) ensure that the associated promise runs. If you don't specify the launch policy, the promise may run lazily in the caller's thread. Lazily means that the promise will execute if and only if the future asks for its result with `get` or `wait`. You can also launch the promise in a separate thread:

Launching each promise in a separate thread

```
auto fut = std::async(std::launch::asnyc, [&conStack]{ conStack.push(2011); });
auto fut1 = std::async(std::launch::asnyc, [&conStack]{ conStack.push(2014); });
auto fut2 = std::async(std::launch::asnyc, [&conStack]{ conStack.push(2017); });
```

Finally, the output of the program:

---
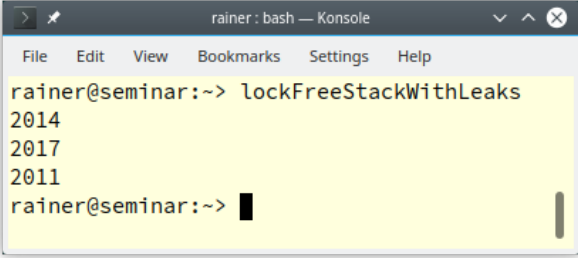
[12]https://en.cppreference.com/w/cpp/utility/optional
[13]https://en.cppreference.com/w/cpp/memory/new/bad_alloc

A lock-free stack

Although the presented lock-free stack supports `push` and `topAndPop`, it has a serious issue: it leaks memory. You may ask: Why can't the `oldHead` just be removed after the call `head.compare_exchange_-strong(oldHead, oldHead->next)` (line 30) in the member function `topAndPop`? The answer is that another thread may use `oldHead`. Let's analyze the member functions `push` and `topAndPop`. Concurrent execution of `push` is no issue because the call `!head.compare_exchange_strong(newNode->next, newNode)` atomically updates `newNode->next` to the new head. This is also valid if only one `topAndPop` call happens concurrently. The issue arises when more `topAndPop` calls interleave with or without a `push` call. Deleting the `oldHead` while another thread uses it would be disastrous, because the deletion of `oldHead` must always happen before or after its update to the new head: `oldHead->next` (line 30).

The easiest way to solve this memory-leak issue is to use a `std::shared_ptr`.

14.2.2.1.1 Atomic Smart Pointer

There are two ways to apply atomic operatons on a `std::shared_ptr`: In C++11, you can use the free atomic functions on `std::shared_ptr`. With C++20, you can use atomic smart pointers.

14.2.2.1.2 C++11

Using atomic operations on `std::shared_ptr` is tedious and error-prone. You can easily forget the atomic operations and all bets are open. The following example shows a lock-free stack based on `std::shared_ptr`.

A lock-free stack based on atomic smart pointers

```
1  // lockFreeStackWithSharedPtr.cpp
2
3  #include <atomic>
4  #include <future>
5  #include <iostream>
6  #include <stdexcept>
7  #include <memory>
8
9  template<typename T>
```

```
10    class LockFreeStack {
11     public:
12        struct Node {
13            T data;
14            std::shared_ptr<Node> next;
15        };
16        std::shared_ptr<Node> head;
17     public:
18        LockFreeStack() = default;
19        LockFreeStack(const LockFreeStack&) = delete;
20        LockFreeStack& operator= (const LockFreeStack&) = delete;
21
22        void push(T val) {
23            auto newNode = std::make_shared<Node>();
24            newNode->data = val;
25            newNode->next = std::atomic_load(&head);
26            while( !std::atomic_compare_exchange_strong(&head, &newNode->next, newNode) );
27        }
28
29        T topAndPop() {
30            auto oldHead = std::atomic_load(&head);
31            while( oldHead && !std::atomic_compare_exchange_strong(&head, &oldHead,
32                                              std::atomic_load(&oldHead->next)) ) {
33                if ( !oldHead ) throw std::out_of_range("The stack is empty!");
34            }
35            return oldHead->data;
36        }
37    };
38
39    int main(){
40
41        LockFreeStack<int> lockFreeStack;
42
43        auto fut = std::async([&lockFreeStack]{ lockFreeStack.push(2011); });
44        auto fut1 = std::async([&lockFreeStack]{ lockFreeStack.push(2014); });
45        auto fut2 = std::async([&lockFreeStack]{ lockFreeStack.push(2017); });
46
47        auto fut3 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
48        auto fut4 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
49        auto fut5 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
50
51        fut.get(), fut1.get(), fut2.get();
52
53        std::cout << fut3.get() << '\n';
54        std::cout << fut4.get() << '\n';
```

```
55        std::cout << fut5.get() << '\n';
56
57  }
```

This lock-free stack implementation is quite similar to the previous one without memory reclamation. The main difference is that the nodes are of type `std::shared_ptr<Node>`. All operations `std::shared_‐ptr<Node>` are done atomically by using the free atomic operations `std::load` (lines 25, and 32), and `std::atomic_compare_exchange_strong` (lines 26, and 31). The free atomics operations require a pointer. I want to emphasize it explicitly, the read operation of the next node in `oldHead->next` (line 32) must be atomic because `oldHead->next` can be used by other threads. Finally, here is the output of the program.

A lock-free stack based on smart pointers

Let's jump nine years into the future and use C++20.

14.2.2.1.3 C++20

C++20 supports partial specializations of `std::atomic` for `std::shared_ptr` and `std::weak_ptr`. The following implementation puts the nodes of the lock-free stack into a `std::atomic<std::shared_‐ptr<Node>>`.

A lock-free stack based on atomic smart pointers

```cpp
1   // lockFreeStackWithAtomicSharedPtr.cpp
2
3   #include <atomic>
4   #include <future>
5   #include <iostream>
6   #include <stdexcept>
7   #include <memory>
8
9   template<typename T>
10  class LockFreeStack {
11   private:
12      struct Node {
```
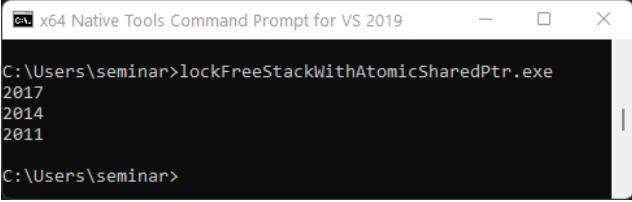
```cpp
13          T data;
14          std::shared_ptr<Node> next;
15      };
16      std::atomic<std::shared_ptr<Node>> head;
17  public:
18      LockFreeStack() = default;
19      LockFreeStack(const LockFreeStack&) = delete;
20      LockFreeStack& operator= (const LockFreeStack&) = delete;
21
22      void push(T val) {
23          auto newNode = std::make_shared<Node>();
24          newNode->data = val;
25          newNode->next = head;
26          while( !head.compare_exchange_strong(newNode->next, newNode) );
27      }
28
29      T topAndPop() {
30          auto oldHead = head.load();
31          while( oldHead && !head.compare_exchange_strong(oldHead, oldHead->next) ) {
32              if ( !oldHead ) throw std::out_of_range("The stack is empty!");
33          }
34          return oldHead->data;
35      }
36  };
37
38  int main(){
39
40      LockFreeStack<int> lockFreeStack;
41
42      auto fut = std::async([&lockFreeStack]{ lockFreeStack.push(2011); });
43      auto fut1 = std::async([&lockFreeStack]{ lockFreeStack.push(2014); });
44      auto fut2 = std::async([&lockFreeStack]{ lockFreeStack.push(2017); });
45
46      auto fut3 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
47      auto fut4 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
48      auto fut5 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
49
50      fut.get(), fut1.get(), fut2.get();
51
52      std::cout << fut3.get() << '\n';
53      std::cout << fut4.get() << '\n';
54      std::cout << fut5.get() << '\n';
55
56  }
```

This main difference between the previous implementation and this implementation is that the `Node` is embedded into a `std::atomic<std::shared_ptr<Node>>` (line 16). Consequentially, the member function `push` (line 22) creates a `std::shared_ptr<Node>` and the call `head.load()` in the member function `topAndPop` returns a `std::atomic<std::shared_ptr<Node>>`. Here is the output of the program.

A lock-free stack based on atomic smart pointers

14.2.2.1.4 `std::atomic<std::shared_ptr>` is not Lock-Free on Windows

Honestly, I cheated in the previous programs using a atomic operations on a `std::shared_ptr`, and using a `std::atomic<shared_ptr>`. You have to assume, that atomics operations on a `std::shared_ptr` are not lock-free. Additionally, an implementation of `std::atomic<std::shared_ptr>` can use a locking mechanism to support all partial and full specializations of `std::atomic`. This happened in the case of the used Visual Studio C++ compiler. The call `atom.lock_free()` on a `std::atomic<std::shared_ptr<Node>>` returns `false`.

Atomic Smart pointer uses locks under the hood

```cpp
1    // atomicSmartPointer.cpp
2
3    #include <atomic>
4    #include <iostream>
5    #include <memory>
6
7    template <typename T>
8    struct Node {
9        T data;
10       std::shared_ptr<Node> next;
11   };
12
13   int main() {
14
15       std::cout << '\n';
16
17       std::cout << std::boolalpha;
18
19       std::atomic<std::shared_ptr<Node<int>>> node;
20       std::cout << "node.is_lock_free(): "  << node.is_lock_free() << '\n';
21
```
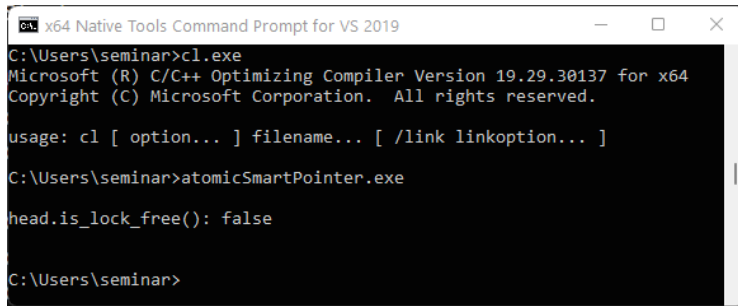
```
22        std::cout << '\n';
23
24    }
```

The Visual Studio compiler 19.29.30137 for x64 uses a locking mechanism to support atomic smart pointers.

std::atomic<std::shared_ptr> uses on Windows a locking mechanism

Therefore, we I'm back to square one and have to take care of memory management.

14.2.2.2 A Simple Garbage Collector for Nodes

I discussed in the section about the lock-free stack implementation leaking memory that concurrent execution of more than one topAndPush call is a Race Condition. Consequentially, I can safely delete a node if not more than one topAndPush call is concurrently executing. This observation is crucial for solving this memory leak issue: I store removed nodes on a to be deleted list, and I delete the nodes on this list if no more than one topAndPush call is active. There is only one challenge left: How can I be sure that not more than one topAndPush call is active? I use an atomic counter that is incremented at the start of topAndPush and decremented at its end. The counter is zero or one if no or one topAndPush call is active.

The following program implements the presented strategy. I use the lockFreeStackWithLeaks.cpp as starting point.

A lock-free stack including garbage collection

```cpp
1   // lockFreeStackWithGarbageCollection.cpp
2
3   #include <atomic>
4   #include <future>
5   #include <iostream>
6   #include <stdexcept>
7   #include <thread>
8
9   template<typename T>
10  class LockFreeStack {
11   private:
12      struct Node {
13          T data;
14          Node* next;
15          Node(T d): data(d), next(nullptr){ }
16      };
17
18      std::atomic<Node*> head{nullptr};
19      std::atomic<int> topAndPopCounter{};
20      std::atomic<Node*> toBeDeletedNodes{nullptr};
21
22      void tryToDelete(Node* oldHead) {
23          if (topAndPopCounter  == 1) {
24              Node* copyOfToBeDeletedNodes = toBeDeletedNodes.exchange(nullptr);
25              if (topAndPopCounter == 1) deleteAllNodes(copyOfToBeDeletedNodes);
26              else addNodeToBeDeletedNodes(copyOfToBeDeletedNodes);
27              delete oldHead;
28          }
29          else addNodeToBeDeletedNodes(oldHead);
30      }
31
32      void addNodeToBeDeletedNodes(Node* oldHead) {
33          oldHead->next = toBeDeletedNodes;
34          while( !toBeDeletedNodes.compare_exchange_strong(oldHead->next, oldHead));
35      }
36
37      void deleteAllNodes(Node* currentNode) {
38          while (currentNode) {
39              Node* nextNode = currentNode->next;
40              delete currentNode;
41              currentNode = nextNode;
42          }
43      }
44
```

```cpp
45      public:
46          LockFreeStack() = default;
47          LockFreeStack(const LockFreeStack&) = delete;
48          LockFreeStack& operator= (const LockFreeStack&) = delete;
49
50          void push(T val) {
51              Node* const newNode = new Node(val);
52              newNode->next = head.load();
53              while( !head.compare_exchange_strong(newNode->next, newNode) );
54          }
55
56          T topAndPop() {
57              ++topAndPopCounter;
58              Node* oldHead = head.load();
59              while( oldHead && !head.compare_exchange_strong(oldHead, oldHead->next) ) {
60                  if ( !oldHead ) throw std::out_of_range("The stack is empty!");
61              }
62              auto topElement = oldHead->data;
63              tryToDelete(oldHead);
64              --topAndPopCounter;
65              return topElement;
66          }
67      };
68
69      int main(){
70
71          LockFreeStack<int> lockFreeStack;
72
73          auto fut = std::async([&lockFreeStack]{ lockFreeStack.push(2011); });
74          auto fut1 = std::async([&lockFreeStack]{ lockFreeStack.push(2014); });
75          auto fut2 = std::async([&lockFreeStack]{ lockFreeStack.push(2017); });
76
77          auto fut3 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
78          auto fut4 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
79          auto fut5 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
80
81          fut.get(), fut1.get(), fut2.get();
82
83          std::cout << fut3.get() << '\n';
84          std::cout << fut4.get() << '\n';
85          std::cout << fut5.get() << '\n';
86
87      }
```

The lock-free stack has two new attributes and three new member functions. The atomic counter topAndPopCounter counts (line 19) the number of active topAndPop calls, and the atomic pointer toBeDeletedNodes (line 20) is a pointer to the list of the to be deleted nodes. Additionally, the member function tryToDelete (line 22) tries to delete removed nodes. The member functions addNodeToBeDeletedNodes ads a node to the to be deleted list, and the member function deleteAllNodes (line 37) deletes all nodes.

