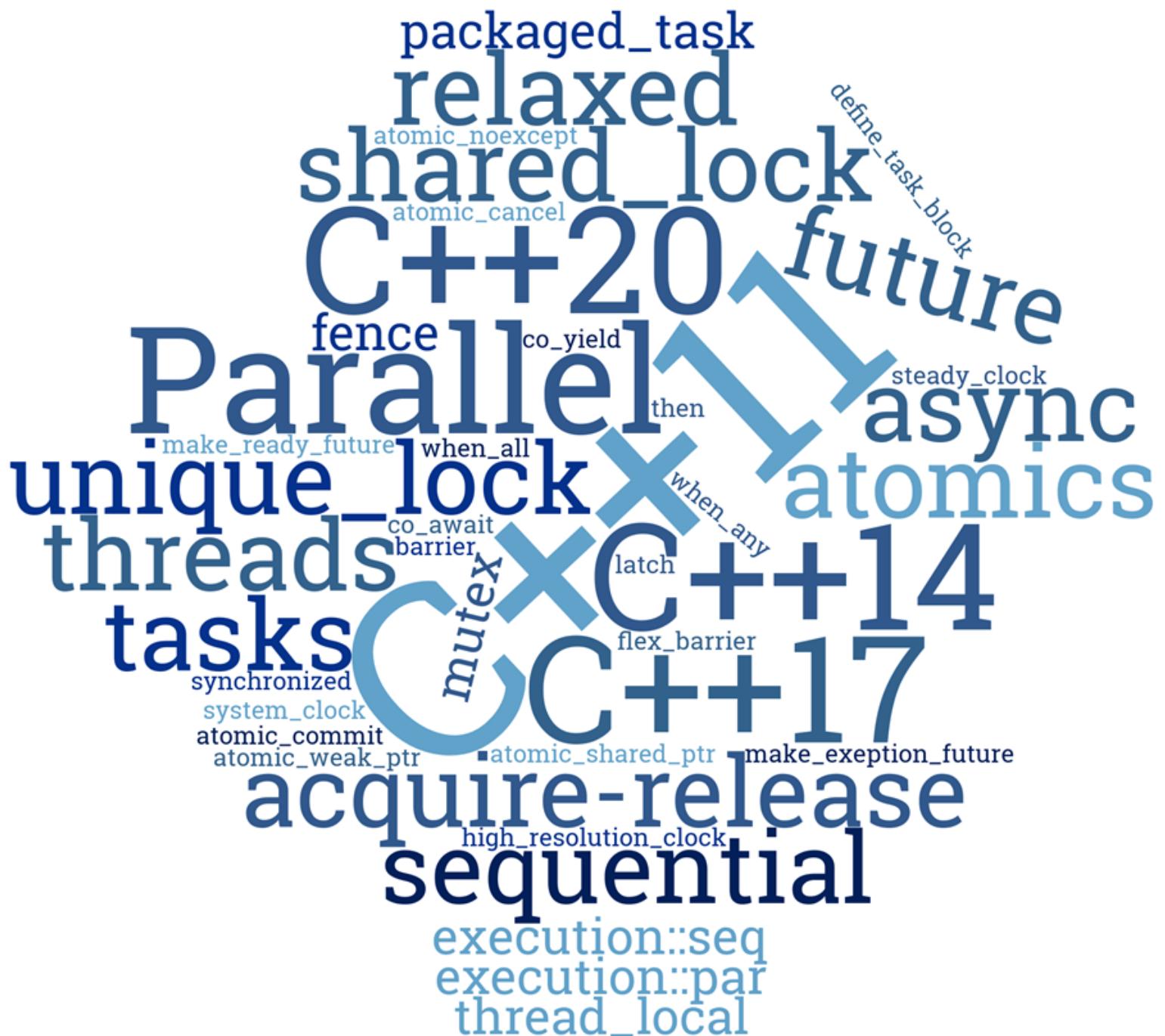


Concurrency with Modern C++

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



Rainer
Grimm

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 2018-11-18



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

© 2017 - 2018 Rainer Grimm

Tweet This Book!

Please help Rainer Grimm by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#rainer_grim](#)mm.

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

[#rainer_grim](#)mm

Contents

Reader Testimonials	i
Introduction	iii
Conventions	iii
Special Fonts	iii
Special Symbols	iii
Special Boxes	iv
Source Code	iv
Run the Programs	iv
How you should read the book?	v
Personal Notes	v
Acknowledgements	v
About Me	vi
My Special Circumstances	vi
A Quick Overview	1
Concurrency with Modern C++	2
C++11 and C++14: The Foundation	2
Memory Model	2
Multithreading	3
C++17: Parallel Algorithms of the Standard Template Library	5
Execution Policy	6
New Algorithms	6
Case Studies	6
Calculating the Sum of a Vector	6
Thread-Safe Initialisation of a Singleton	6
Ongoing Optimisation with CppMem	6
C++20/23: The Concurrent Future	7
Executors	7
std::jthread	7
Atomic Smart Pointers	8
Extended futures	8

CONTENTS

Latches and Barriers	8
Coroutines	8
Transactional Memory	8
Task Blocks	9
Challenges	9
Best Practices	9
Time Library	9
CppMem	9
Glossary	10
The Details	11
Memory Model	12
Basics of the Memory Model	13
What is a memory location?	13
What happens if two threads access the same memory location?	14
The Contract	15
The Foundation	16
The Challenges	16
Atoms	19
Strong versus Weak Memory Model	19
The Atomic Flag	21
The Class Template <code>std::atomic</code>	27
All Atomic Operations	38
Free Atomic Functions	39
<code>std::shared_ptr</code>	40
The Synchronisation and Ordering Constraints	43
The Six Variants of Memory Orderings in C++	43
Sequential Consistency	45
Acquire-Release Semantic	48
<code>std::memory_order_consume</code>	60
Relaxed Semantic	64
Fences	67
<code>std::atomic_thread_fence</code>	67
<code>std::atomic_signal_fence</code>	77
Multithreading	79
Threads	80
Thread Creation	80
Thread Lifetime	81
Thread Arguments	84
Methods	88

CONTENTS

Shared Data	92
Mutexes	94
Locks	101
Thread-safe Initialisation	111
Thread-Local Data	118
Condition Variables	122
The Predicate	124
Lost Wakeup and Spurious Wakeup	126
The Wait Workflow	126
Tasks	128
Tasks versus Threads	128
std::async	130
std::packaged_task	136
std::promise and std::future	141
std::shared_future	147
Exceptions	152
Notifications	155
Parallel Algorithms of the Standard Template Library	158
Execution Policies	159
Algorithms	164
The New Algorithms	164
More overloads	171
The functional Heritage	171
Case Studies	173
Calculating the Sum of a Vector	174
Single Threaded addition of a Vector	174
Multithreaded Summation with a Shared Variable	180
Thread-Local Summation	186
Summation of a Vector: The Conclusion	197
Thread-Safe Initialisation of a Singleton	199
Double-Checked Locking Pattern	199
Performance Measurement	201
Thread-Safe Meyers Singleton	203
std::lock_guard	205
std::call_once with std::once_flag	207
Atomics	208
Performance Numbers of the various Thread-Safe Singleton Implementations	212
Ongoing Optimisation with CppMem	214
CppMem: Non-Atomic Variables	216
CppMem: Locks	221
CppMem: Atomics with Sequential Consistency	222

CONTENTS

CppMem: Atomics with Acquire-Release Semantic	230
CppMem: Atomics with Non-atomics	233
CppMem: Atomics with Relaxed Semantic	235
Conclusion	238
The Future: C++20/23	239
Executors	240
First Examples	241
Goals of an Executor Concept	244
Terminology	244
Execution Functions	245
A Cooperatively Interruptible Joining Thread	247
Automatically Joining	247
Interrupt a <code>std::jthread</code>	249
Atomic Smart Pointers	254
A thread-safe singly linked list	255
Extended Futures	256
Concurrency TS v1	256
Unified Futures	261
Latches and Barriers	267
<code>std::latch</code>	267
<code>std::barrier</code>	268
<code>std::flex_barrier</code>	269
Coroutines	272
A Generator Function	272
Details	275
Transactional Memory	278
ACI(D)	278
Synchronized and Atomic Blocks	279
<code>transaction_safe</code> versus <code>transaction_unsafe</code> Code	282
Task Blocks	283
Fork and Join	283
<code>define_task_block</code> versus <code>define_task_block_restore_thread</code>	284
The Interface	285
The Scheduler	286
Further Information	287
Challenges	288
ABA Problem	288
Blocking Issues	292
Breaking of Program Invariants	294
Data Races	295