Let's analyze the member function topAndPop (lines 56 - 66). At the beginning and the end of topAndPop, topAndPopCounter is incremented and decremented. oldHead is removed from the stack and can, therefore, eventually be deleted with the call tryToDelete (line 63). The member function tryToDelete first checks if one or more topAndPush calls are active. If one topAndPush call is active (line 23), oldHead is deleted. If not, oldHead is added to the to be deleted list (line 29). I assume that only one topAndPush call is active. In this case, I create a local pointer copyOfToBeDeletedNodes to the to be deleted nodes, and set the toBeDeletedNodes pointer to a nullptr (line 24). Before I delete the nodes, I check that no additional topAndPush call is active in the meantime. If the current execution topAndPush is still the only one, I use the local pointer copyOfToBeDeletedNodes to delete the list of all to be deleted nodes (line 25). If another topAndPush call interleaved, I use the local pointer copyOfToBeDeletedNodes to update toBeDeletedNodes pointer.

Both helper member functions addNodeToBeDeletedNodes and deleteAllNodes iterate through the list. deleteAllNodes is only invoked if one topAndPop call is active (line 25). Consequentially, no synchronization is necessary. This observation does not hold for the member function addNodeToBeDeletedNodes (lines 26 and 29). It must be synchronized because more than one topAndPop call can be active. The while loop makes the oldHead the first node in the to be deleted nodes and uses a compare_-exchange_strong to deal with the fact that topAndPop calls can interleave. Interleaving topAndPop call may cause that oldHead->next != toBeDeletedNodes (line 34) and oldHead->next has to be updated to toBeDeletedNodes.

A lock-free stack with garbage collection

So-far, this lock-free stack implementation works as expected but has a few flaws. When many topAndPop calls interleave it may happen that the counter topAndPopCounter never becomes one. This means that the nodes in the to be deleted lists of nodes are not deleted, and we have a memory leaks. Additionally, the number of the to be deleted nodes become a resource issue.

14.2.2.3 Hazard Pointers

The term hazard pointers goes back to Maged Michael. He described them in his paper "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects"[14]. Hazard pointers solve the classical problem of lock-free data structures such as a lock-free stack: When can a thread safely delete a node of a data structure while other threads may use this node concurrently?

Although a hazard pointer provides a general solution for the common problem of safe memory reclamation in lock-free data structures, I want to present it from the perspective of our lock-free stack.
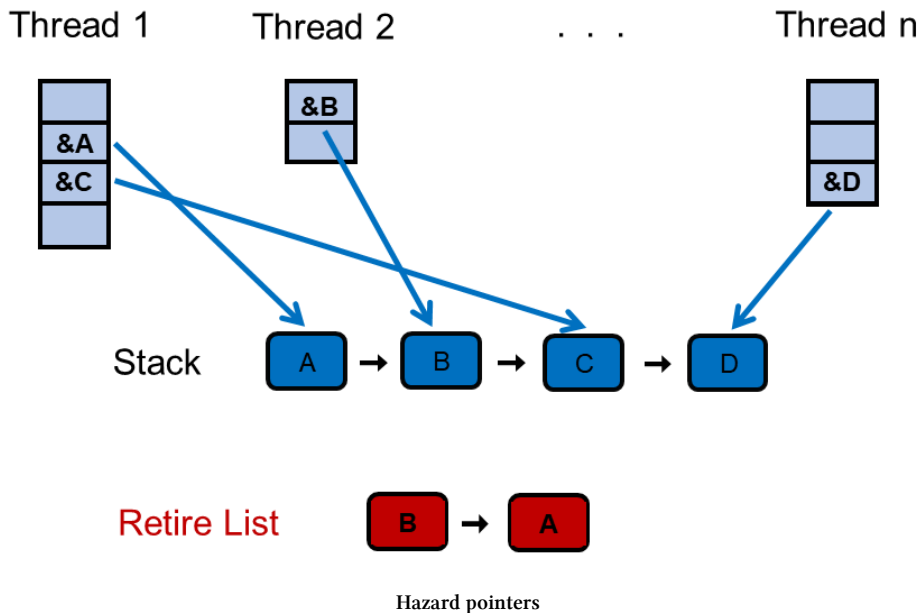
Hazard pointers

A hazard pointer is a single-writer multi-reader pointer. All hazard pointers build a linked list and are initialized with a null pointer. When a thread uses a stack node, it puts the node's address into a hazard pointer, indicating, that it uses this node and is also the exclusive owner of the used hazard pointer. When the thread is done using the node, it sets the pointer of the hazard pointer back to a null pointer and, therefore, releases its ownership. A thread keeps a list of hazard pointers standing for the nodes the thread is using and can not be deleted. When a thread wants to delete a node, it traverses the list of all hazard pointers and checks if the node is used. If the node is not in use, it deletes it. If the node is in use, it puts it eventually on a retire list of the to be deleted nodes. Eventually, because the node is only added to the retire list if it is not yet on the list.

This means in the case of our lock-free stack. The member function `topAndPop` has two jobs regarding memory reclamation. First, it manages the to be deleted node, and second, it traverses the retire list

---

[14]http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.395.378&rep=rep1&type=pdf

of nodes and deletes them if they aren't used anymore.

I need the following member function in a new implementation of `topAndPop` based on the previous description: `getHazardPointer` to get a reference to a hazard pointer, `retireList.addNode`, and `retireList.deleteUnusedNodes` to add a node to the retire list `retireList`. Additionally `retireListe.deleteUnusedNod` to delete all nodes from the retire list that are not used anymore. Additionally, the member function `retireList.deleteUnusedNodes` uses the helper function `retireList.isInUse` that decides if a node is currently used. The member function `isInUse` is also handy in `topAndPop` to decide if the current node should be added to the retire list or directly deleted.

What does this mean for my previous `LockFreeStack` implementation without memory reclamation? Let's see. The following program shows the lock-free stack implementation based on hazard pointers. I analyze it step by step.

A lock-free stack using hazard pointers

```cpp
// lockFreeStackHazardPointers.cpp

#include <atomic>
#include <cstddef>
#include <future>
#include <iostream>
#include <stdexcept>
#include <thread>

template <typename T>
concept Node = requires(T a) {
    {T::data};
    { *a.next } -> std::same_as<T&>;
};

template <typename T>
struct MyNode {
    T data;
    MyNode* next;
    MyNode(T d): data(d), next(nullptr){ }
};

constexpr std::size_t MaxHazardPointers = 50;

template <typename T, Node MyNode = MyNode<T>>
struct HazardPointer {
    std::atomic<std::thread::id> id;
    std::atomic<MyNode*> pointer;
};

template <typename T>
```

```cpp
HazardPointer<T> HazardPointers[MaxHazardPointers];

template <typename T, Node MyNode = MyNode<T>>
class HazardPointerOwner {

    HazardPointer<T>* hazardPointer;

 public:
    HazardPointerOwner(HazardPointerOwner const &) = delete;
    HazardPointerOwner operator=(HazardPointerOwner const &) = delete;

    HazardPointerOwner() : hazardPointer(nullptr) {
        for (std::size_t i = 0; i < MaxHazardPointers; ++i) {
            std::thread::id old_id;
            if (HazardPointers<T>[i].id.compare_exchange_strong(
                                 old_id, std::this_thread::get_id())) {
                hazardPointer = &HazardPointers<T>[i];
                break;
            }
        }
        if (!hazardPointer) {
            throw std::out_of_range("No hazard pointers available!");
        }
    }

    std::atomic<MyNode*>& getPointer() {
        return hazardPointer->pointer;
    }

    ~HazardPointerOwner() {
        hazardPointer->pointer.store(nullptr);
        hazardPointer->id.store(std::thread::id());
    }
};

template <typename T, Node MyNode = MyNode<T>>
std::atomic<MyNode*>& getHazardPointer() {
    thread_local static HazardPointerOwner<T> hazard;
    return hazard.getPointer();
}

template <typename T, Node MyNode = MyNode<T>>
class RetireList {

    struct RetiredNode {
```

```cpp
        MyNode* node;
        RetiredNode* next;
        RetiredNode(MyNode* p) : node(p), next(nullptr) { }
        ~RetiredNode() {
            delete node;
        }
    };

    std::atomic<RetiredNode*> RetiredNodes;

    void addToRetiredNodes(RetiredNode* retiredNode) {
        retiredNode->next = RetiredNodes.load();
        while (!RetiredNodes.compare_exchange_strong(retiredNode->next, retiredNode));
    }

public:

    bool isInUse(MyNode* node) {
        for (std::size_t i = 0; i < MaxHazardPointers; ++i) {
            if (HazardPointers<T>[i].pointer.load() == node) return true;
        }
        return false;
    }

    void addNode(MyNode* node) {
        addToRetiredNodes(new RetiredNode(node));
    }

    void deleteUnusedNodes() {
        RetiredNode* current = RetiredNodes.exchange(nullptr);
        while (current) {
            RetiredNode* const next = current->next;
            if (!isInUse(current->node)) delete current;
            else addToRetiredNodes(current);
            current = next;
        }
    }

};

template<typename T, Node MyNode = MyNode<T>>
class LockFreeStack {

    std::atomic<MyNode*> head;
    RetireList<T> retireList;
```

```cpp
public:
    LockFreeStack() = default;
    LockFreeStack(const LockFreeStack&) = delete;
    LockFreeStack& operator= (const LockFreeStack&) = delete;

    void push(T val) {
        MyNode* const newMyNode = new MyNode(val);
        newMyNode->next = head.load();
        while( !head.compare_exchange_strong(newMyNode->next, newMyNode) );
    }

    T topAndPop() {
        std::atomic<MyNode*>& hazardPointer = getHazardPointer<T>();
        MyNode* oldHead = head.load();
        do {
            MyNode* tempMyNode;
            do {
                tempMyNode = oldHead;
                hazardPointer.store(oldHead);
                oldHead = head.load();
            } while( oldHead != tempMyNode );
        } while( oldHead && !head.compare_exchange_strong(oldHead, oldHead->next) ) ;
        if ( !oldHead ) throw std::out_of_range("The stack is empty!");
        hazardPointer.store(nullptr);
        auto res = oldHead->data;
        if ( retireList.isInUse(oldHead) ) retireList.addNode(oldHead);
        else delete oldHead;
        retireList.deleteUnusedNodes();
        return res;
    }
};

int main(){

    LockFreeStack<int> lockFreeStack;

    auto fut = std::async([&lockFreeStack]{ lockFreeStack.push(2011); });
    auto fut1 = std::async([&lockFreeStack]{ lockFreeStack.push(2014); });
    auto fut2 = std::async([&lockFreeStack]{ lockFreeStack.push(2017); });

    auto fut3 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
    auto fut4 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
    auto fut5 = std::async([&lockFreeStack]{ return lockFreeStack.topAndPop(); });
```
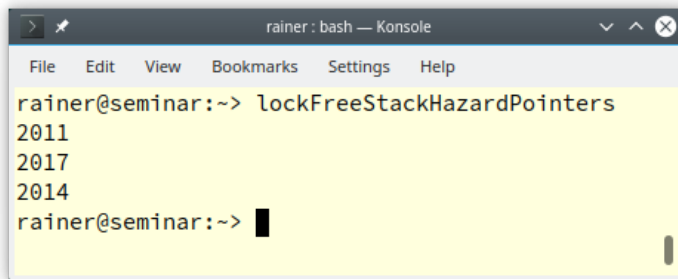
```
      fut.get(), fut1.get(), fut2.get();

      std::cout << fut3.get() << '\n';
      std::cout << fut4.get() << '\n';
      std::cout << fut5.get() << '\n';

}
```

The program runs as expected.

Hazard pointers

`MyNode` is a class template, parametrized by the type it holds: `data`. `MyNode` models the concept `Node`.

MyNode

```
1   template <typename T>
2   concept Node = requires(T a) {
3       {T::data};
4       { *a.next } -> std::same_as<T&>;
5   };
6
7   template <typename T>
8   struct MyNode {
9       T data;
10      MyNode* next;
11      MyNode(T d): data(d), next(nullptr){ }
12  };
```

Concepts are compile-time predicates. They put semantic constraints on template parameters. The concept `Node` requires a member `data` (line 3) and a pointer `next` (line 4) that returns a `Node`. The types in the program `lockFreeStackHazardPointers.cpp` are essentially parametrized on the `data` member and the concept `Node`. `MyNode` models the concept `Node`. For example, here is the declaration of the `LockFreeStack`:

```
template<typename T, Node MyNode = MyNode<T>>
class LockFreeStack;
```

You can read more details about concepts in my blogposts on ModernesCpp/concepts[15] or in my C++20[16] book.

Let me continue my analysis of the program with the lock-free stack.

The `LockFreeStack`

```
1   template<typename T, Node MyNode = MyNode<T>>
2   class LockFreeStack {
3
4       std::atomic<MyNode*> head;
5       RetireList<T> retireList;
6
7    public:
8       LockFreeStack() = default;
9       LockFreeStack(const LockFreeStack&) = delete;
10      LockFreeStack& operator= (const LockFreeStack&) = delete;
11
12      void push(T val) {
13          MyNode* const newMyNode = new MyNode(val);
14          newMyNode->next = head.load();
15          while( !head.compare_exchange_strong(newMyNode->next, newMyNode) );
16      }
17
18      T topAndPop() {
19          std::atomic<MyNode*>& hazardPointer = getHazardPointer<T>();
20          MyNode* oldHead = head.load();
21          do {
22              MyNode* tempMyNode;
23              do {
24                  tempMyNode = oldHead;
25                  hazardPointer.store(oldHead);
26                  oldHead = head.load();
27              } while( oldHead != tempMyNode );
28          } while( oldHead && !head.compare_exchange_strong(oldHead, oldHead->next) ) ;
29          if ( !oldHead ) throw std::out_of_range("The stack is empty!");
30          hazardPointer.store(nullptr);
31          auto res = oldHead->data;
32          if ( retireList.isInUse(oldHead) ) retireList.addNode(oldHead);
33          else delete oldHead;
34          retireList.deleteUnusedNodes();
```

---

[15]https://www.modernescpp.com/index.php/tag/concepts
[16]https://leanpub.com/c20

```
35          return res;
36      }
37  };
```

The push call is not critical from a concurrency perspective because head is updated in an atomic step. Additionally, the compare_exchange_strong (line 15) call guarantees that head is always the current head of the stack.

Due to hazard pointers, the call topAndPop becomes more complicated. First, the function getHazardPointer references the hazard pointer for the current thread (line 19). The call hazardPointer.store(oldHead) (line 25) makes the current thread to the owner of the hazard pointer, and the call hazardPointer.store(nullptr) (line 30) releases its ownership. First, let me analyze the inner and outer do-while loops (lines 21 - 28). The inner do-while loop sets the hazard pointer to the head of the stack. The do-while loop ends when the following holds: oldHead == tempNode. Both nodes are equal if oldHead is still the current head of the lock-free stack. oldHead was set in line 20 and could not be the current head anymore (line 28) because another thread may kicked in and already managed oldHead. The outer do-while loop should be familiar from the previous lock-free stack implementations. I iterate in the while loop using compare_exchange_strong and set the head to oldHead->next. On end, head is the head of the stack. Remember, the member function topAndPop should return the value of the head and remove it. Before I use oldHead I have to check if oldHead is not a null pointer. If oldHead is a null pointer, I throw an exception. The rest of the topAndPoP is straightforward. The call retirenListe.isInUse(oldHead) checks if oldHead is still in use. Depending on this check, oldHead is added to the retire list retireList.addNode (line 32) if it is not yet on the list or deleted (line 33). The last call retireList.deleteUnusedNodes (line 37) is the most labour-intensive call in the member function topAndPop. The member function retireListe.deleteUnusedNodes traverses the entire retire list and deletes all nodes that are not used anymore.

For performance reasons, the call retireList.deleteUnusedNodes should not be executed in each call of topAndPop. An improved strategy is to invoke the member function deleteUnusedNodes if the length of the retire list exceeds a specific threshold. For example, when the length of the retire list is twice the length of the stack, at least half of the nodes can be deleted. This threshold value is a trade-off between performance requirements and memory consumption.

Let me continue my explanation with the free function getHazardPointer and the retire list.

The free function getHazardPointer

```
template <typename T, Node MyNode = MyNode<T>>
std::atomic<MyNode*>& getHazardPointer() {
    thread_local static HazardPointerOwner<T> hazard;
    return hazard.getPointer();
}
```

The function getHazardPointer references a hazard pointer using the hazard pointer owner hazard. hazard is a thread-local and static variable. Therefore, each thread gets its copy of the hazard pointer

owner, and its lifetime is bound to the lifetime of the owning thread. The bound lifetime of the hazard pointer owner is crucial because it guarantees the hazard pointer is cleared when the thread-local hazard pointer owner is destroyed. I write more about this RAII object in my analysis of the class type HazardPointerOwner.

The retire list has the public member functions isInUse, addNode, and deleteUnusedNodes. Additionally, it has the inner class RetireNode, an atomic member of it, and the private member function addToRetiredNodes.

The RetireList

```
1   template <typename T, Node MyNode = MyNode<T>>
2   class RetireList {
3
4       struct RetiredNode {
5           MyNode* node;
6           RetiredNode* next;
7           RetiredNode(MyNode* p) : node(p), next(nullptr) { }
8           ~RetiredNode() {
9               delete node;
10          }
11      };
12
13      std::atomic<RetiredNode*> RetiredNodes;
14
15      void addToRetiredNodes(RetiredNode* retiredNode) {
16          retiredNode->next = RetiredNodes.load();
17          while (!RetiredNodes.compare_exchange_strong(retiredNode->next, retiredNode));
18      }
19
20    public:
21
22      bool isInUse(MyNode* node) {
23          for (std::size_t i = 0; i < MaxHazardPointers; ++i) {
24              if (HazardPointers<T>[i].pointer.load() == node) return true;
25          }
26          return false;
27      }
28
29      void addNode(MyNode* node) {
30          addToRetiredNodes(new RetiredNode(node));
31      }
32
33      void deleteUnusedNodes() {
34          RetiredNode* current = RetiredNodes.exchange(nullptr);
35          while (current) {
```

```
36              RetiredNode* next = current->next;
37              if (!isInUse(current->node)) delete current;
38              else addToRetiredNodes(current);
39              current = next;
40          }
41      }
42
43  };
```

Let's start with the interface of the type `RetireList`.

The member function `isInUse` (line 22) checks if `node` is in use. It does its job by traversing the variable template[17] `HazardPointers` that is parameterized on the type of data the node holds. `HazardPointer` is a C-array of `HazardPointer` of length 50. A `HazardPointer` consists of an atomic thread id and an atomic pointer to a node.

The variable template `HazardPointers`

```
constexpr std::size_t MaxHazardPointers = 50;

template <typename T, Node MyNode = MyNode<T>>
struct HazardPointer {
    std::atomic<std::thread::id> id;
    std::atomic<MyNode*> pointer;
};

template <typename T>
HazardPointer<T> HazardPointers[MaxHazardPointers];
```

Using an STL container such as `std::set`[18] as `HazardPointers` would be way more convenient. `std::set` is already ordered and guarantees constant access time on average but has a big issue: it's not thread-safe.

The member function `addNode` (line 29) takes a node, invokes the private member function `addToRetiredNodes` (line 15) and puts the `node` into an `RetiredNode`. `RetiredNode` (line 4 - 9) is an RAII object and guarantees that the wrapped `node` is always destroyed and, therefore, its memory is released. All retired nodes build a singly-linked list (line 13).

The member function `deleteUnusedNodes` (line 33) traverses the singly-linked list of retired nodes by applying the following pattern:

---

[17]https://en.cppreference.com/w/cpp/language/variable_template
[18]https://en.cppreference.com/w/cpp/container/set

Iteration of a singly-linked list

```
1  void deleteUnusedNodes() {
2      RetiredNode* current = RetiredNodes.exchange(nullptr);
3      while (current) {
4          RetiredNode* const next = current->next;
5          if (!isInUse(current->node)) delete current;
6          else addToRetiredNodes(current);
7          current = next;
8      }
9  }
```

It checks the current node (line 3), points the `next` node to `current->next` (line 4), and updates the `current` node with the `next` node (line 7). Finally, the `current` node is destroyed if not used anymore (line 5), or added to the retired nodes (line 6). The private member function `addToRetireNodes` (line 17) adds the retired nodes to the singly-linked list. To perform its job, it loads the `RetiredNodes` and makes the new node `retiredNode` to the new head of the singly-linked list (line 17). Before `retiredNode` becomes the new head of the singly-linked list, I have to ensure that `RetiredNode` is still the head of the singly-linked list because another thread could kick in and changed the head of the singly-linked list in the meantime. Thanks to the while-loop (line 17), `retiredNode` becomes only the new head if ` retiredNode->next = RetiredNodes.load() holds. If not, retiredNode->next is updated to RetiredNodes.load()`.

There is only one peace of the puzzle left:

The `HazardPointerOwner`

```
1  template <typename T, Node MyNode = MyNode<T>>
2  class HazardPointerOwner {
3
4      HazardPointer<T>* hazardPointer;
5
6   public:
7      HazardPointerOwner(HazardPointerOwner const &) = delete;
8      HazardPointerOwner operator=(HazardPointerOwner const &) = delete;
9
10     HazardPointerOwner() : hazardPointer(nullptr) {
11         for (std::size_t i = 0; i < MaxHazardPointers; ++i) {
12             std::thread::id old_id;
13             if (HazardPointers<T>[i].id.compare_exchange_strong(
14                                       old_id, std::this_thread::get_id())) {
15                 hazardPointer = &HazardPointers<T>[i];
16                 break;
17             }
18         }
19         if (!hazardPointer) {
```

```
20                throw std::out_of_range("No hazard pointers available!");
21            }
22        }
23
24        std::atomic<MyNode*>& getPointer() {
25            return hazardPointer->pointer;
26        }
27
28        ~HazardPointerOwner() {
29            hazardPointer->pointer.store(nullptr);
30            hazardPointer->id.store(std::thread::id());
31        }
32    };
```

HazardPointerOwner holds a hazardPointer. This hazardPointer is set in the constructor by traversing all HazardPointers (lines 13 and 14). The compare_exchange_strong call checks in an atomic step if the currently traversed hazard pointer is not set and sets its id to the id of the now executed thread (std::this_thread::get_id()). In the success case, hazardPointer becomes the new hazard pointer returned to the client invoking the member function getPointer (line 24). When all of the hazard pointers of HazardPointers are used, the constructor throws a std::out_of_range exception (line 20). Finally, HazardPointerOwner's destructor sets the hazardPointer to its default state.

14.3 Concurrent Queue

Distilled Information

- Stack and a queue a typical lock-free data structures.
- Deleting a node of a lock-free data structure often face you with one challenge. How can you be sure that no other thread is still using a node you want to delete.
- Hazard pointers are an elegant and performant way to solve the challenge of safe memory reclamation in lock-free data structures.

Further Information

15. Challenges

Programming concurrent applications is inherently complicated. This still holds if you use C++11 and C++14 features, and that is before I mention the memory model. I hope that if I dedicate a whole chapter to the challenges of concurrent programming, you may become more aware of the pitfalls.

15.1 ABA Problem

ABA means you read a value twice, and each time it returns the same value A. Therefore, you conclude that nothing changed in between. However, you missed the fact that the value was updated to B somewhere in between.

Let me first use a simple scenario to introduce the problem.

15.1.0.1 An Analogy

The scenario consists of you sitting in a car and waiting for the traffic light to become green. Green stands in our case for B, and red for A. What's happening?

1. You look at the traffic light, and it is red (A).
2. Because you are bored, you begin to check the news on your smartphone and forget the time.
3. You look once more at the traffic light. Damn, it is still red (A).

Of course, the traffic light became green (B) between your two checks. Therefore, what seems to be one red phase was a complete cycle.

What does this mean for threads (processes)? Now more formally.

1. Thread 1 reads the variable `var` with value A.
2. Thread 1 is preempted, and thread 2 runs.
3. Thread 2 changes the variable `var` from A to B to A.
4. Thread 1 continues to run and checks the value of variable `var` and gets A because of the value A, thread 1 proceeds.

Often that is not a problem, and you can ignore it.

15.1.0.2 Non-critical ABA

The functions `compare_exchange_strong` and `compare_exchange_weak` suffer the ABA problem that can be observed in the `fetch_mult` (line 6). Here, it is non-critical. `fetch_mult` multiplies a `std::atomic<T>&` `shared` by `mult`.

An atomic multiplication with `compare_exchange_strong`

```cpp
// fetch_mult.cpp

#include <atomic>
#include <iostream>

template <typename T>
T fetch_mult(std::atomic<T>& shared, T mult){
  T oldValue = shared.load();
  while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
  return oldValue;
}

int main(){
  std::atomic<int> myInt{5};
  std::cout << myInt << '\n';
  fetch_mult(myInt,5);
  std::cout << myInt << '\n';
}
```

The critical observation is that there is a small-time window between the reading of the old value `T oldValue = shared.load` in line 8 and the new value in line 9. Therefore, another thread can kick in and change the `oldValue` from `oldValue` to another value and back to `oldValue`. The `oldValue` is the A, and another value is the B in ABA.

Often it makes no difference if Read-Operations address the same, unchanged variable. However, in a lock-free concurrent data structure, ABA may have a significant impact.

15.1.0.3 A lock-free data structure

I do not present a lock-free data structure in detail here. I use a lock-free stack that is implemented as a singly linked list. The stack supports only two operations.

1. Pop the top object and return a pointer to it.
2. Push the specified object to stack.

Let me describe the pop operation in pseudo-code to give you an idea of the ABA problem. The pop operation executes the following steps until the operation is successful.

1. Get the head node: `head`
2. Get the subsequent node: `headNext`
3. Make `headNext` to the new head if `head` is still the head of the stack

Here are the first two nodes of the stack:

```
Stack: TOP -> head -> headNext -> ...
```

Let's construct the ABA problem.

15.1.0.4 ABA in Action

Let's start with the following stack:

```
Stack: TOP -> A -> B -> C
```

Thread 1 is active and wants to pop the head of the stack.

- Thread 1 stores

```
head = A
headNext = B
```

Before thread 1 finishes the pop step, thread 2 kicks in.

- Thread 2 pops A

```
Stack: TOP -> B -> C
```

- Thread 2 pops B and deletes B

```
Stack: TOP -> C
```

- Thread 2 pushed A back

```
Stack: TOP -> A -> C
```

Thread 1 is rescheduled and checks if `A == head`. Because of `A == head`, `headNext`, which is B becomes the new head. However, B was already deleted. Therefore, the program has undefined behavior.

There are a few remedies to the ABA problem.

15.1.0.5 Remedies

The conceptional problem of ABA is quite easy to understand. A node such as `B == headNext` was deleted although another node `A == head` was referring to it. The solution to our problem is to get rid of the premature deletion of the node. Here are a few remedies.

15.1.0.5.1 Tagged state reference

You can use the address's low bits to add a tag to each node indicating how often the node has been successfully modified. The result is that compare and swap (CAS) member function eventually fails, although the check returns true. This idea does not solve the issue because the tag bits may eventually wrap around. Architectures that support a double word CAS operation can have a bigger counter.

Tagged state reference are typically used in transactional memory.

The following three techniques are based on the idea of deferred reclamation.

15.1.0.5.2 Garbage Collection

Garbage collection guarantees that the variables are only deleted if it is not needed anymore. This sounds promising but has a significant drawback. Most garbage collectors are not lock-free. Therefore, even if you have a lock-free data structure, the overall system won't be lock-free.

15.1.0.5.3 Hazard Pointers

From Wikipedia: Hazard Pointers[1]:

Each thread keeps a list of hazard pointers in a hazard-pointer system, indicating which nodes the thread is currently accessing. (In many systems, this "list" maybe probably limited to only one or two elements.) Nodes on the hazard pointer list must not be modified or deallocated by any other thread. (…) When a thread wishes to remove a node, it places it on a list of nodes "to be freed later", but does not deallocate the node's memory until no other thread's hazard list contains the pointer. A dedicated garbage-collection thread can do this manual garbage collection (if the list "to be freed later" is shared by all the threads); alternatively, cleaning up the "to be freed" list can be done by each worker thread as part of an operation such as "pop".

15.1.0.6 RCU

RCU stands for **R**ead **C**opy **U**pdate and is a synchronization technique for almost read-only data structures. RCU was created by Paul McKenney and has been used in the Linux Kernel since 2002.

The idea is quite simple and follows the acronym. To modify data, you make a copy of the data and modify that copy. In contrast, all readers work with the original data. If there is no reader, you can safely replace the data structure with its copy.

For more details about RCU, read the article What is RCU, Fundamentally?[2] by Paul McKenney.

🔑 **Two new proposals**

As part of a concurrency toolkit, there are two proposals for future C++ standards: the proposal P0233r0[3] for hazard pointers and the proposal P0461R0[4] for RCU.