CONTENTS

Deadlocks	297
False Sharing	298
Lifetime Issues of Variables	302
Moving Threads	303
Race Conditions	305
Best Practices	307
General	307
Code Reviews	307
Minimise Sharing of mutable data	309
Minimise Waiting	311
Prefer Immutable Data	312
Use pure functions	313
Look for the Right Abstraction	314
Use Static Code Analysis Tools	314
Use Dynamic Enforcement Tools	315
Multithreading	316
Threads	316
Data Sharing	321
Condition Variables	329
Promises and Futures	332
Memory Model	333
Don't use volatile for synchronisation	333
Don't program Lock Free	333
If you program Lock-Free, use well-established patterns	333
Don't build your abstraction, use guarantees of the language	334
Don't reinvent the wheel	334
The Time Library	335
The Interplay of Time Point, Time Duration, and Clock	336
Time Point	337
From Time Point to Calendar Time	337
Cross the valid Time Range	339
Time Duration	341
Calculations	343
Clocks	346
Accuracy and Steadiness	346
Epoch	349
Sleep and Wait	352
CppMem - An Overview	358
The simplified Overview	358
1. Model	359
2. Program	359

CONTENTS

3. Display Relations	360
4. Display Layout	360
5. Model Predicates	360
The Examples	360
Glossary	366
ACID	366
CAS	366
Callable Unit	366
Concurrency	366
Critical Section	367
Executor	367
Function Objects	367
Lambda Functions	368
Lock-free	368
Lost Wakeup	368
Math Laws	368
Memory Location	369
Memory Model	369
Modification Order	369
Monad	369
Non-blocking	370
Parallelism	371
Predicate	371
RAII	371
Release Sequence	371
Sequential Consistency	371
Sequence Point	371
Spurious Wakeup	372
Thread	372
Total order	372
volatile	372
wait-free	372
Index	373

Reader Testimonials

Bart Vandewoestyné



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 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 vectorised.
- 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. Therefore you can combine theory with practice to get the most out of it.

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

Conventions

Only a few conventions.

Special Fonts

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

Bold: I use **Bold** to emphasise 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 special information, tips, and warnings.



Information headline

Information text.



Tip headline

Tip description.



Warning headline

Warning description.

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 both the [GCC](#)¹ and the [clang](#)² compiler, the C++ standard must be specified as well as the threading library to link against. 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

¹<https://gcc.gnu.org/>

²<https://clang.llvm.org/>

The same command-line works 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 a lot of online compilers available. Arne Mertz' blog post [C++ Online Compiler](https://arne-mertz.de/2017/05/online-compilers/)³ gives an excellent overview.

With C++17 and C++20/23, the story becomes quite complicated. I installed the [HPX \(High Performance ParalleX\)](http://stellar.cct.lsu.edu/projects/hpx/)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++20/23](#). Please refer to the corresponding sections in the chapter [The Future: C++20/23](#) and read how you can see the code examples in action.

How you should 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, something that is sometimes quite challenging as it requires a basic understanding of the memory model.

The chapter [The Future: C++20/23](#) is optional. I am very curious about the future. I hope you are too!

The last part, [Further Information](#) provides you additional guidance towards a better understanding of my book and getting the most out of it.

Personal Notes

Acknowledgements

To write this book in English, I started a request in my English blog: www.ModernesCpp.com⁵ and 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, who was the first proofreader.

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

³<https://arne-mertz.de/2017/05/online-compilers/>

⁴<http://stellar.cct.lsu.edu/projects/hpx/>

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

About Me

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

My Special Circumstances

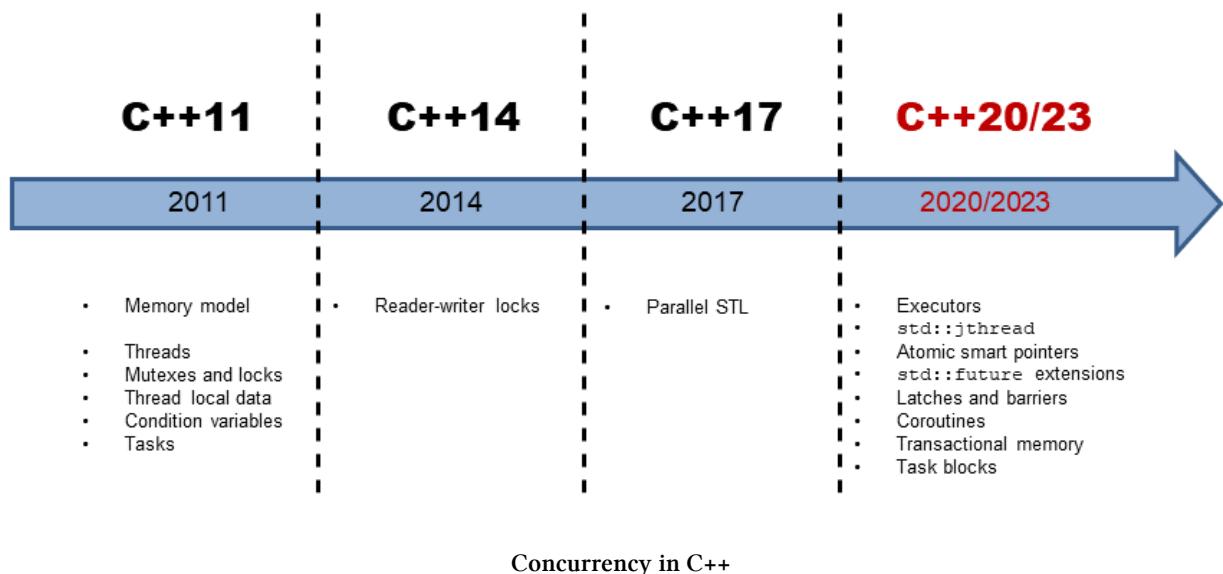
I began Concurrency With Modern C++ in Oberstdorf while getting a new hip joint. Formally, it was a total endoprosthesis of my left hip joint. I wrote the first half of this book during my stay in the clinic and the rehabilitation clinic. To be honest, writing a book helped me a lot during this challenging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

A handwritten signature in blue ink that reads "Rainer Grimm". The signature is fluid and cursive, with the "R" and "G" being particularly prominent.

A Quick Overview

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.



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

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

Multithreading was introduced in C++11. This support consists of two parts: A *well-defined* memory model, and a standardised threading interface. C++14 added reader-writer locks to the multithreading facilities of C++.

Memory Model

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

- Atomic operations: operations that can be 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 was 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 behaviour 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).

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 establish synchronisation and ordering constraints that can also hold for non-atomic types.

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

Multithreading

[Multithreading](#) in C++ consists of threads, synchronisation primitives for shared data, thread-local data and tasks.

Threads

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 thread that is detached from its creator 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.

Shared Data

You have to coordinate access to a shared variable if more than one thread is using it at the same time and the variable is mutable (non-const). Reading and writing a shared variable at the same time is a [data race](#) and therefore undefined behaviour. Coordinating access to a shared variable is achieved with mutexes and locks in C++.

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.

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` 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.

Thread-safe Initialisation 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`.

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 a thread-local data is bound to the lifetime of its thread.