---

[1]https://en.wikipedia.org/wiki/Hazard_pointer
[2]https://lwn.net/Articles/262464/
[3]http://www.modernescpp.com/open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0233r0.pdf
[4]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0461r0.pdf

15.2 Blocking Issues

To make my point clear, you have to use a condition variable in combination with a predicate. If you don't, your program may become a victim of a spurious wakeup or lost wakeup.

If you use a condition variable without a predicate, the notifying thread may send its notification before the waiting thread is waiting. Therefore, the waiting thread waits forever. This phenomenon is called a lost wake-up.

Here is the program.

Blocking condition variables

```cpp
// conditionVariableBlock.cpp

#include <iostream>
#include <condition_variable>
#include <mutex>
#include <thread>

std::mutex mutex_;
std::condition_variable condVar;

bool dataReady;


void waitingForWork(){

    std::cout << "Worker: Waiting for work." << '\n';

    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck);
    // do the work
    std::cout << "Work done." << '\n';

}

void setDataReady(){

    std::cout << "Sender: Data is ready."  << '\n';
    condVar.notify_one();

}

int main(){

```

```
34    std::cout << '\n';
35
36    std::thread t1(setDataReady);
37    std::thread t2(waitingForWork);
38
39    t1.join();
40    t2.join();
41
42    std::cout << '\n';
43
44  }
```

By chance, the first invocation of the program works fine. The second invocation locks because the notify call (line 28) happens before the thread t2 (line 37) is waiting (line 19).

A blocking condition variable

Of course, deadlocks and livelocks are side effects of race conditions. A deadlock depends in general on the interleaving of the threads and may sometimes occur or not. A livelock is similar to a deadlock. While a deadlock blocks, a livelock seems to make progress, emphasizing "seems". Think about a transaction in a transactional memory use case. Each time the transaction should be committed, a conflict happens; therefore, a rollback takes place. Here are the details of transactions.

15.3 Breaking of Program Invariants

Program invariants are invariants that should hold for the entire lifetime of your program.

Malicious race condition breaks an invariant of the program. The invariant of the program is that the sum of all balances should be the same amount. Which in our case is 200 euros because each account starts with 100 euro (line 9). I neither want to create money by transferring it, nor do I want to destroy it.

Breaking an invariant of the program

```cpp
1    // breakingInvariant.cpp
2
3    #include <atomic>
4    #include <functional>
5    #include <iostream>
6    #include <thread>
7
8    struct Account{
9      std::atomic<int> balance{100};
10   };
11
12   void transferMoney(int amount, Account& from, Account& to){
13     using namespace std::chrono_literals;
14     if (from.balance >= amount){
15       from.balance -= amount;
16       std::this_thread::sleep_for(1ns);
17       to.balance += amount;
18     }
19   }
20
21    void printSum(Account& a1, Account& a2){
22     std::cout << (a1.balance + a2.balance) << '\n';
23   }
24
25   int main(){
26
27     std::cout << '\n';
28
29     Account acc1;
30     Account acc2;
31
32     std::cout << "Initial sum: ";
33     printSum(acc1, acc2);
34
35     std::thread thr1(transferMoney, 5, std::ref(acc1), std::ref(acc2));
36     std::thread thr2(transferMoney, 13, std::ref(acc2), std::ref(acc1));
37     std::cout << "Intermediate sum: ";
38     std::thread thr3(printSum, std::ref(acc1), std::ref(acc2));
39
40     thr1.join();
41     thr2.join();
42     thr3.join();
43
44     std::cout << "      acc1.balance: " << acc1.balance << '\n';
```

```
45    std::cout << "       acc2.balance: " << acc2.balance << '\n';
46
47    std::cout << "Final sum: ";
48    printSum(acc1, acc2);
49
50    std::cout << '\n';
51
52  }
```

In the beginning, the sum of the accounts is 200 euros. Line 33 displays the sum by using the function printSum in lines 21 - 23. Line 38 makes the invariant visible. Because there is a short sleep of 1ns in line 16, the intermediate sum is 182 euro. In the end, all is fine; each account has the right balance (line 44 and line 45), and the sum is 200 euro (line 48).

Here is the output of the program.

```
Initial sum: 200
Intermediate sum: 182
      acc1.balance: 108
      acc2.balance: 92
Final sum: 200
```

The invariant of the accounts

15.4 Data Races

A data race is a situation in which at least two threads access a shared variable simultaneously. At least one thread tries to modify the variable.

If your program has a data race, it has undefined behavior. This means all outcomes are possible, and therefore, reasoning about the program makes no sense anymore.

Let me show you a program with a data race.
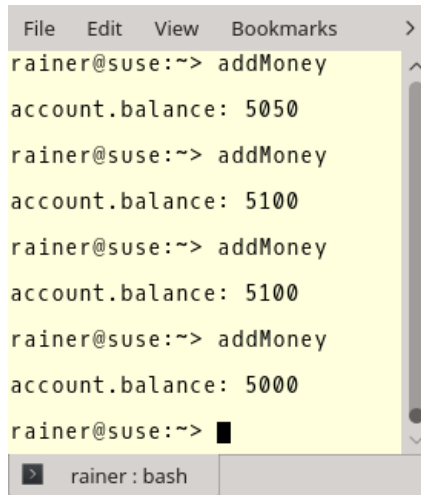
A data race

```cpp
1   // addMoney.cpp
2
3   #include <functional>
4   #include <iostream>
5   #include <thread>
6   #include <vector>
7
8   struct Account{
9     int balance{100};
10  };
11
12  void addMoney(Account& to, int amount){
13    to.balance += amount;
14  }
15
16  int main(){
17
18    std::cout << '\n';
19
20    Account account;
21
22    std::vector<std::thread> vecThreads(100);
23
24
25    for (auto& thr: vecThreads) thr = std::thread(addMoney, std::ref(account), 50);
26
27    for (auto& thr: vecThreads) thr.join();
28
29
30    std::cout << "account.balance: " << account.balance << '\n';
31
32    std::cout << '\n';
33
34  }
```

One hundred threads are adding 50 euros (line 25) to the same account (line 20). They use the function addMoney. The critical observation is that the writing to the account is done without synchronization. Therefore we have a data race, and the result is not valid. This is undefined behavior, and the final balance (line 30) differs between 5000 and 5100 euro.

A data race causes incorrect balances

15.5 Deadlocks

A deadlock is a state in which at least one thread is blocked forever because it waits for the release of a resource, it does never get.

There are two main reasons for deadlocks:

1. A mutex has not been unlocked.
2. You lock your mutexes in a different order.

For overcoming the second issue, techniques such as lock hierachies[5] are used in classical C++.

For the details about deadlocks and how to overcome them with modern C++, read the subsection issues of mutexes and locks.

---

[5]http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/2008/0801/071201hs01/071201hs01.html

Locking a non-recursive mutex more than once

Locking a non-recursive mutex more than once is undefined behavior.

Locking more than once

```cpp
// lockTwice.cpp

#include <iostream>
#include <mutex>

int main(){

    std::mutex mut;

    std::cout << '\n';

    std::cout << "first lock call" << '\n';

    mut.lock();

    std::cout << "second lock call" << '\n';

    mut.lock();

    std::cout << "third lock call" << '\n';

}
```

Typically, you get a deadlock.

A deadlock with non-recursive mutexes

15.6 False Sharing

When a processor reads a variable such as an `int` from main memory, it reads more than the size of an `int` from memory. The processor reads an entire cache line (typically 64 bytes) from memory. False

sharing occurs if two threads read at the same time different variables a and b that are located on the same cache line. Although a and b are logically separated, they are physically connected. An expensive hardware synchronization on the cache line is necessary because a and b share the same. The result is that you get the right results, but the performance of your concurrent application decreases. Precisely this phenomenon happen in the following program.

False sharing

```cpp
// falseSharing.cpp

#include <algorithm>
#include <chrono>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

constexpr long long size{100000000};

struct Sum{
    long long a{0};
    long long b{0};
};

int main(){

    std::cout << '\n';

    Sum sum;

    std::cout << &sum.a << '\n';
    std::cout << &sum.b << '\n';

    std::cout << '\n';

    std::vector<int> randValues, randValues2;
    randValues.reserve(size);
    randValues2.reserve(size);

    std::mt19937 engine;
    std::uniform_int_distribution<> uniformDist(1,10);

    int randValue;
    for (long long i = 0; i < size; ++i){
        randValue = uniformDist(engine);
        randValues.push_back(randValue);
```
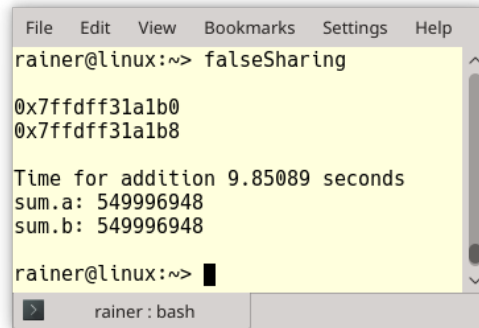
```
39            randValues2.push_back(randValue);
40        }
41
42        auto sta = std::chrono::steady_clock::now();
43
44        std::thread t1([&sum, &randValues]{
45            for (auto val: randValues) sum.a += val;
46        });
47
48        std::thread t2([&sum, &randValues2]{
49            for (auto val: randValues2) sum.b += val;
50        });
51
52        t1.join(), t2.join();
53
54        std::chrono::duration<double> dur= std::chrono::steady_clock::now() - sta;
55        std::cout << "Time for addition " << dur.count()
56                << " seconds" << '\n';
57
58        std::cout << "sum.a: " << sum.a << '\n';
59        std::cout << "sum.b: " << sum.b << '\n';
60
61        std::cout << '\n';
62
63    }
```
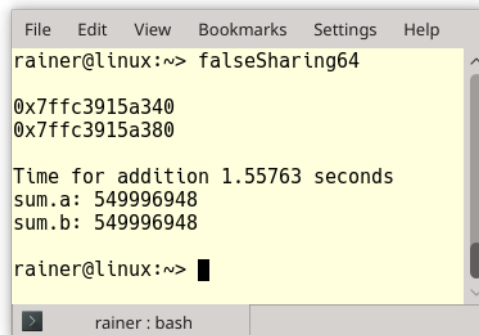
The variables a and b in lines 13 and 14 share the same cache line. Thread t1 (line 44) and the thread t2 use both variables to concurrently add up the vectors randValues and randValues2. Both vectors have 100 million integers between 1 and 10. The program's output shows interesting facts. a and b are aligned at 8-byte boundaries because this is the alignment for long long ints on my system.

False sharing

What happens if I change the alignment of a and b to 64 bytes? 64 is the size of the cache line on my system. Here is the slight change I have to make to the struct Sum. I don't use a seed for my random number generator; therefore, I get the same random numbers each time.

```cpp
struct Sum{
    alignas(64) long long a{0};
    alignas(64) long long b{0};
};
```


False sharing resolved

Now a and b are aligned at 64-byte boundaries, and the program becomes more than six times faster. The reason is that a and b are now on different cache lines.

🔑 The optimizer detects the false sharing

If I compile the last programs with maximum optimization, my optimizer detects the false sharing and eliminate it. This means that I get the same performance numbers with and without false sharing. This also holds for Windows. Here are the optimized numbers.

False sharing resolved by the optimizer

🔑 `std::hardware_destructive_interference_size` and `std::hardware_constructive_interference_size` with C++17

The functions `std::hardware_destructive_interference_size` and `std::hardware_-constructive_interference_size` let you deal in a portable way with the cache line size. `std::hardware_destructive_interference_size` returns the minimum offset between two objects to avoid false sharing and `std::hardware_constructive_interference_size` returns the maximum size of contiguous memory to promote true sharing.

With C++17, `Sum` can be written in a platform-independent way.

```cpp
struct Sum{
  alignas(std::hardware_destructive_interference_size) long long a{0};
  alignas(std::hardware_destructive_interference_size) long long b{0};
};
```

15.7 Lifetime Issues of Variables

Creating a C++ example with lifetime related issues is relatively easy. Let the created thread t run in the background (i.e. it was detached with a call to t.detach()) and let it be only half completed. The creator thread doesn't wait until its child is done. In this case, you have to be extremely careful not to use anything in the child thread that belongs to the creator thread.

Lifetime issues of variables

```cpp
// lifetimeIssues.cpp

#include <iostream>
#include <string>
#include <thread>

int main(){

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

  std::string mess{"Child thread"};

  std::thread t([&mess]{ std::cout << mess << '\n';});
  t.detach();

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

}
```

This is too simple. The thread t is using std::cout and the variable mess. Both belong to the main thread. The effect is that we don't see the output of the child thread in the second run. Only "Begin:" (line 9) and "End:" (line 16) are printed.

Lifetime issues of variables

15.8 Moving Threads

Moving threads make the lifetime issues of threads even harder.

A thread supports the move semantic but not the copy semantic. The reason being the copy constructor of `std::thread` is set to `delete`: `thread(const thread&) = delete;`. Imagine what happens if you copy a thread while the thread is holding a lock.

Let's move a thread.

Erronous moving a thread

```cpp
// threadMoved.cpp

#include <iostream>
#include <thread>
#include <utility>

int main(){

    std::thread t([]{std::cout << std::this_thread::get_id();});
    std::thread t2([]{std::cout << std::this_thread::get_id();});

    t = std::move(t2);
    t.join();
    t2.join();

}
```

Both threads t and t2 should do their simple job: printing their IDs. In addition to that, thread t2 is moved to t (line 12). In the end, the main thread takes care of its children and joins them. But wait, the result is very different from my expectations:
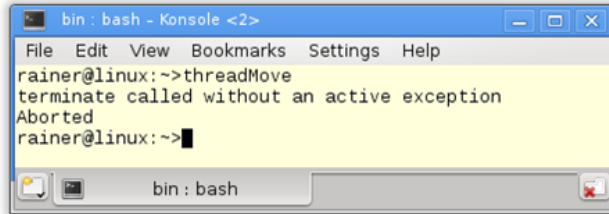
Erronous moving a thread

What is going wrong? We have two issues:

1. By moving the thread t2, t gets a new callable unit, and its destructor is called. As a result, t's destructor calls std::terminate, because it is still joinable.

2. Thread t2 has no associated callable unit. The invocation of join on a thread without callable unit leads to the exception std::system_error.

Knowing this, fixing the errors is straightforward.