Condition Variables

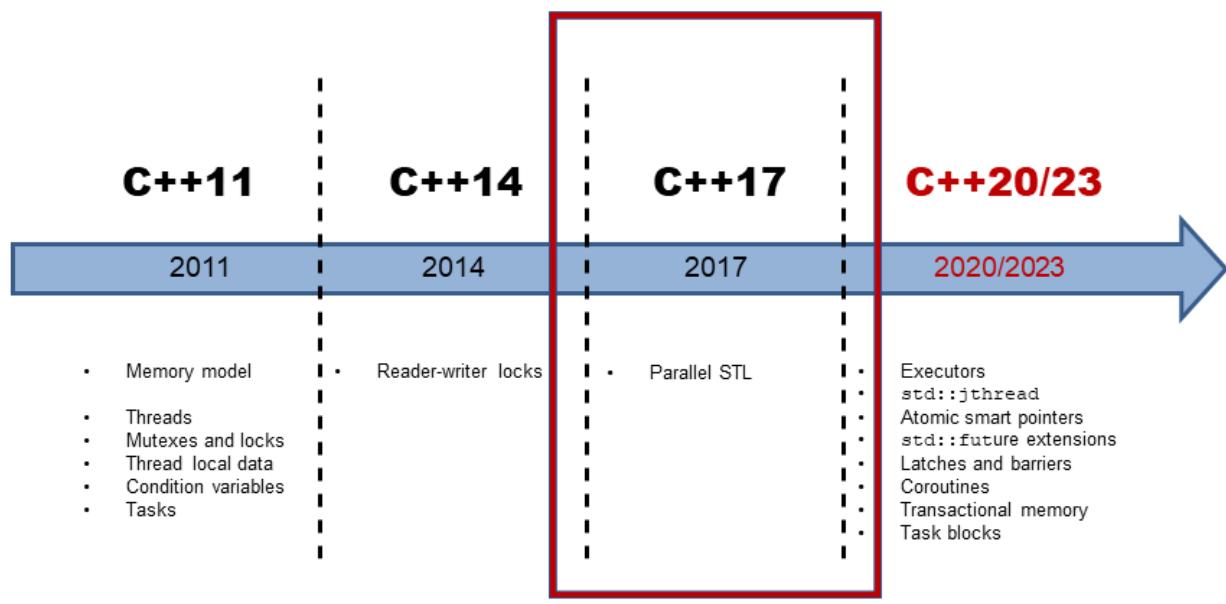
[Condition variables](#) enable threads to be synchronized via messages. One thread acts as the sender while the other one acts as the receiver of the message, where the receiver blocks waiting for the message from the sender. Typical use cases for condition variables are producer-consumer workflows. 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.

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, in the simple case of `std::async`, the lifetime of the task.

Tasks are like data channels between two communication endpoints. They enable thread-safe communication between threads. The promise at 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++17: Parallel Algorithms of the Standard Template Library



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!

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`).

New Algorithms

In addition to the 69 algorithms that are available in overloaded versions for parallel, or parallel and vectorised execution, we get [eight additional algorithms](#). These new ones are well suited for parallel reducing, scanning, or transforming.

Case Studies

After presenting the theory of the memory model and the multithreading interface, I apply the theory in a few [case studies](#).

Calculating the Sum of a Vector

[Calculating the sum of an 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 Initialisation of a Singleton

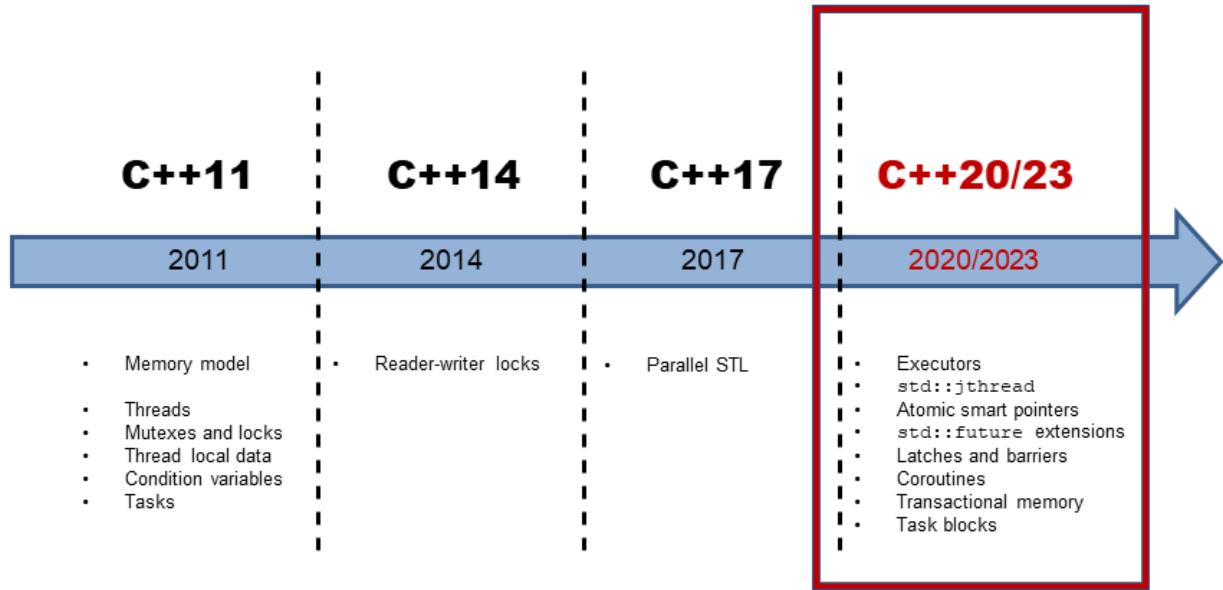
[Thread-safe initialisation of a singleton](#) is the classical use-case for thread-safe initialisation of a shared variable. There are many ways to do it, with varying performance characteristics.

Ongoing Optimisation with CppMem

I start with a small program and successively improve it. I verify each step of my process of [ongoing optimisation with CppMem](#). CppMem⁶ is an interactive tool for exploring the behaviour of small code snippets using the C++ memory model.

⁶<http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/>

C++20/23: The Concurrent Future



Concurrency in C++20/23

It is difficult to make predictions, especially about the future (Niels Bohr⁷). I make statements about the [concurrency features of C++20/23](#).

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 [N4734⁸](#) 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.

`std::jthread`

`std::jthread` is an enhanced replacement for `std::thread`. In addition to `std::thread`, a `std::jthread` can signal an interrupt and can automatically join the started thread.

⁷https://en.wikipedia.org/wiki/Niels_Bohr

⁸<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4734.pdf>

Atomic Smart Pointers

The smart pointer `std::shared_ptr`⁹ and `std::weak_ptr`¹⁰ have a conceptional issue in concurrent programs. They share intrinsically mutable state; therefore, they are prone to data races and thus, leading to undefined behaviour. `std::shared_ptr` and `std::weak_ptr` guarantee that the incrementing or decrementing of the reference counter is an atomic operation and the resource is deleted exactly once, but neither of them can guarantee that the access to its resource is atomic. The new atomic smart pointers `std::atomic<std::shared_ptr<T>>` and `std::atomic<std::weak_ptr<T>>` solve this issue. Both are partial template specialisations of `std::atomic`.

Extended futures

Tasks called promises and futures, introduced in C++11, have a lot to offer, but they also have a drawback: tasks are not composable into powerful workflows. That limitation does not hold for the [extended futures](#) in C++20/23. 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 become ready.

Latches and Barriers

C++14 has no semaphores. Semaphores are used to control access to a limited number of resources. Worry no longer, because C++20/23 proposes [latches and barriers](#). You can use latches and barriers for waiting at a synchronisation point until the counter becomes zero. The difference between latches and barriers is that a `std::latch` can only be used once while a `std::barrier` and `std::flex_barrier` can be used more than once. In contrast to a `std::barrier`, a `std::flex_barrier` can adjust its counter after each iteration.

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.

Transactional Memory

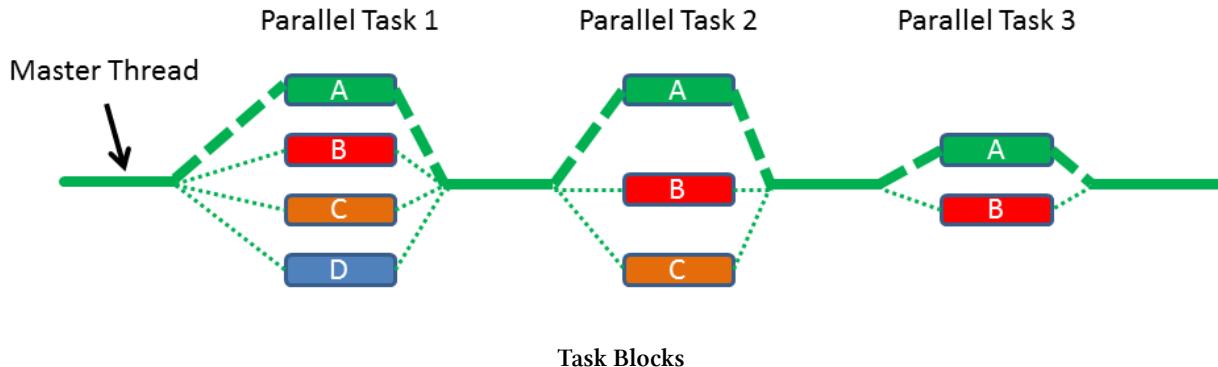
The [transactional memory](#) is based on the ideas underlying transactions in database theory. A transaction is an action that provides the first three properties of ACID database transactions: Atomicity, Consistency, and Isolation. The durability that is characteristic for databases holds not for the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

⁹http://en.cppreference.com/w/cpp/memory/shared_ptr

¹⁰http://en.cppreference.com/w/cpp/memory/weak_ptr

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 synchronise them.