Moving a thread
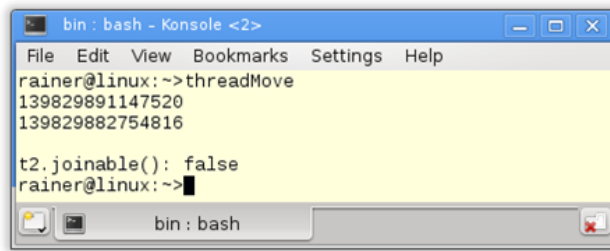
```
1   // threadMovedFixed.cpp
2
3   #include <iostream>
4   #include <thread>
5   #include <utility>
6
7   int main(){
8
9       std::thread t([]{std::cout << std::this_thread::get_id() << '\n';});
10      std::thread t2([]{std::cout << std::this_thread::get_id() << '\n';});
11
12      t.join();
13      t = std::move(t2);
14      t.join();
15
16      std::cout << "\n";
17      std::cout << std::boolalpha << "t2.joinable(): " << t2.joinable() << '\n';
18
19  }
```

The result is that thread t2 is not joinable anymore.

Erronous moving resolved

15.9 Race Conditions

A race condition is a situation in which an operation's result depends on the interleaving of certain individual operations.

Race conditions are challenging to spot. It depends on the interleaving of the threads whether they occur. That means the number of cores, the utilization of your system, or your executable optimization level may all be reasons why a race condition appears or does not.

Race conditions are not bad per se. It is the nature of threads that they interleave in different ways, but this can often cause serious problems. In this case, I call them malign race conditions. Typical effects of malign race conditions are data races, breaking of program invariants, blocking issues of threads, or lifetime issues of variables.

16. The Time Library

A book dealing with concurrency in modern C++ would not be complete without writing a chapter about the time library. The time library consists of three parts: time point, time duration, and clock. They depend on each other.

16.1 The Interplay of Time Point, Time Duration, and Clock

Time point
> The time point is given by its starting point - the so-called epoch[1] - and the time that has elapsed since the epoch (expressed as a time duration)."

Time duration
> The time duration is the difference between two time points. It is measured in the number of time ticks.

Clock
> The clock consists of a starting point and a time tick. This information enables you to calculate the current time.

You can compare time points. When you add a time duration to a time point, you get a new time point. The time tick is the accuracy of the clock in which you measure the time duration. The birth of Jesus is, in my culture, the starting time point, and a year is a typical time tick.

I illustrate the three concepts using the lifetime of Dennis Ritchie[2]. The creator of C died in 2011. For the sake of simplicity, I'm only interested in the years.

Here is the lifetime.
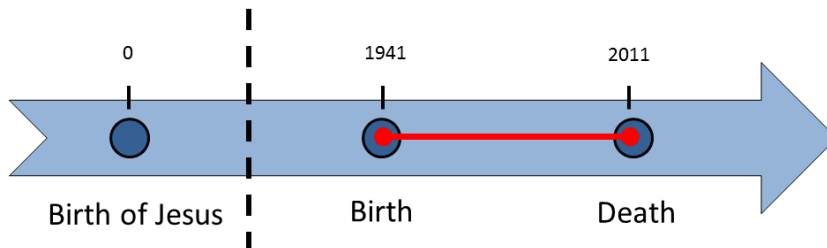
The lifetime of Dennis Ritchie

The birth of Jesus is our epoch. The time points 1941, and 2011 are defined by the epoch and the time duration. Of course, the epoch is also a time point. When I subtract 1941 from 2011, I get the time duration. This dime duration is measured to an accuracy of one year in our example. Dennis Ritchie died at 70.

Let's dive deeper into the components of the time library.

---

[1]https://en.wikipedia.org/wiki/Epoch_(reference_date)
[2]https://en.wikipedia.org/wiki/Dennis_Ritchie

16.2 Time Point

The time point `std::chrono::time_point` is defined by the starting point (epoch) and the additional time duration. The class template consists of two components: clock and time duration. By default, the time duration is derived from the clock.

The class template std::chrono::time_point

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

The following four special time points depend on the clock:

- **epoch**: the starting point of the clock
- **now**: the current time
- **min**: the minimum time point that the clock can have
- **max**: the maximum time point that the clock can have

The accuracy of the minimum and maximum time point depends on the clock used: `std::system::system_-clock`, `std::chrono::steady_clock`, or `std::chrono::high_resolution_clock`.

C++ gives no guarantee about the accuracy, the starting point or the valid time range of a clock. The starting point of `std::chrono::system_clock` is typically 1st January 1970, the so-called UNIX-epoch[3]. It holds further that `std::chrono::high_resolution_clock` has the highest accuracy.

16.2.1 From Time Point to Calendar Time

Thanks to `std::chrono::system_clock::to_time_t` you can convert a time point that internally uses `std::chrono::system_clock` to an object of type `std::time_t`. Further conversion of the `std::time_t` object with the function `std::gmtime`[4] gives you the calendar time, expressed in Coordinated Universal Time[5] (UTC). In the end, this calendar time can be used as the input for the function `std::asctime`[6] to get a textual representation of the calendar time.

---

[3]https://en.wikipedia.org/wiki/Unix_time
[4]http://en.cppreference.com/w/cpp/chrono/c/gmtime
[5]https://en.wikipedia.org/wiki/Coordinated_Universal_Time
[6]http://en.cppreference.com/w/cpp/chrono/c/asctime

Display the calendar time
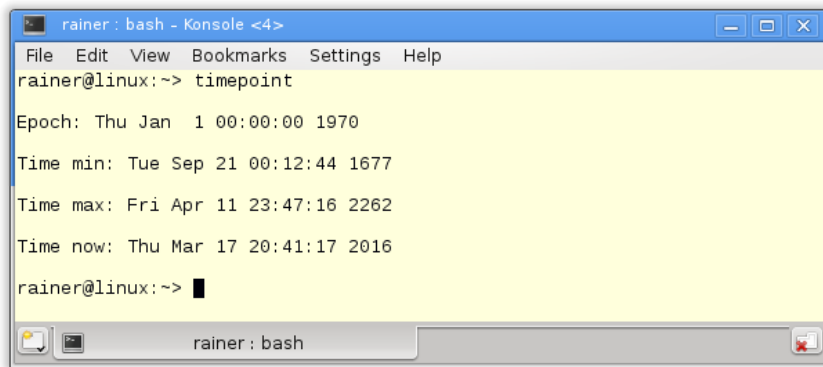
```cpp
1    // timepoint.cpp
2
3    #include <chrono>
4    #include <ctime>
5    #include <iostream>
6    #include <string>
7
8    int main(){
9
10       std::cout << '\n';
11
12       std::chrono::time_point<std::chrono::system_clock> sysTimePoint;
13       std::time_t tp= std::chrono::system_clock::to_time_t(sysTimePoint);
14       std::string sTp= std::asctime(std::gmtime(&tp));
15       std::cout << "Epoch: " << sTp << '\n';
16
17       tp= std::chrono::system_clock::to_time_t(sysTimePoint.min());
18       sTp= std::asctime(std::gmtime(&tp));
19       std::cout << "Time min: " << sTp << '\n';
20
21       tp= std::chrono::system_clock::to_time_t(sysTimePoint.max());
22       sTp= std::asctime(std::gmtime(&tp));
23       std::cout << "Time max: " << sTp << '\n';
24
25       sysTimePoint= std::chrono::system_clock::now();
26       tp= std::chrono::system_clock::to_time_t(sysTimePoint);
27       sTp= std::asctime(std::gmtime(&tp));
28       std::cout << "Time now: " << sTp << '\n';
29
30    }
```

The output of the program shows the valid range of `std::chrono::system_clock`. On my Linux PC `std::chrono::system_clock` has the UNIX-epoch as starting point and can have time points between the years 1677 and 2262.

The valid range of std::chrono::system_clock

You can add time durations to time points to get new time points. Adding time durations beyond the valid time range is undefined behavior.

16.2.2 Cross the valid Time Range

The following example uses the current time and adds or subtracts 1000 years. For the sake of simplicity, I ignore leap years and assume that a year has 365 days.

Crossing the valid time range
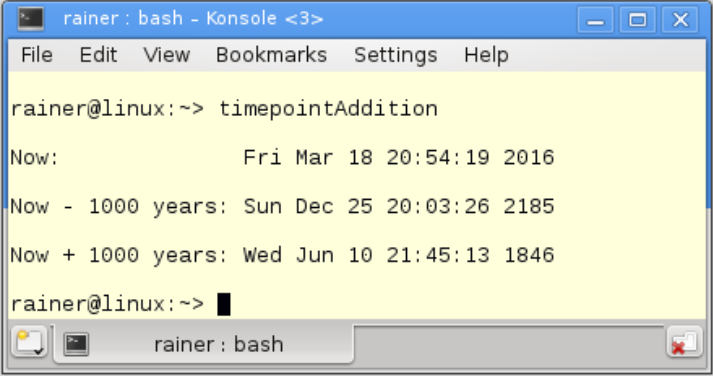
```cpp
// timepointAddition.cpp

#include <chrono>
#include <ctime>
#include <iostream>
#include <string>

using namespace std::chrono;
using namespace std;

string timePointAsString(const time_point<system_clock>& timePoint){
  time_t tp= system_clock::to_time_t(timePoint);
  return asctime(gmtime(&tp));
}

int main(){

  cout << '\n';

  time_point<system_clock> nowTimePoint= system_clock::now();
  cout << "Now:                    " << timePointAsString(nowTimePoint) << '\n';
```

```
22
23      const auto thousandYears=  hours(24*365*1000);
24      time_point<system_clock>  historyTimePoint= nowTimePoint - thousandYears;
25      cout << "Now - 1000 years: " << timePointAsString(historyTimePoint) << '\n';
26
27      time_point<system_clock>  futureTimePoint= nowTimePoint + thousandYears;
28      cout << "Now + 1000 years: " << timePointAsString(futureTimePoint) << '\n';
29
30  }
```

For readability, I introduced the namespace `std::chrono`. The program's output shows that an overflow of the time points in lines 25 and 28 causes incorrect results. Subtracting 1000 years from the current time point gives a time point in the future; adding 1000 years to the current time point gives a time point in the past, respectively.

Overflow of the valid time range

The difference between two time points is a time duration. Time durations support the basic arithmetic and can be displayed in different time ticks.

16.3 Time Duration

Time duration `std::chrono::duration` is a class template that consists of the type of the tick `Rep` and the length of a tick `Period`.

The class template std::chrono::duration

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

The tick length is by default `std::ratio<1>`. `std::ratio<1>` stands for a second and can also be written as `std::ratio<1, 1>`. The rest is quite easy. `std::ratio<60>` is a minute and `std::ratio<1,1000>` a millisecond. When the type of `Rep` is a floating-point number, you can use it to hold fractions of time ticks.

C++11 predefines the most important time durations:

Important time durations

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

How much time has passed since the UNIX epoch (1.1.1970)? Thanks to type aliases for the different time durations, I can answer the question quite easily. In the following example, I ignore leap years and assume that a year has 365 days.

Crossing the valid time range

```cpp
1   // timeSinceEpoch.cpp
2
3   #include <chrono>
4   #include <iostream>
5
6   using namespace std;
7
8   int main(){
9
10     cout << fixed << '\n';
11
12     cout << "Time since 1.1.1970:\n" << '\n';
```
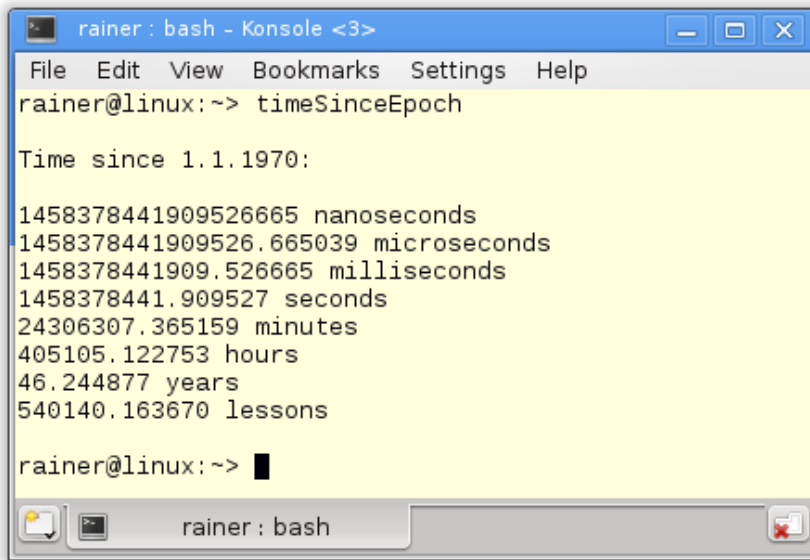
```
13
14      const auto timeNow= chrono::system_clock::now();
15      const auto duration= timeNow.time_since_epoch();
16      cout << duration.count() << " nanoseconds " << '\n';
17
18      typedef chrono::duration<long double, ratio<1, 1000000>> MyMicroSecondTick;
19      MyMicroSecondTick micro(duration);
20      cout << micro.count() << " microseconds" << '\n';
21
22      typedef chrono::duration<long double, ratio<1, 1000>> MyMilliSecondTick;
23      MyMilliSecondTick milli(duration);
24      cout << milli.count() << " milliseconds" << '\n';
25
26      typedef chrono::duration<long double> MySecondTick;
27      MySecondTick sec(duration);
28      cout << sec.count() << " seconds " << '\n';
29
30      typedef chrono::duration<double, ratio<60>> MyMinuteTick;
31      MyMinuteTick myMinute(duration);
32      cout << myMinute.count() << " minutes" << '\n';
33
34      typedef chrono::duration<double, ratio<60*60>> MyHourTick;
35      MyHourTick myHour(duration);
36      cout << myHour.count() << " hours" << '\n';
37
38      typedef chrono::duration<double, ratio<60*60*24*365>> MyYearTick;
39      MyYearTick myYear(duration);
40      cout << myYear.count() << " years" << '\n';
41
42      typedef chrono::duration<double, ratio<60*45>> MyLessonTick;
43      MyLessonTick myLesson(duration);
44      cout << myLesson.count() << " lessons" << '\n';
45
46      cout << '\n';
47
48  }
```

The typical time durations are microsecond (line 18), millisecond (line 22), second (line 26), minute (line 30), hour (line 34), and year (line 38). Also, I define the German school hour (45 min) in line 42.

Time since epoch

It's pretty convenient to calculate with time durations, as the next section illustrates.