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 contions](#), [data races](#), and [deadlocks](#).

Best Practices

Concurrent programming is inherently complicated, therefore having [best practices](#) for [data sharing](#), the [right abstraction](#), and [static code analysis](#) makes a lot of sense.

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](#).

CppMem

[CppMem](#) is an interactive tool to get deeper inside into the memory model. It provides two very valuable services. First, you can verify your lock-free code and second, you can analyse your lock-free code and get, therefore, a more robust understanding of your code. I often uses 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.

Glossary

The [glossary](#) contains non-exhaustive explanations on the most essential terms.

The Details

Memory Model

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 deeper insight into the multithreading challenges.

However, first of all, what is a memory model?

Basics of the Memory Model

From the concurrency perspective, 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.

What is a memory location?

A memory location is according to [cppreference.com](http://en.cppreference.com)¹¹

- 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 to a memory location:

```
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 a seven sub-objects and the two bit fields `b`, and `c` share the same memory location.

Here are a few import observations:

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

Now, to the crucial part of multithreading.

¹¹http://en.cppreference.com/w/cpp/language/memory_model

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 synchronisation 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 causes the happens-before relations and are, therefore, key fundamental part of the memory model.

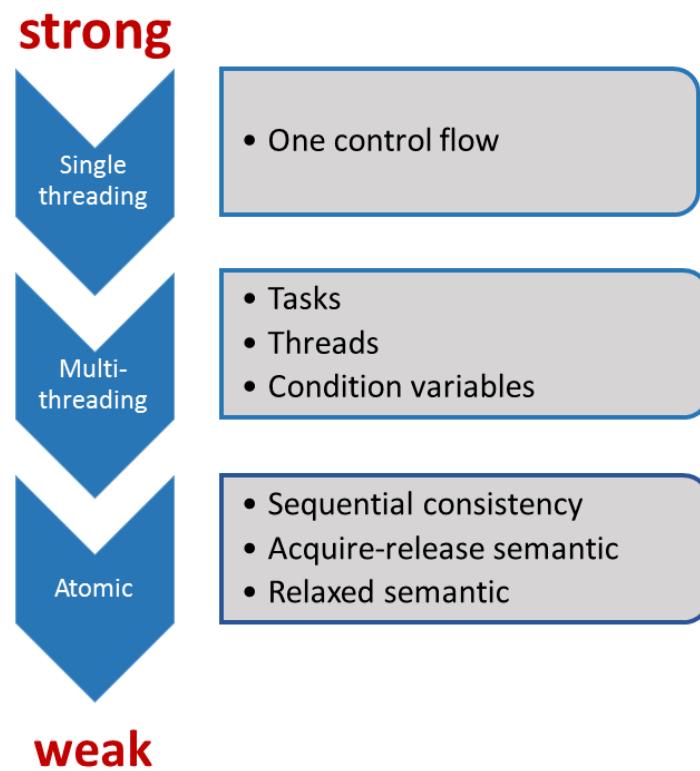
This was my first formal approach to the memory model. Now, I want to give you a mental model for the memory model. The C++ memory model defines a contract.

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 state of the program. Each of the participants wants to optimise its part. For example, the compiler uses registers or modifies loops; the processor performs out of order execution or branch prediction; the caches applies prefetching of instructions or buffering of values. The result is - in the good case - a *well-defined* executable, that is fully optimised 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 optimised executable.

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

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



Three levels of the contract

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

[multithreading](#) or [atomics](#). The system only knew about one control flow, and therefore there were only restricted opportunities to optimise the executable. The key point of the system was to guarantee for the programmer that the observed behaviour 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 arguments is evaluated first, so the behaviour is unspecified. The reason is straightforward - the comma operator is not a sequence point, and this 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* behaviour of threads is the C++ memory model that was heavily inspired by the [Java memory model](#)¹², but 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](#) being called the weak memory model.

The Foundation

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

- Atomic operations: operations that can be performed 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 [synchronisation and order constraints](#) on the program execution. These synchronisation 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 synchronisations and order constraints to your needs.

The Challenges

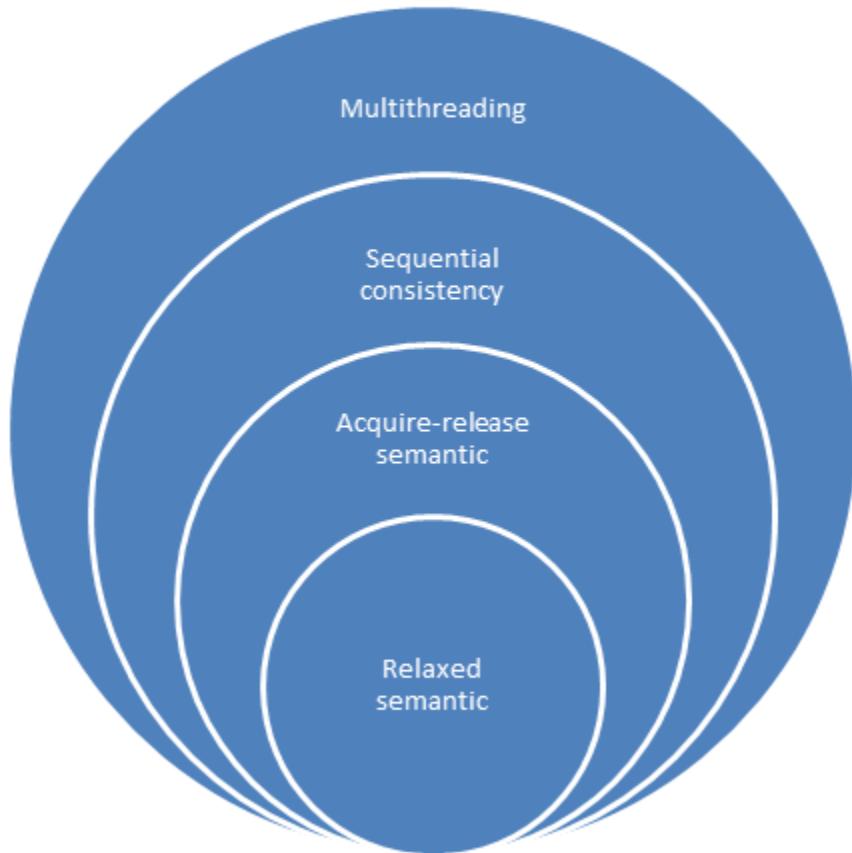
The more we weaken the memory model, the more we change our focus towards other things, such as:

¹²https://en.wikipedia.org/wiki/Java_memory_model

- The program has more optimisation 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-optimisation.

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 end point of our journey.

Let's start with atomics.

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.

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](#).

Strong Memory Model

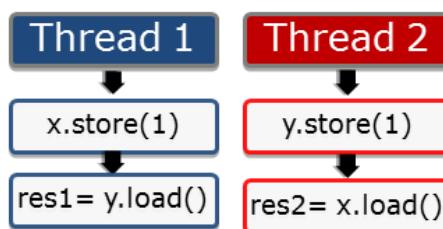
Java 5.0 got its current memory model in 2004, C++ in 2011. Before that, Java had an erroneous memory model, and C++ had no memory model. Those who think this is the endpoint of a long process are entirely wrong. The foundations of multithreaded programming are 40 to 50 years old. [Leslie Lamport¹³](#) defined the concept of sequential consistency in 1979.

Sequential consistency provides two guarantees:

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

Before I dive deeper into these two guarantees, I want to explicitly emphasise that these statements 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

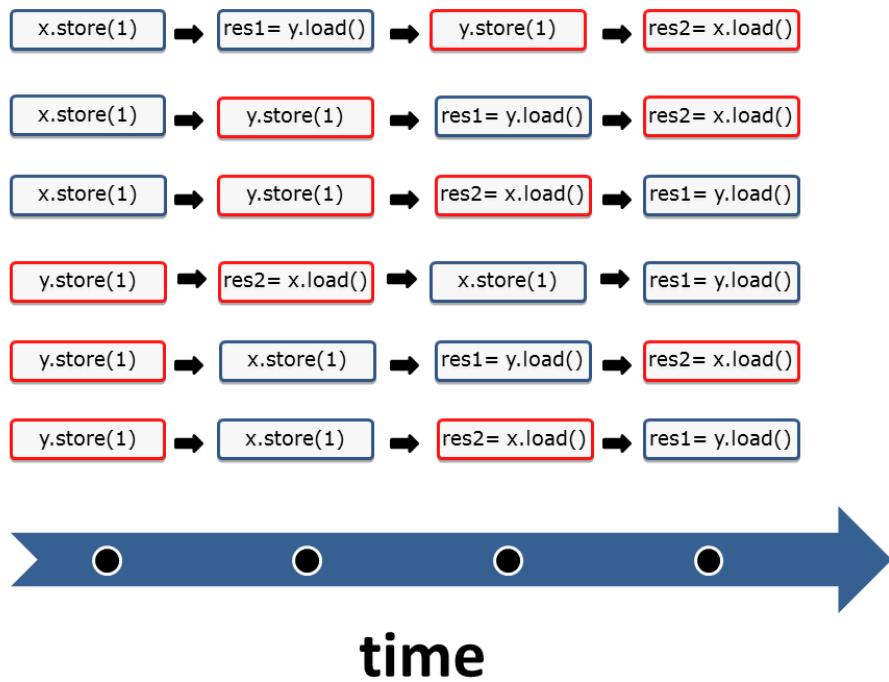
Because the variables are atomic, the operations are executed atomically. By default, sequential consistency applies. The question is: in which order can the statements be executed?

The first guarantee of the 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.

¹³https://en.wikipedia.org/wiki/Leslie_Lamport

The second guarantee of the sequential consistency is that all instructions of all threads 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 which 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.

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 behaviour 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 optimisation possibilities.

With the relaxed semantic - also called weak memory model - there are a lot more combinations of the four operations possible. The *counter-intuitive* behaviour 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 optimisation possibilities. The acquire-release semantic is the key to a deeper understanding of synchronisation and partial ordering in multithreading programming because the threads are synchronised at specific synchronisation points in the code. Without these synchronisation points, there is no *well-defined* behaviour of threads, tasks or condition variables possible.

In the last section, I introduced sequential consistency as the default behaviour 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.

So 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`.

The Atomic Flag

`std::atomic_flag` is a 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` method enables you to set its value to `false`. With the `test_and_set` method you can set the value back to `true` and return the previous value. There is no method to ask for the current value. To use `std::atomic_flag` it must be initialised to `false` with the constant `ATOMIC_FLAG_INIT`.



ATOMIC_FLAG_INIT

The `std::atomic_flag` flag has to be initialised with the statement `std::atomic_flag flag = ATOMIC_FLAG_INIT`. Other initialisation 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 method 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**.



`std::is_always_lock_free`

You can check for each instance of an atomic type `obj` if its lock-free: `obj.is_lock_free()`. This check is performed at runtime. Thanks to `constexpr` function `atomic<type>::is_always_lock_free` you can check for each atomic type if its 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.



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`. It saves the expensive context switch in the wait state from the user space to the kernel space, but it entirely utilises the CPU and wastes CPU cycles. If threads are typically blocked for a short time period, 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 period. If this does not succeed the thread is then be 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 methods `lock` and `unlock`. In addition to this, the `std::atomic_flag` is initialised with class member initialisation 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 initial value of the flag 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 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 value of the flag to `false` so that `t2` must wait until thread `t1` executes the `unlock` method 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 method `test_and_set` of `std::atomic_flag`. The method `test_and_set` consists of two operations: reading and writing. It's critical that both operations are 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 behaviour.

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

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

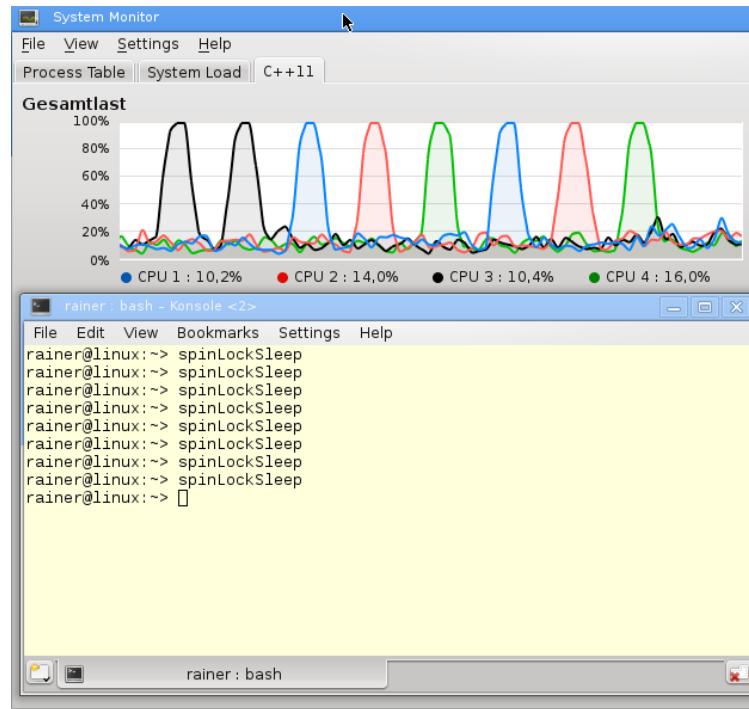
```