16.3.1 Calculations

The time durations support basic arithmetic operations. This means that you can multiply or divide a time duration by a number. Of course, you can compare time durations. I explicitly want to emphasize that all these calculations and comparisons respect the units.

With the C++14 standard, it gets even better. The C++14 standard supports the typical time literals.

Predefined time literals

Type	Suffix	Example
std::chrono::hours	h	5h
std::chrono::minutes	min	5min
std::chrono::seconds	s	5s
std::chrono::milliseconds	ms	5ms
std::chrono::microseconds	us	5us
std::chrono::nanoseconds	ns	5ns

How much time does my 17 years old son Marius spend during a typical school day? I answer the question in the following example and show the result in various time duration formats.

A typical school-day in various time durations
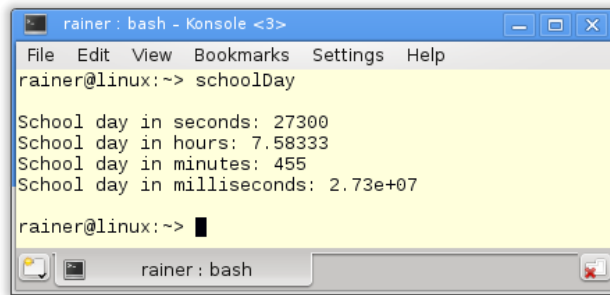
```cpp
// schoolDay.cpp

#include <iostream>
#include <chrono>

using namespace std::literals::chrono_literals;
using namespace std::chrono;
using namespace std;

int main(){

  cout << '\n';

  constexpr auto schoolHour= 45min;

  constexpr auto shortBreak= 300s;
  constexpr auto longBreak= 0.25h;

  constexpr auto schoolWay= 15min;
  constexpr auto homework= 2h;

  constexpr auto schoolDaySec= 2*schoolWay + 6 * schoolHour + 4 * shortBreak +
                               longBreak + homework;

  cout << "School day in seconds: " << schoolDaySec.count() << '\n';

  constexpr duration<double, ratio<3600>> schoolDayHour = schoolDaySec;
  constexpr duration<double, ratio<60>> schoolDayMin = schoolDaySec;
  constexpr duration<double, ratio<1,1000>> schoolDayMilli= schoolDaySec;

  cout << "School day in hours: " << schoolDayHour.count() << '\n';
  cout << "School day in minutes: " << schoolDayMin.count() << '\n';
  cout << "School day in milliseconds: " << schoolDayMilli.count() << '\n';

  cout << '\n';

}
```

I have time durations for a German school hour (line 14), for a short break (line 16), for an extended break (line 17), for Marius's way to school (line 19), and his homework (line 20). The result of the calculation schoolDaysInSeconds (line 22) is available at compile time.

A typical school-day in various time durations

Evaluation at compile time

The time literals (lines 14 - 20), the `schoolDaySec` in line 22, and the various durations (lines 28 - 30) are all constant expressions (`constexpr`). Therefore, all values are evaluated at compile time. Just the output is performed at runtime.

The accuracy of the time tick is dependent on the clock used. In C++ we have the clocks `std::chrono::system_clock`, `std::chrono::steady_clock`, and `std::chrono::high_resolution_clock`.

16.4 Clocks

The fact that there are three different types of clocks begs the question: What are the differences?

- **std::chrono::sytem_clock**: is the system-wide real-time clock (wall-clock[7]). The clock has the auxiliary functions `to_time_t` and `from_time_t` to convert time points into calendar time.
- **std::chrono::steady_clock**: is the only clock to provide the guarantee that you can not adjust it. Therefore, `std::chrono::steady_clock` is the preferred clock to measure time intervals.
- **std::chrono::high_resolution_clock**: is the clock with the highest accuracy but it can be simply an alias for the clocks `std::chrono::system_clock` or `std::chrono::steady_clock`.

> 🔑 **No guarantees about the accuracy, starting point, and valid time range**
>
> The C++ standard provides no guarantee about the accuracy, the starting point or the valid time range of the clocks. Typically, the starting point of `std::chrono:system_clock` is the 1.1.1970, the so-called UNIX-epoch, while for `std::chrono::steady_clock` it is usually the boot time of your PC.

16.4.1 Accuracy and Steadiness

It is pretty interesting to know which clocks are steady and what accuracy they provide. Steady means that the clock cannot be adjusted. You can get the answers directly from the clocks.

Accuracy and steadiness of the three clocks

```
1   // clockProperties.cpp
2
3   #include <chrono>
4   #include <iomanip>
5   #include <iostream>
6
7   using namespace std::chrono;
8   using namespace std;
9
10  template <typename T>
11  void printRatio(){
12      cout << "  precision: " << T::num << "/" << T::den << " second " << '\n';
13      typedef typename ratio_multiply<T,kilo>::type MillSec;
14      typedef typename ratio_multiply<T,mega>::type MicroSec;
15      cout << fixed;
```

---

[7]https://en.wikipedia.org/wiki/Wall-clock_time

```
16      cout << "                    " << static_cast<double>(MillSec::num)/MillSec::den
17                                    << " milliseconds " << '\n';
18      cout << "                    " << static_cast<double>(MicroSec::num)/MicroSec::den
19                                    << " microseconds " << '\n';
20  }
21
22  int main(){
23
24      cout << boolalpha << '\n';
25
26      cout << "std::chrono::system_clock: " << '\n';
27      cout << "  is steady: " << system_clock::is_steady << '\n';
28      printRatio<chrono::system_clock::period>();
29
30      cout << '\n';
31
32      cout << "std::chrono::steady_clock: " << '\n';
33      cout << "  is steady: " << chrono::steady_clock::is_steady << '\n';
34      printRatio<chrono::steady_clock::period>();
35
36      cout << '\n';
37
38      cout << "std::chrono::high_resolution_clock: " << '\n';
39      cout << "  is steady: " << chrono::high_resolution_clock::is_steady
40          << '\n';
41      printRatio<chrono::high_resolution_clock::period>();
42
43      cout << '\n';
44
45  }
```
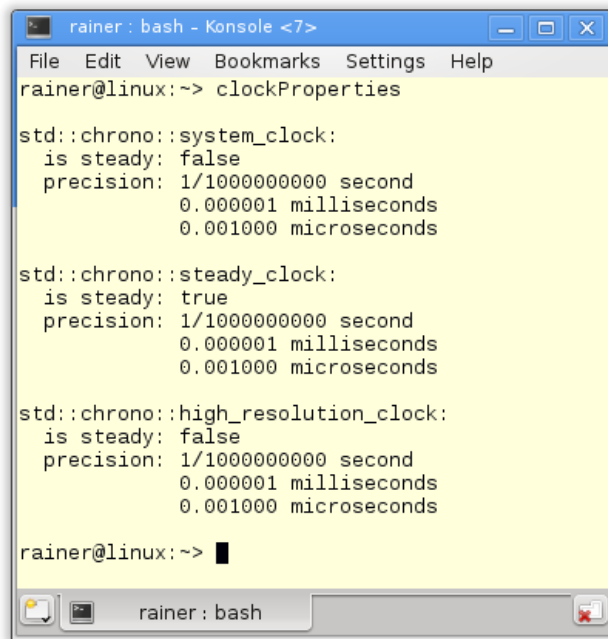
I show in lines 27, 33, and 39 for each clock whether it is steady. The function printRatio (lines 10 - 20) is more challenging to read. First, I display the accuracy of the clocks as a fraction with the unit in seconds. Additionally, I use the function template std::ratio_multiply and the constants std::kilo and std::mega to adjust the units to milliseconds and microseconds displayed as floating-point numbers. You can get the details of the calculation at compile time at cppreference.com[8].

The output on Linux differs from that on Windows. std::chrono::system_clock is far more accurate on Linux; std::chrono::high_resultion_clock is steady on Windows.

---

[8]http://en.cppreference.com/w/cpp/numeric/ratio

Accuracy and steadiness of the three clocks on Linux

Accuracy and steadiness of different clocks on Windows

Although the C++ standard doesn't specify the epoch of the clock, you can calculate it.

16.4.2 Epoch

Thanks to the auxiliary function `time_since_epoch`[9], each clock returns how much time has passed since the epoch.

Calculating the epoch for each clock

```cpp
1    // now.cpp
2
3    #include <chrono>
4    #include <iomanip>
5    #include <iostream>
6
7    using namespace std::chrono;
8
9    template <typename T>
10   void durationSinceEpoch(const T dur){
11       std::cout << "    Counts since epoch:  " << dur.count() << '\n';
12       typedef duration<double, std::ratio<60>> MyMinuteTick;
13       const MyMinuteTick myMinute(dur);
14       std::cout << std::fixed;
15       std::cout << "    Minutes since epoch: "<< myMinute.count() << '\n';
16       typedef duration<double, std::ratio<60*60*24*365>> MyYearTick;
17       const MyYearTick myYear(dur);
18       std::cout << "    Years since epoch:   " << myYear.count() << '\n';
19   }
20
21   int main(){
22
23       std::cout << '\n';
24
25       system_clock::time_point timeNowSysClock = system_clock::now();
26       system_clock::duration timeDurSysClock= timeNowSysClock.time_since_epoch();
27       std::cout << "system_clock: " << '\n';
28       durationSinceEpoch(timeDurSysClock);
29
30       std::cout << '\n';
31
32       const auto timeNowStClock = steady_clock::now();
33       const auto timeDurStClock= timeNowStClock.time_since_epoch();
34       std::cout << "steady_clock: " << '\n';
35       durationSinceEpoch(timeDurStClock);
36
37       std::cout << '\n';
```

---

[9]http://en.cppreference.com/w/cpp/chrono/time_point/time_since_epoch
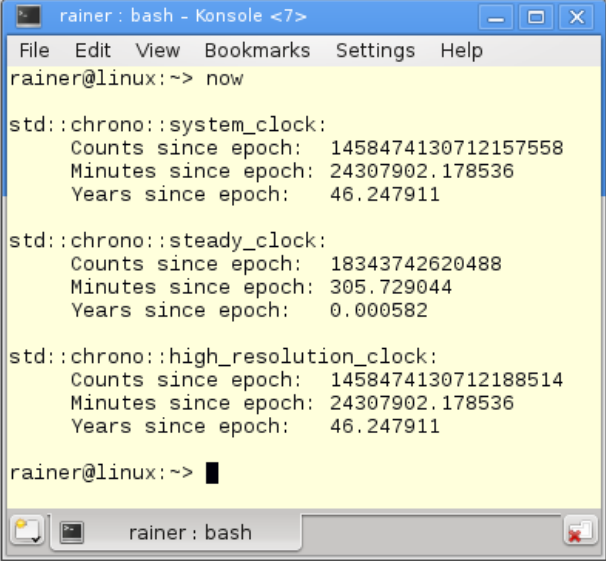
```
38
39      const auto timeNowHiRes = high_resolution_clock::now();
40      const auto timeDurHiResClock= timeNowHiRes.time_since_epoch();
41      std::cout << "high_resolution_clock: " << '\n';
42      durationSinceEpoch(timeDurHiResClock);
43
44      std::cout << '\n';
45
46  }
```

The variables `timeDurSysClock` (line 26), `timeDurStClock` (line 33), and `timeDurHiResClock` (line 40) contain the amount of time that has passed since the starting point of the corresponding clock. Without automatic type deduction with `auto`, the exact types of the time point and time duration are extremely verbose to write. In the function `durationSinceEpoch` (lines 9 - 19) I display the time duration in different resolutions. First, I display the number of time ticks (line 11), then the number of minutes (line 15), and at the end of the years (lines 18) since the epoch. All values depend on the clock used. For the sake of simplicity, I ignore leap years and assume that a year has 365 days.

Once more, the results are different on Linux and Windows.

The epoch for each clock on Linux

The epoch for each clock on Windows

To draw the right conclusion, I have to mention that my Linux PC had been running for about 5 hours (305 minutes), and my Windows PC had been running for more than 6 hours (391 minutes).

`std::chrono::system_clock` and `std::chrono::high_resolution_clock` have the UNIX-epoch as starting point on my linux PC. The starting point of `std::chrono::steady_clock` is the boot time of my PC. While it seems that `std::high_resolution_clock` is an alias for `std::system_clock` on Linux, `std::high_resolution_clock` seems to be an alias for `std::steady_clock` on Windows. This conclusion is in accordance with the result from the previous subsection Accuracy and Steadiness.

Thanks to the time library, you can put a thread to sleep. The arguments of the sleep and wait functions are time points or time durations.

16.5 Sleep and Wait

One crucial feature that multithreading components such as threads, locks, condition variables, and futures have in common is the notion of time.

16.5.0.1 Conventions

The member functions for handling time in multithreading programs follow a simple convention. Member functions ending with _for have to be parametrized by a time duration; member functions ending with _until by a time point. Here is a concise overview of the member functions dealing with sleeping, blocking, and waiting.

Member functions for sleeping, blocking, and waiting

Multithreading Component	_until	_for
std::thread th	th.sleep_until(in2min)	th.sleep_for(2s)
std::unique_lock lk	lk.try_lock_until(in2min)	lk.try_lock(2s)
std::condition_variable cv	cv.wait_until(in2min)	cv.wait_for(2s)
std::future fu	fu.wait_until(in2min)	fu.wait_for(2s)
std::shared_future shFu	shFu.wait(in2min)	shFu.wait_for(2s)

in2min stands for a time 2 minutes in the future. 2s is a time duration of 2 seconds. Although I use auto in the initialization of the time point in2min, the following is still verbose:

Defining a time point

```
auto in2min= std::chrono::steady_clock::now() + std::chrono::minutes(2);
```

Time literals from C++14 come to our rescue when using time durations. 2s stands for 2 seconds.

Let's look at different waiting strategies.

16.5.0.2 Various waiting strategies

The main idea of the following program is that the promise provides its result for four shared futures. That is possible because more than one shared_future can wait to notify the same promise. Each future has a different waiting strategy. Both the promise and every future are executed in different threads. For simplicity reasons I speak in the rest of this subsection only about a waiting thread, although it is the corresponding future that is waiting. Here are the details of the promises and the futures.

Here are the strategies for the four waiting threads:

- **consumeThread1**: waits up to 4 seconds for the result of the promise.
- **consumeThread2**: waits up to 20 seconds for the result of the promise.

- **consumeThread3**: asks the promise for the result and goes back to sleep for 700 milliseconds.
- **consumeThread4**: asks the promise for the result and goes back to sleep. Its sleep duration starts with one millisecond and doubles each time.

Here is the program.