1 // spinLockSleep.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     std::this_thread::sleep_for(std::chrono::milliseconds(2000));
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 }
```

If the theory is correct, one of the four cores of my PC is fully utilised. This is precisely what happens. Take a look at the screenshot.



A spinlock which 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

```

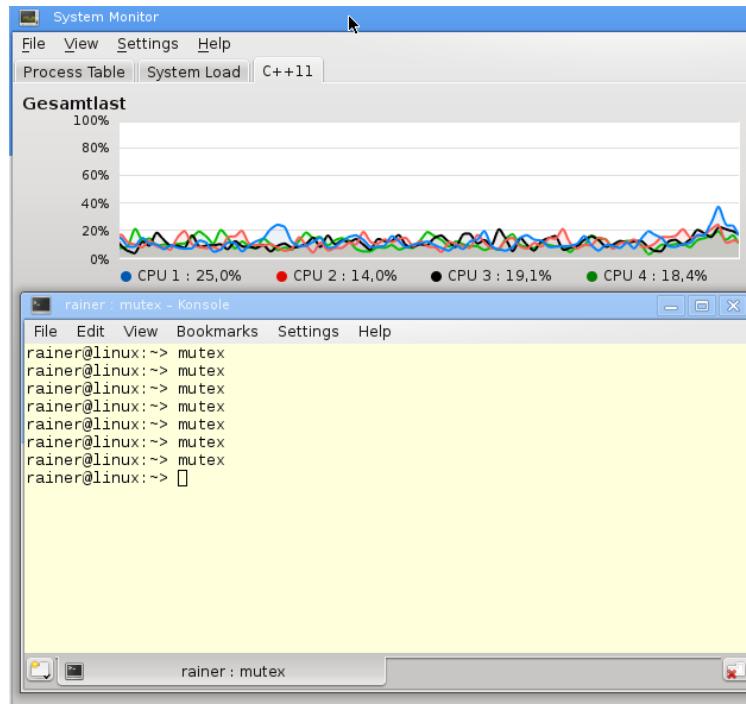
1 // mutex.cpp
2
3 #include <mutex>
4 #include <thread>
5
6 std::mutex mut;
7
8 void workOnResource(){
9     mut.lock();
10    std::this_thread::sleep_for(std::chrono::milliseconds(5000));
11    mut.unlock();
12 }
13
14 int main(){
15
16     std::thread t(workOnResource);
17     std::thread t2(workOnResource);

```

```

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 which sleeps for two seconds

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

The Class Template `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 full specialisations for integral types: `std::atomic<integral type>`.

The atomic booleans and atomic user-defined types support the same interface. The atomic pointer extends the interface of the atomic booleans and atomic integral types. The same applies to the atomic integral types: they extend the interface of the atomic pointers.