Various waiting strategies

```cpp
// sleepAndWait.cpp

#include <utility>
#include <iostream>
#include <future>
#include <thread>
#include <utility>

using namespace std;
using namespace std::chrono;

mutex coutMutex;

long double getDifference(const steady_clock::time_point& tp1,
                          const steady_clock::time_point& tp2){
    const auto diff= tp2 - tp1;
    const auto res= duration <long double, millT> (diff).count();
    return res;
}

void producer(promise<int>&& prom){
    cout << "PRODUCING THE VALUE 2011\n\n";
    this_thread::sleep_for(seconds(5));
    prom.set_value(2011);
}

void consumer(shared_future<int> fut,
              steady_clock::duration dur){
    const auto start = steady_clock::now();
    future_status status= fut.wait_until(steady_clock::now() + dur);
    if ( status == future_status::ready ){
        lock_guard<mutex> lockCout(coutMutex);
        cout << this_thread::get_id() << " ready => Result: " << fut.get()
            << '\n';
    }
    else{
        lock_guard<mutex> lockCout(coutMutex);
        cout << this_thread::get_id() << " stopped waiting." << '\n';
```

```
39            }
40        const auto end= steady_clock::now();
41        lock_guard<mutex> lockCout(coutMutex);
42        cout << this_thread::get_id() << " waiting time: "
43            << getDifference(start,end) << " ms" << '\n';
44    }
45
46    void consumePeriodically(shared_future<int> fut){
47        const auto start = steady_clock::now();
48        future_status status;
49        do {
50            this_thread::sleep_for(milliseconds(700));
51            status = fut.wait_for(seconds(0));
52            if (status == future_status::timeout) {
53                lock_guard<mutex> lockCout(coutMutex);
54                cout << "     " << this_thread::get_id()
55                    << " still waiting." << '\n';
56            }
57            if (status == future_status::ready) {
58                lock_guard<mutex> lockCout(coutMutex);
59                cout << "     " << this_thread::get_id()
60                    << " waiting done => Result: " << fut.get() << '\n';
61            }
62        } while (status != future_status::ready);
63        const auto end= steady_clock::now();
64        lock_guard<mutex> lockCout(coutMutex);
65        cout << "     " << this_thread::get_id() << " waiting time: "
66            << getDifference(start,end) << " ms" << '\n';
67    }
68
69    void consumeWithBackoff(shared_future<int> fut){
70        const auto start = steady_clock::now();
71        future_status status;
72        auto dur= milliseconds(1);
73        do {
74            this_thread::sleep_for(dur);
75            status = fut.wait_for(seconds(0));
76            dur *= 2;
77            if (status == future_status::timeout) {
78                lock_guard<mutex> lockCout(coutMutex);
79                cout << "          " << this_thread::get_id()
80                    << " still waiting." << '\n';
81            }
82            if (status == future_status::ready) {
83                lock_guard<mutex> lockCout(coutMutex);
```

```
84                cout << "           " << this_thread::get_id()
85                     << " waiting done => Result: " << fut.get() << '\n';
86          }
87      } while (status != future_status::ready);
88      const auto end= steady_clock::now();
89      lock_guard<mutex> lockCout(coutMutex);
90      cout << "           " << this_thread::get_id()
91          << " waiting time: " << getDifference(start,end) << " ms" << '\n';
92  }
93
94  int main(){
95
96      cout << '\n';
97
98      promise<int> prom;
99      shared_future<int> future= prom.get_future();
100     thread producerThread(producer, move(prom));
101
102     thread consumerThread1(consumer, future, seconds(4));
103     thread consumerThread2(consumer, future, seconds(20));
104     thread consumerThread3(consumePeriodically, future);
105     thread consumerThread4(consumeWithBackoff, future);
106
107     consumerThread1.join();
108     consumerThread2.join();
109     consumerThread3.join();
110     consumerThread4.join();
111     producerThread.join();
112
113     cout << '\n';
114
115 }
```

I create the promise in the main function (line 98), use the promise to create the associated future (line 99), and move the promise into a separate thread (line 100). I have to move the promise into the thread because it does not support the copy semantic. That is not necessary for the shared futures (lines 102 - 105); they support the copy semantic and can hence be copied.

Before I talk about the thread's work package, let me say a few words about the auxiliary function getDifference (lines 14 - 19). The function takes two time points and returns the time duration between this two timepoints in milliseconds. I use the function a few times.

What about the five created threads?

- **producerThread**: executes the function producer (lines 21 - 25) and publishes its result 2011 after 5 seconds of sleep. This is the result the futures are waiting for.

- **consumerThread1**: executes the function `consumer` (lines 27 - 44). The thread is waiting for at most 4 seconds (line 30) before it continues with its work. This waiting period is not long enough to get the result from the promise.

- **consumerThread2**: executes the function `consumer` (lines 27 - 44). The thread is waiting at most 20 seconds before it continues with its work.

- **consumerThread3**: executes the function `consumePeriodically` (lines 46 - 67). It sleeps for 700 milliseconds (line 50) and asks for the result of the promise (line 60). Because of the `std::chrono::seconds(0)` in line 51, there is no waiting. If the result of the calculation is available, it is displayed in line 60.

- **consumerThread4**: executes the function `consumeWithBackoff` (lines 69 - 92). It sleeps in the first iteration 1 second and doubles its sleeping period every iteration. Otherwise, its strategy is similar to the strategy of `consumerThread3`.

Now to the synchronization of the program. Both the clock determining the current time and `std::cout` are shared variables, but no synchronization is necessary. Firstly, the member function call `std::chrono::steady_clock::now()` is thread-safe (for example in lines 30 and 40), secondly, the C++ runtime guarantees that the characters are written *thread-safe* to `std::cout`. I only used a `std::lock_guard` to wrap `std::cout` (for example in lines 32, 37, and 41).

Although the threads write one after the other to `std::cout`, the output is not easy to understand.

Usage of different waiting strategies

The first output is from the promise. The left outputs are from the futures. At first `consumerThread4` asks for the result. 8 characters indent the output. `consumerThread4` also displays its ID. `consumerThread3` immediately follows. 4 characters indent its output. The output of `consumerThread1` and `consumerThread2` is not indented.

- **consumeThread1**: waits unsuccessfully 4000.18 ms seconds without getting the result.

- **consumeThread2**: gets the result after 5000.3 ms, although its waiting duration is up to 20 seconds.

- **consumeThread3**: gets the result after 5601.76 ms. That's about 5600 milliseconds= 8 * 700 milliseconds.

- **consumeThread4**: gets the result after 8193.81 ms. To say it differently. It waits 3 seconds too long.

17. CppMem - An Overview

CppMem[1] is an interactive tool for exploring the behavior of small code snippets using the C++ memory model. It has to be in the toolbox of each programmer who seriously deals with the memory model.

The online version of CppMem - you can also install it on your PC - provides valuable services in a twofold way:

1. CppMem verifies the behavior of small code snippets. Based on the C++ memory model's chosen variant, the tool considers all possible interleavings of threads, visualizes each of them in a graph, and annotates these graphs with additional details.

2. The very accurate analysis of CppMem gives you deep insight into the C++ memory model. In short, CppMem is a tool that helps you to get a better understanding of the memory model.

Of course, it's often the nature of powerful tools that you first have to overcome a few hurdles. The nature of things is that CppMem gives you a very detailed analysis related to this incredibly challenging topic and is highly configurable. Therefore, I plan to present the components of the tool.

17.1 The simplified Overview

My simplified overview of CppMem is based on the default configuration. This overview only provides you with the base for further experiments and should help you understand my ongoing optimization process.

---

[1]http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/
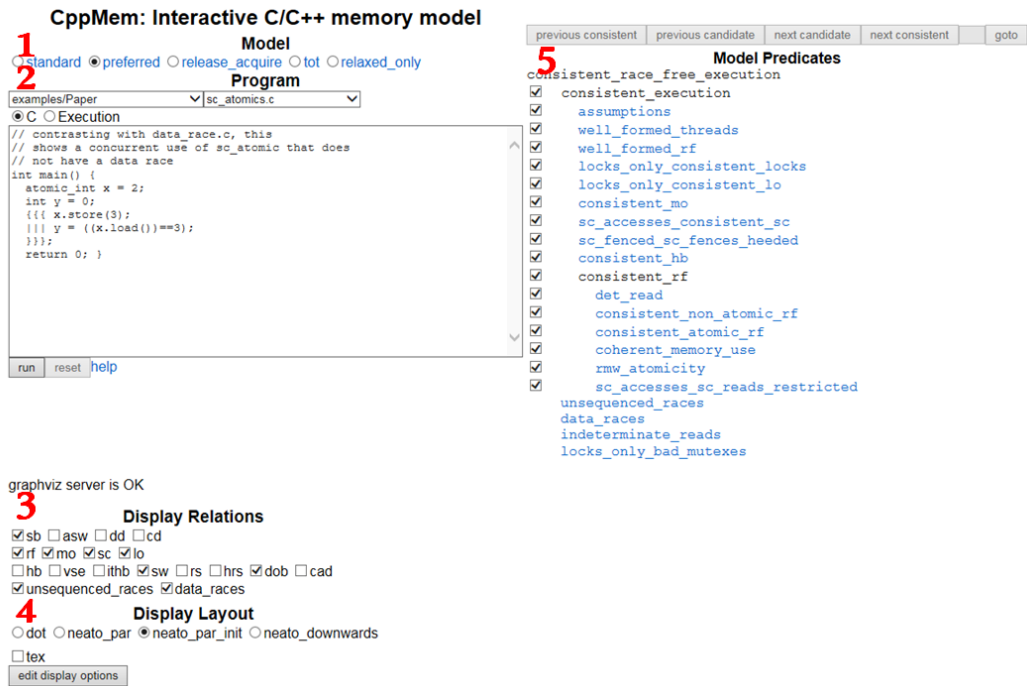
The default configuration of CppMem

For the sake of simplicity, I refer to the red numbers in the screenshot.

17.1.1 1. Model

- Specifies the C++ memory model. *preferred* is a simplified but equivalent variant of the C++11 memory model.

17.1.2 2. Program

- Contains the executable program in a simplified C++11 like syntax. To be precise, you cannot directly copy *C or C++* code programs into CppMem.
- You can choose between many programs that implement typical multithreading scenarios. To get these programs' details read the very well-written article Mathematizing C++ Concurrency[2]. Of course, you can also run your code.
- CppMem is about multithreading; therefore, there are shortcuts for multithreading available.
  - You can easily define two threads using the expression `{{{ ... ||| ... }}}`. The three dots (`...`) represents the work package of each thread.

---

[2]http://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf

- If you use the expression `x.readvalue(1)`, CppMem evaluates the interleavings of the threads for which the thread execution gives the value 1 for `x`.

17.1.3 3. Display Relations

- Describes the relations between the read, write, and read-write modifications on atomic operations, fences and locks.
- You can explicitly enable the relations in the annotated graph with the checkboxes.
- There are three classes of relations. The coarser distinction between original and derived relations is the most interesting one. Here are the default values.
  - Original relations:
    * **sb**: sequenced-before
    * **rf**: read from
    * **mo**: modification order
    * **sc**: sequentially consistent
    * **lo**: lock order
  - Derived relations:
    * **sw**: synchronises-with
    * **dob**: dependency-ordered-before
    * **unsequenced_races**: races in a single thread
    * **data_races**: inter-thread data races

17.1.4 4. Display Layout

- With this switch you can choose which Doxygraph[3] graph is used.

17.1.5 5. Model Predicates

- With this button, you can set the predicates for the chosen model, which can cause a non-consistent (not data-race-free) execution; therefore, if you get a non-consistent execution, you see precisely the reason for the non-consistent execution. I do not use these buttons in this book.

See the documentation[4] for more details.

This is sufficient as a starting point for CppMem. Now, it is time to give CppMem a try.

CppMem provides many examples.

17.1.6 The Examples

The examples show typical use-case when working with concurrent and, in particular, with lock-free code. The examples are grouped into categories.

---

[3]https://sourceforge.net/projects/doxygraph/
[4]http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/help.html

17.1.6.1 Paper

The `examples/Paper` category gives you a few examples which are intensely discussed in the paper Mathematizing C++ Concurrency[5].

- `data_race.c`: data race on x
- `partial_sb.c`: sequenced-before to the evaluation order in a single-threaded program
- `unsequenced_race.c`: unsequenced race on x according to the evaluation order
- `sc_atomics.c`: correct use of atomics
- `thread_create_and_asw.c`: additional synchronize-with due thread creation

Let's start with the first example.

17.1.6.1.1 The Test Run

Choose the program `data_race.c` from the CppMem samples. The run button shows immediately there is a data race.

---

[5]https://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf

A data race with CppMem

For simplicity, I refer to the red numbers in my explanation.

1. The data race is quite easy to see. A thread writes x (x = 3) and another thread reads x (x==3) without synchronization.

2. Two interleavings of threads are possible due to the C++ memory model. Only one of them is consistent with the chosen model. This is the case if, in the expression x==3, the value of x is written by the expression int x = 2 in the main function. The graph displays this relation in the edge annotated with rf and sw.

3. Switching between the different interleaving of threads is fascinating.

4. The graph shows all relations which you enabled in Display Relations.

- a:Wna x=2 is in the graphic the a-th statement, a non-atomic write. Wna stands for "Write non-atomic".

- The graph's key edge is the edge between the writing of x (b:Wna) and the reading of x (C:Rna). That's the data race on x.

17.1.6.2 Further Categories

The other categories focus on specific aspects of lock-free programming. The example of each category is available in various forms. Each form using different memory orderings. For an additional discussion to the categories, read the already mentioned paper Mathematizing C++ Concurrency[6]. If possible, I present the program with sequential consistency.

17.1.6.2.1 Store Buffering (`examples/SB_store_buffering`)

Two threads write to separate locations and then read from the other location.

SB+sc_sc+sc_sc.c

```
// SB+sc_sc+sc_sc
// Store Buffering (or Dekker's), with all four accesses SC atomics
// Question: can the two reads both see 0 in the same execution?
int main() {
  atomic_int x=0; atomic_int y=0;
  {{{ { y.store(1,memory_order_seq_cst);
        r1=x.load(memory_order_seq_cst); }
  ||| { x.store(1,memory_order_seq_cst);
        r2=y.load(memory_order_seq_cst); }  }}}
  return 0;
}
```

17.1.6.2.2 Message Passing (`examples/MP_message_passing`)

One thread writes data (non-atomic) and sets an atomic flag, while the second thread waits for the flag and reads data (non-atomic).