The downside of the various variations of `std::atomic` is that you do not have the guarantee that they are **lock-free**.

```
std::atomic<bool>
```

Let's start with `bool: std::atomic<bool>` uses the primary template.

`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 optimised 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 exactly here. The keyword `volatile` in Java and C# has the meaning of `std::atomic` in C++. Alternatively, `volatile` has no multithreading semantic 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 optimised storing in caches takes place.

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

Therefore, let's first use a condition variable.

Usage of a condition variable

```

1 // conditionVariable.cpp
2
3 #include <condition_variable>
4 #include <iostream>
5 #include <thread>
6 #include <vector>
7
8 std::vector<int> mySharedWork;
9 std::mutex mutex_;
10 std::condition_variable condVar;
```

```
11
12 bool dataReady{false};
13
14 void waitingForWork(){
15     std::cout << "Waiting " << std::endl;
16     std::unique_lock<std::mutex> lck(mutex_);
17     condVar.wait(lck, []{ return dataReady; });
18     mySharedWork[1] = 2;
19     std::cout << "Work done " << std::endl;
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" << std::endl;
29     condVar.notify_one();
30 }
31
32 int main(){
33
34     std::cout << std::endl;
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 synchronise on the same mutex `mutex_`. How does the workflow run?

- Thread t1
 - prepares the work package `mySharedWork = {1, 0, 3}`
 - set the non-atomic boolean `dataReady` to `true`
 - send its notification `condVar.notify_one`
- Thread t2
 - 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 t1 sets to `true` and thread t2 checks in the lambda-function `[]{ return dataReady; }` is a kind of memory for the stateless condition variable. Condition variables may be 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 gets to a wait state.

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

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

```

1 // atomicCondition.cpp
2
3 #include <atomic>
4 #include <chrono>
5 #include <iostream>
6 #include <thread>
7 #include <vector>
8
9 std::vector<int> mySharedWork;
10 std::atomic<bool> dataReady(false);
11
12 void waitingForWork(){
13     std::cout << "Waiting " << std::endl;
14     while (!dataReady.load()){
15         std::this_thread::sleep_for(std::chrono::milliseconds(5));
16     }
17     mySharedWork[1] = 2;
18     std::cout << "Work done " << std::endl;
19 }

```

```

21 void setDataReady(){
22     mySharedWork = {1, 0, 3};
23     dataReady = true;
24     std::cout << "Data prepared" << std::endl;
25 }
26
27 int main(){
28
29     std::cout << std::endl;
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](#)¹⁴.

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

¹⁴http://en.cppreference.com/w/cpp/atomic/memory_order

Both programs produce the same result for `mySharedWork`.

Synchronisations of two threads with an atomic and a condition variable



Push versus Pull Principle

I cheated a little. There is one key difference between the synchronisation of the 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 specialisations of `std::atomic` support the bread and butter of all atomic operations: `compare_exchange_strong` and `compare_exchange_weak`.



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 behaviour may vary a little. `atomicValue.compare_exchange_strong(expected, desired)` has the following behaviour.

- 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 method `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 processor doesn't support an atomic compare-exchange instruction. When called in a loop the weak form should be preferred. On some platforms, the weak form can run 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 (1-2) 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.

In addition to booleans, there are 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.

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 a lot of 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 user-defined type, for all base classes of the user-defined type and all non-static members of the user-defined type, must be trivial. This means that you

must not define the copy assignment operator, but you can request it from the compiler by using `default`¹⁵.

- a user-defined type must not have virtual methods or virtual base classes
- a user-defined type must be bitwise comparable so that the C functions `memcpy`¹⁶ or `memcmp`¹⁷ 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](#)¹⁸.

`std::atomic<T*>`

`std::atomic<T*>` is a partial specialisation of the class template `std::atomic`. The atomic pointer `std::atomic<T*>` supports all methods that `std::atomic<bool>` or `std::atomic<user-defined type>` support. It behaves like a plain pointer `T*`. `std::atomic<T*>` supports pointer arithmetic and pre- and post-increment or pre- and post-decrement operations.

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 known integral data types.

`std::atomic<integral type>`

For each integral type there is a full specialisation `std::atomic<integral type>` of `std::atomic`. An `std::atomic<integral type>` supports all operations that `std::atomic<T*>` support, and more. First of all. Which specialisations for integral types exists? Here are the details:

¹⁵<http://en.cppreference.com/w/cpp/keyword/default>

¹⁶<http://en.cppreference.com/w/cpp/string/byte/memcpy>

¹⁷<http://en.cppreference.com/w/cpp/string/byte/memcmp>

¹⁸http://en.cppreference.com/w/cpp/header/type_traits

- character types: `char`, `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>`¹⁹:
 - `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)

`std::atomic<integral type>` supports the composite assignment operators `+=`, `-=`, `&=`, `|=` and `^=` and their fetch pedants: `fetch_add`, `fetch_sub`, `fetch_and`, `fetch_or` and `fetch_xor`. There is a small difference in the composite assignment and the fetch version. The composite assignment operators return the new value; the fetch variations returns the old value. Additionally, the pre- and post-increment and pre- and post-decrement (`++x`, `x++`, `--x`, and `x--`) are available.

A more in-depth look provides more insight: there is no atomic multiplication, atomic division, nor 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`

```

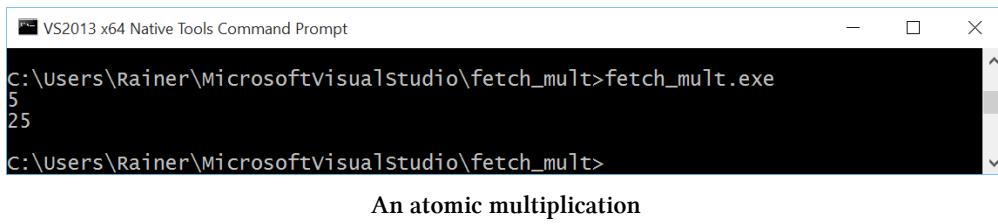
1 // fetch_mult.cpp
2
3 #include <atomic>
4 #include <iostream>
5
6 template <typename T>
7 T fetch_mult(std::atomic<T>& shared, T mult){
8     T oldValue = shared.load();
9     while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
10    return oldValue;

```

¹⁹<http://en.cppreference.com/w/cpp/header/cstdint>

```
11  }
12
13 int main(){
14     std::atomic<int> myInt{5};
15     std::cout << myInt << std::endl;
16     fetch_mult(myInt,5);
17     std::cout << myInt << std::endl;
18 }
```

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 take 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<int> shared` by `mult`. The key observation is that there is a small window of time 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 that there is no per-thread progress guarantee.

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

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
<code>std::atomic_bool</code>	<code>std::atomic<bool></code>
<code>std::atomic_char</code>	<code>std::atomic<char></code>
<code>std::atomic_schar</code>	<code>std::atomic<signed char></code>
<code>std::atomic_uchar</code>	<code>std::atomic<unsigned char></code>
<code>std::atomic_short</code>	<code>std::atomic<short></code>
<code>std::atomic_ushort</code>	<code>std::atomic<unsigned short></code>
<code>std::atomic_int</code>	<code>std::atomic<int></code>
<code>std::atomic_uint</code>	<code>std::atomic<unsigned int></code>
<code>std::atomic_long</code>	<code>std::atomic<long></code>
<code>std::atomic_ulong</code>	<code>std::atomic<unsigned long></code>
<code>std::atomic_llong</code>	<code>std::atomic<long long></code>
<code>std::atomic_ullong</code>	<code>std::atomic<unsigned long long></code>
<code>std::atomic_char16_t</code>	<code>std::atomic<char16_t></code>
<code>std::atomic_char32_t</code>	<code>std::atomic<char32_t></code>
<code>std::atomic_wchar_t</code>	<code>std::atomic<wchar_t></code>
<code>std::atomic_int8_t</code>	<code>std::atomic<std::int8_t></code>
<code>std::atomic_uint8_t</code>	<code>std::atomic<std::uint8_t></code>
<code>std::atomic_int16_t</code>	<code>std::atomic<std::int16_t></code>
<code>std::atomic_uint16_t</code>	<code>std::atomic<std::uint16_t></code>
<code>std::atomic_int32_t</code>	<code>std::atomic<std::int32_t></code>
<code>std::atomic_uint32_t</code>	<code>std::atomic<std::uint32_t></code>
<code>std::atomic_int64_t</code>	<code>std::atomic<std::int64_t></code>
<code>std::atomic_uint64_t</code>	<code>std::atomic<std::uint64_t></code>
<code>std::atomic_int_least8_t</code>	<code>std::atomic<std::int_least8_t></code>
<code>std::atomic_uint_least8_t</code>	<code>std::atomic<std::uint_least8_t></code>
<code>std::atomic_int_least16_t</code>	<code>std::atomic<std::int_least16_t></code>
<code>std::atomic_uint_least16_t</code>	<code>std::atomic<std::uint_least16_t></code>
<code>std::atomic_int_least32_t</code>	<code>std::atomic<std::int_least32_t></code>
<code>std::atomic_uint_least32_t</code>	<code>std::atomic<std::uint_least32_t></code>
<code>std::atomic_int_least64_t</code>	<code>std::atomic<std::int_least64_t></code>
<code>std::atomic_uint_least64_t</code>	<code>std::atomic<std::uint_least64_t></code>
<code>std::atomic_int_fast8_t</code>	<code>std::atomic<std::int_fast8_t></code>
<code>std::atomic_uint_fast8_t</code>	<code>std::atomic<std::uint_fast8_t></code>
<code>std::atomic_int_fast16_t</code>	<code>std::atomic<std::int_fast16_t></code>
<code>std::atomic_uint_fast16_t</code>	<code>std::atomic<std::uint_fast16_t></code>
<code>std::atomic_int_fast32_t</code>	<code>std::atomic<std::int_fast32_t></code>
<code>std::atomic_uint_fast32_t</code>	<code>std::atomic<std::uint_fast32_t></code>
<code>std::atomic_int_fast64_t</code>	<code>std::atomic<std::int_fast64_t></code>
<code>std::atomic_uint_fast64_t</code>	<code>std::atomic<std::uint_fast64_t></code>
<code>std::atomic_intptr_t</code>	<code>std::atomic<std::intptr_t></code>
<code>std::atomic_uintptr_t</code>	<code>std::atomic<std::uintptr_t></code>
<code>std::atomic_size_t</code>	<code>std::atomic<std::size_t></code>
<code>std::atomic_ptrdiff_t</code>	<code>std::atomic<std::ptrdiff_t></code>

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

Type alias	Definition
<code>std::atomic_intmax_t</code>	<code>std::atomic<std::intmax_t></code>
<code>std::atomic_uintmax_t</code>	<code>std::atomic<std::uintmax_t></code>

All Atomic Operations

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

All atomic operations

Method	Description
<code>test_and_set</code>	Atomically set the flag to <code>true</code> and returns the previous value.
<code>clear</code>	Atomically sets the flag to <code>false</code> .
<code>is_lock_free</code>	Checks if the atomic is lock-free.
<code>load</code>	Atomically return the value of the atomic.
<code>store</code>	Atomically replaces the value of the atomic with a non-atomic.
<code>exchange</code>	Atomically replaces the value with the new value. Returns the old value.
<code>compare_exchange_strong</code>	Atomically compares and eventually exchanges the value. Details are here .
<code>compare_exchange_weak</code>	
<code>fetch_add, +=</code>	Atomically adds(subtracts) the value.
<code>fetch_sub, -=</code>	
<code>fetch_or, =</code>	Atomically performs bitwise (OR, AND, and XOR) operation with the value.
<code>fetch_and, &=</code>	
<code>fetch_xor, ^=</code>	
<code>++, --</code>	Increments or decrements (pre- and post-increment) the atomic.

The atomic types have no copy constructor or copy assignment operator but they support assignment from and implicit conversion to the underlying built-in type. The composite assignment operators return the new value; the fetch variations return 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(2011j);
atomObj = 2014;
long long nonAtomObj = atomObj;
```

Each method 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` method can be parametrised 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 all atomic types. The table shows the list of all atomic operations depending on the atomic type.

All atomic operations depending on the atomic type

Method	atomic_flag	atomic<bool>	atomic<user>	atomic<T*>	atomic<integral>
test_and_set	yes				
clear	yes				
is_lock_free		yes	yes	yes	yes
load		yes	yes	yes	yes
store		yes	yes	yes	yes
exchange		yes	yes	yes	yes
compare_exchange_strong		yes	yes	yes	yes
compare_exchange_weak					
fetch_add, +=				yes	yes
fetch_sub, -=					
fetch_or, =					yes
fetch_and, &=					
fetch_xor, ^=					
++, --				yes	yes

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 `_explicit` variations expect a **memory-ordering**.

For each `std::atomic` type their 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 method call `at.store()` on a `std::atomic` becomes `std::atomic_store()`, and `std::atomic_store_except()`. The first overload expects in this case a pointer and the second overload an memory-ordering.

For completeness, here are all overloads: `atomic`²⁰.

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

`std::shared_ptr`

`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 instances of smart pointers should provide a minimum atomicity guarantee in multithreading programs. What is the meaning of the minimal atomicity guarantee 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 assertion that a `std::shared_ptr` provides, are described by `Boost`²¹.

1. A `shared_ptr` instance can be “read” (accessed using only `const` 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.

²⁰<http://en.cppreference.com/w/cpp/atomic>

²¹http://www.boost.org/doc/libs/1_57_0/libs/shared_ptr/shared_ptr.htm#ThreadSafety

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         localPtr = 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 }
```

The lambda-function 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 behaviour.

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.

Atomic Operations on `std::shared_ptr`

There are specialisations 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](http://en.cppreference.com)²². Now it is quite easy to modify a shared pointer that is 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 `auto localPtr = std::make_shared<int>(2014)` is thread-safe. All is well? NO! Finally, we need atomic smart pointers.



Atomic Smart Pointers

That is not the end of the story for atomic smart pointers. With C++20 we can expect with high probability two new smart pointers: `std::atomic<std::shared_ptr>` and `std::atomic<std::weak_ptr>`. For the impatient reader here are the details of the upcoming [atomic smart pointers](#).

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

²²http://en.cppreference.com/w/cpp/memory/shared_ptr

The Synchronisation and Ordering Constraints

You cannot configure the atomicity of an atomic data type, but you can accurately adjust the synchronisation and ordering constraints of atomic operations. This is a possibility which 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 key question is what their characteristics are?

The Six Variants of Memory Orderings in C++

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

The memory orderings

```
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 synchronisation and ordering constraints are defined by the six variants?

My plan is quite simple: I answer both questions.

Kind of Atomic Operation

There are three different kinds of operations:

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

`memory_order_relaxed` defines no synchronisation 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		
store		yes	
exchange			yes
compare_exchange_strong			yes
compare_exchange_weak			
fetch_add, +=			yes
fetch_sub, -=			
fetch_or, =			yes
fetch_and, &=			
fetch_xor, ^=			
++, --			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); operation atomVar.load(std::memory_order_release)` is equivalent to `atomVar.load(std::memory_order_relaxed)`.

Different Synchronisation and Ordering Constraints

There are, roughly speaking, three different types of synchronisation 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, reads of an atomic object by a particular thread 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 interesting, but also challenging, topic. Let us discuss the synchronisation and ordering constraints of the sequential consistency, the acquire-release semantic, and the relaxed semantic.

Sequential Consistency

Let us dive deeper into sequential consistency. The key for 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 synchronise threads.

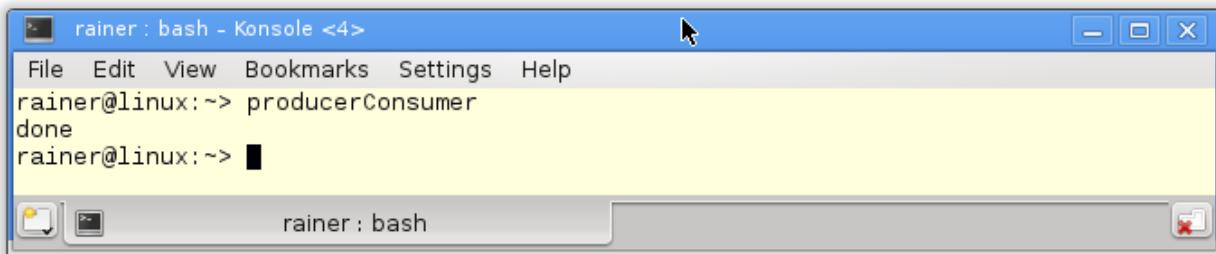
The following program synchronises the producer and the consumer thread with the help of sequential consistency.

Producer-Consumer synchronisation with sequential consistency