MP+na_sc+sc_na.c

```
// MP+na_sc+sc_na
// Message Passing, of data held in non-atomic x,
//   with sc atomic stores and loads on y  giving release/acquire synchronization
// Question: is the read of x required to see the new data value 1
//   rather than the initial state value 0?
int main() {
  int x=0; atomic_int y=0;
  {{{ { x=1;
        y.store(1,memory_order_seq_cst); }
  ||| { r1=y.load(memory_order_seq_cst).readsvalue(1);
        r2=x; }  }}}
  return 0;
}
```

---

[6]https://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf

17.1.6.2.3 Load Buffering (`examples/LB_load_buffering`)

Can two reads see the later write of the other thread?

Lb+sc_sc+sc_sc.c

```
// LB+sc_sc+sc_sc
// Load Buffering, with all four accesses sequentially consistent atomics
// Question: can the two reads both see 1 in the same execution?
int main() {
  atomic_int x=0; atomic_int y=0;
  {{{ { r1=x.load(memory_order_seq_cst);
        y.store(1,memory_order_seq_cst); }
  ||| { r2=y.load(memory_order_seq_cst);
        x.store(1,memory_order_seq_cst); }  }}}
  return 0;
}
```

17.1.6.2.4 Write-to-Read Causality (`examples/WRC`)

Does the third thread see the write from the first thread? * The first thread writes to `x`. * The second thread reads from that and writes to `y`. * The third thread reads from that and then reads `x`.

WRC+rel+acq_rel+acq_rlx.c

```
// WRC
// the question is whether the final read is required to see 1
// With two release/acquire pairs, it is
int main() {
  atomic_int x = 0;
  atomic_int y = 0;

  {{{ x.store(1,mo_release);
  ||| { r1=x.load(mo_acquire).readsvalue(1);
        y.store(1,mo_release); }
  ||| { r2=y.load(mo_acquire).readsvalue(1);
        r3=x.load(mo_relaxed); }
  }}}
  return 0;
}
```

17.1.6.2.5 Independent Reads of Independent Writes (`examples\IRIW`)

Two threads write to different locations. Can the second thread see those writes in a different order?

IRIW+rel+rel+acq_acq+acq_acq.c

```
// IRIW with release/acquire
// the question is whether the reading threads have
// to see the writes to x and y in the same order.
// With release/acquire, they do not.
int main() {
  atomic_int x = 0; atomic_int y = 0;
  {{{ x.store(1, memory_order_release);
  ||| y.store(1, memory_order_release);
  ||| { r1=x.load(memory_order_acquire).readsvalue(1);
        r2=y.load(memory_order_acquire).readsvalue(0); }
  ||| { r3=y.load(memory_order_acquire).readsvalue(1);
        r4=x.load(memory_order_acquire).readsvalue(0); }
  }}};
  return 0;
}
```

18. Glossary

The idea of this glossary is by no means to be exhaustive but to provide a reference for the essential terms.

18.1 `adress_free`

Atomic operations that are lock-free should also be address-free. Address-free means that atomic operations from different processes on the same memory location are atomic.

18.2 ACID

A transaction is an action that has the properties Atomicity, Consistency, Isolation, and Durability (ACID). Except for durability, all properties hold for transactional memory in C++.

- **Atomicity**: either all or no statement of the block is performed.
- **Consistency**: the system is always in a consistent state. All transactions build a total order.
- **Isolation**: each transaction runs in complete isolation from the other transactions.
- **Durability**: a transaction is recorded.

18.3 CAS

CAS stands for *compare-and_swap* and is an atomic operation. It compares a memory location with a given value and modifies the memory location if the memory location and the given value are the same. The CAS operations in C++ are called `std::compare_exchange_strong`, and `std::compare_-exchange_weak`.

18.4 Callable Unit

A callable unit (short callable) is something that behaves like a function. Not only are these named functions but also function objects or lambda expressions. If a callable accepts one argument, it's called unary callable; if taking two arguments, binary callable.

18.5 Complexity

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 independent of the container's size. Conversely, O(n) means that the runtime depends linear on the number of the container elements.

Predicates are special callables that return a boolean as a result.

18.6 Concepts

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

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

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

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

Requirements such as `ValueSwappable` or `LegacyRandomAccessIterator` are so-called named requirements. Some of these requirements are formalized in C++20 in concepts[1].

18.7 Concurrency

Concurrency means that the execution of several tasks overlaps. Concurrency is a superset of parallelism.

18.8 Critical Section

A critical section is a section of code that at most one thread can use at a time.

---

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

18.9 Deadlock

A deadlock is a state in which at least one thread is blocked forever because it waits for the release of a resource that it will never get.

There are two main reasons for deadlocks:

1. A mutex has not been unlocked.
2. You lock your mutexes in an incorrect order.

18.10 Eager Evaluation

In case of eager evaluation, the expression is evaluated immediately. This evaluation strategy is orthogonal to lazy evaluation. Eager evaluation is often called greedy evaluation.

18.11 Executor

An executor is an object associated with a specific execution context. It provides one or more execution functions for creating execution agents from a callable function object.

18.12 Function Objects

First of all, don't call them functors[2]. That's a *well-defined* term from a branch of mathematics called category theory[3].

Function objects are objects that behave like functions. They achieve this by implementing the function call operator. As function objects are objects, they can have attributes and, therefore, state.

```cpp
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
```

[2]https://en.wikipedia.org/wiki/Functor
[3]https://en.wikipedia.org/wiki/Category_theory

🔑 **Instantiate function objects to use them**

It's a common error that the name of the function object (`Square`) is used in an algorithm instead of the instance of function object (`Square()`) itself: `std::for_each(myVec.begin(), myVec.end(), Square)`. Of course, that's a typical error. You have to use the instance: `std::for_each(myVec.begin(), myVec.end(), Square())`

18.13 Lambda Functions

Lambda functions provide their functionality in-place. The compiler gets its information right on the spot and has therefore excellent optimization potential. Lambda functions can receive their arguments by value or by reference. They can capture the variables of their defining environment by value or by reference as well.

```cpp
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 generally faster than a function or a function object and easier to understand.

18.14 Lazy evaluation

In the case of lazy evaluation[4], the expression is only be evaluated if needed. This evaluation strategy is orthogonal to eager evaluation. Lazy evaluation is often called call-by-need.

18.15 Lock-free

A non-blocking algorithm is lock-free if there is guaranteed system-wide progress.

The following algorithm incrementing the atomic `counter` is lock-free.

---

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

```
std::atomic<int> counter;
int cnt = counter.load();
while (!counter.compare_exchange_strong(cnt, cnt + 1)) {}
```

On thread executing this algorithm succeed and, therefore, there is guaranteed system-wide progress.

A lock-free algorithm is obstruction-free.

18.16 Lock-based

In a lock-based algorithm, at most one thread holding the lock can make progress, but this progress is not guaranteed.

```
std::mutex mut;
int counter;
{
    std::lock_guard<std::mutex> lock(mut);
    ++counter;
}
```

18.17 Lost Wakeup

A lost wakeup is a situation in which a thread misses its wake-up notification due to a race condition. That may happen if you use a condition variable without a predicate.

18.18 Math Laws

A binary operation (*) on some set X is

- **associative** , if it satisfies the associative law for all x, y, z in X: (x * y) * z = x * (y * z)
- **commutative** , if it satisfies the commutative law for all x, y in X: x * y = y * x

18.19 Memory Location

A memory location is according to cppreference.com[5]

- an object of scalar type (arithmetic type, pointer type, enumeration type, or 'std::nullptr_t),
- or the largest contiguous sequence of bit fields of non-zero length.

---

[5]http://en.cppreference.com/w/cpp/language/memory_model

18.20 Memory Model

The memory model defines the relationship between objects and memory location and deals with the question: What happens if two threads access the exact memory locations.

18.21 Modification Order

All modifications to a particular atomic object M occur in some particular total order. This total order is called the modification order of M. Consequently, reads of an atomic object by a particular thread never sees "older" values than those the thread has already observed.

18.22 Monad

Haskell as a pure functional language has only pure functions. A vital feature of these pure functions is that they always return the same result when given the same arguments. Thanks to this property called referential transparency[6], a Haskell function cannot have side effects; therefore, Haskell has a conceptional issue. The world is full of calculations that have side effects. These are calculations that can fail, that can return an unknown number of results, or depend on the environment. To solve this conceptional issue, Haskell uses monads and embeds them in the pure functional language.

The classical monads encapsulate one side effect:

- **I/O monad**: Calculations that deal with input and output.
- **Maybe monad**: Calculations that maybe return a result.
- **Error monad**: Calculations that can fail.
- **List monad**: Calculations that can have an arbitrary number of results.
- **State monad**: Calculations that build a state.
- **Reader monad**: Calculations that read from the environment.

The monad concept is from category theory[7], which is a part of the mathematics that deals with objects and mapping between these objects. Monads are abstract data types (type classes), which transform simple types into enriched types. Values of these enriched type are called monadic values. Once in a monad, a value can only be transformed by a function composition into another monadic value.

This composition respects the unique structure of a monad; therefore, the error monad interrupts its calculation if an error occurs or the state monad builds its state.

A monad consists of three components:

---

[6] https://en.wikipedia.org/wiki/Referential_transparency
[7] https://en.wikipedia.org/wiki/Category_theory

- **Type constructor**: The type constructor defines how the simple data type becomes a monadic data type.
- **Functions**:
  - **Identity function**: Introduces a simple value into the monad.
  - **Bind operator**: Defines how a function is applied to a monadic value to get a new monadic value.
- **Rules for the functions**:
  - The identity function has to be the left and the right identity element.
  - The composition of functions has to be associative.

For the error monad to become an instance of the type class monad, the error monad has to support the bind operator's identity function. Both functions define how the error monad deals with an error in the calculation. If you use an error monad, the error handling is done in the background.

A monad consists of two control flows. The explicit control for calculating the result and the implicit control flow for dealing with the specific side effect.

Of course, you can define monad in fewer words: "A monad is just a monoid in the category of endofunctors."

Monads are becoming more and more import in C++. With C++17 we get std::optional[8], a kind of a Maybe monad. With C++20/23 we will probably get extended futures and the ranges library[9] from Eric Niebler. Both are monads.

18.23 Non-blocking

An algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread. This definition is from the excellent book Java concurrency in practice[10].

18.24 obstruction-free

A non-blocking algorithm is obstruction-free if it the guarantee that a thread can proceed if all other threads are suspended.

18.25 Parallelism

Parallelism means that several tasks are performed at the same time. Parallelism is a subset of Concurrency.

---

[8]http://en.cppreference.com/w/cpp/utility/optional
[9]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4128.html
[10]http://jcip.net/

18.26 Predicate

Predicates are callable units that return a boolean as a result. If a predicate has one argument, it's called a unary predicate. If a predicate has two arguments, it's called a binary predicate.

18.27 Pattern

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution." Christopher Alexander[11]

18.28 RAII

Resource **A**cquisition **I**s **I**nitialization, in short RAII, stands for a popular technique in C++, in which the resource acquisition and release are bound to the lifetime of an object. This means for a lock that the mutex will be locked in the constructor and unlocked in the destructor. This RAII implementation is also known as scoped locking.

Typical use cases in C++ are locks that handle the lifetime of its underlying mutex, smart pointers that handle the lifetime of its resource (memory), or containers of the standard template library[12] that handle the lifetime of its elements.

18.29 Release Sequence

A release sequence headed by a release operation A on an atomic object M is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first operation is A, and every subsequent operation.

- is performed by the same thread that performed A, or
- is an atomic read-modify-write operation.

18.30 Sequential Consistency

Sequential consistency has two essential characteristics:

1. The instructions of a program are executed in source code order.
2. There is a global order of all operations on all threads.

---

[11]https://en.wikipedia.org/wiki/Christopher_Alexander
[12]https://en.cppreference.com/w/cpp/container

18.31 Sequence Point

A sequence point defines any point in the execution of a program at which it is guaranteed that all effects of previous evaluations have been performed, and no effects from subsequent evaluations have yet been performed.

18.32 Spurious Wakeup

A spurious wake-up is an erroneous notification. It may happen that the waiting component of a condition variable or an atomic flag gets a notification, although the notification component didn't send a signal.

18.33 Thread

In computer science, a thread of execution is the smallest sequence of programmed instructions that a scheduler can manage independently, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases, a thread is a process component. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time. For the details, read the Wikipedia article about threads[13].

18.34 Total order

A total order is a binary relation (<=) on some set X which is antisymmetric, transitive, and total.

- **Antisymmetric**: if a <= b and b <= a then a == b
- **Transitivity**: if a <= b and b <= c then a <= c
- **Totality**: a <= b or b <= a

Applied to concurrency, the definition of total order becomes quite handy. Actions such as operation on the same atomic variable or transaction build a total order. This mean, all threads see the effects of these actions in the same order.

---

[13]https://en.wikipedia.org/wiki/Thread_(computing)

18.35 TriviallyCopyable

TriviallyCopyable objects can be copied by copying their object representations manually (`std::memmove`). All data types compatible with the C language (POD types) are trivially copyable.

Formally, TriviallyCopyable is a concept that has to fulfill the following requirements:

- Every copy constructor is trivial or deleted
- Every move constructor is trivial or deleted
- Every copy assignment operator is trivial or deleted
- Every move assignment operator is trivial or deleted
- at least one copy constructor, move constructor, copy assignment operator, or move assignment operator is non-deleted
- Trivial non-deleted destructor

This implies that the class has no virtual functions or virtual base classes. Trivial, in essence, means that none of the specified special member functions is user-defined. This includes also all base classes and non-static class types.

18.36 Undefined Behavior

All bets are off. Your program can produce the correct result, the wrong result, can crash at run time, or may not even compile. That behavior might change when porting to a new platform, upgrading to a new compiler, or as a result of an unrelated code change.

18.37 volatile

`volatile` is typically used to denote objects which can change independently of the regular program flow. For example, these are objects in embedded programming that represent an external device (memory-mapped I/O). Because these objects can change independently of the regular program flow and their value is directly be written into main memory, no optimized storing in caches takes place.

18.38 wait-free

A non-blocking algorithm is wait-free if there is guaranteed per-thread progress.

Each thread incrementing the `counter` in the following program makes progress.

```
std::atomic<int> counter;
counter.fetch_add(1);
```

A wait-free algorithm is lock-free.

Index

Entries in capital letters stand for sections and subsections.