```
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 << std::endl;
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 }
```

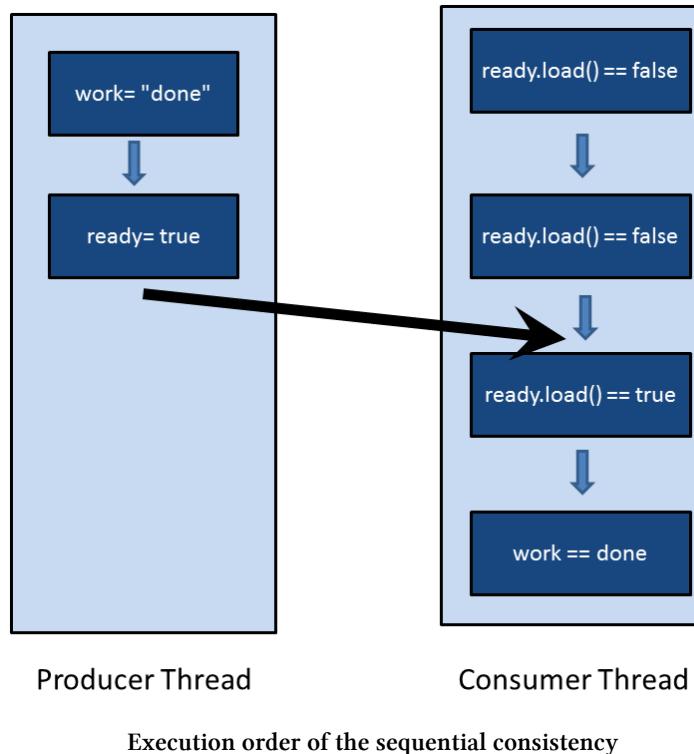
The output of the program is not very exciting.



Producer-Consumer synchronisation 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 continues its work.



It is quite easy to understand that the program always return “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 operations of the other thread in the same order. Both threads follow the same universal clock. This synchronisation does also hold - with the help of the `while(!ready.load()){}` loop - for the synchronisation 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`
 \Rightarrow `work = "done"` *happens-before* `ready = true`
2. `while(!ready.load()){}` is *sequenced-before* `std::cout << work << std::endl`
 \Rightarrow `while(!ready.load()){}` *happens-before* `std::cout << work << std::endl`
3. `ready = true` *synchronizes-with* `while(!ready.load()){}`
 \Rightarrow `ready = true` *inter-thread happens-before* `while (!ready.load()){}`
 \Rightarrow `ready = true` *happens-before* `while (!ready.load()){}`

The final 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 << std::endl`

In sequential consistency, a thread sees the operations of another thread 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. This is an area where C# and Java does not follow. That's also an area where our intuition begins to wane.

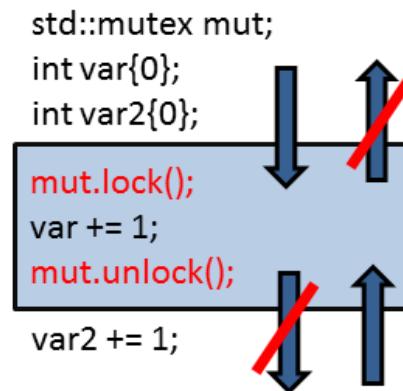
Acquire-Release Semantic

There is no global synchronisation between threads in the acquire-release semantic; there is only synchronisation between atomic operations **on the same atomic variable**. A write operation on one thread synchronises 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 synchronises with an acquire operation on the same atomic and establishes an ordering constraint. This means all read and write operations cannot be moved after a release operation, and all read and write operations cannot be moved before an acquire operation.

What is an acquire or release operation? The reading of an atomic variable with `load` or `test_and_set` is an acquire operation. There is more: There is more: the releasing of 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` on 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. In particular the acquire-release semantic helps you to get a better understanding of the high-level synchronisation 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 synchronisation 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 synchronises 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 behaviour 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).

Transitivity

A release operation synchronises with an acquire operation on the same atomic variable and, additionally, establishes ordering constraint. These are the components to synchronise 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 synchronise threads that are independent.

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

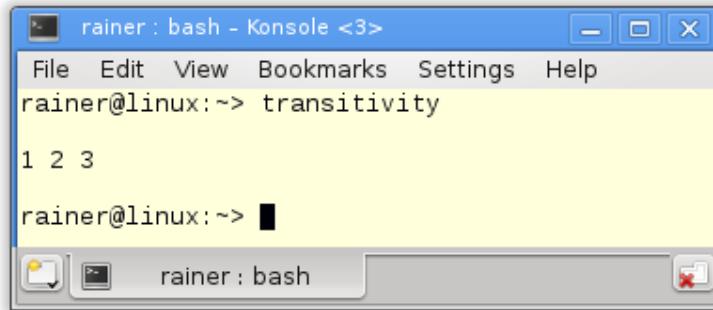
Transitivity of the acquire-release semantics

```
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 << std::endl;
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.



```

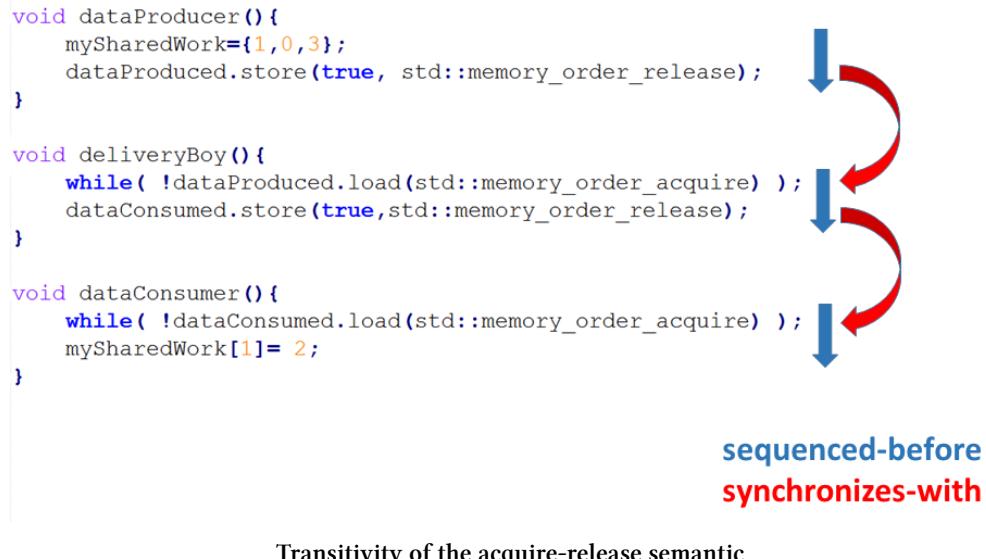
rainer : bash - Konsole <3>
File Edit View Bookmarks Settings Help
rainer@linux:~> transitivity
1 2 3
rainer@linux:~> 
```

Output of the program `transitivity.cpp`

There are two important 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.



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 synchronisation 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 corresponds to the direction of the arrows from top to bottom. Finally, we have the 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 synchronise threads, if The typical misunderstanding is about the if.

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.

Waiting Included

Here is a simple program as a starting point.

Acquire-release with waiting

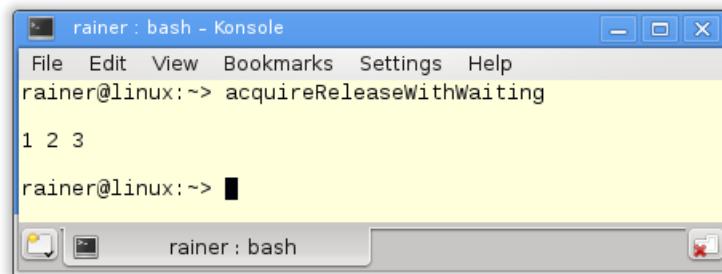
```

1 // acquireReleaseWithWaiting.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     while( !dataProduced.load(std::memory_order_acquire) );
18     mySharedWork[1] = 2;
19 }
```

```

20
21 int main(){
22
23     std::cout << std::endl;
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 }
```

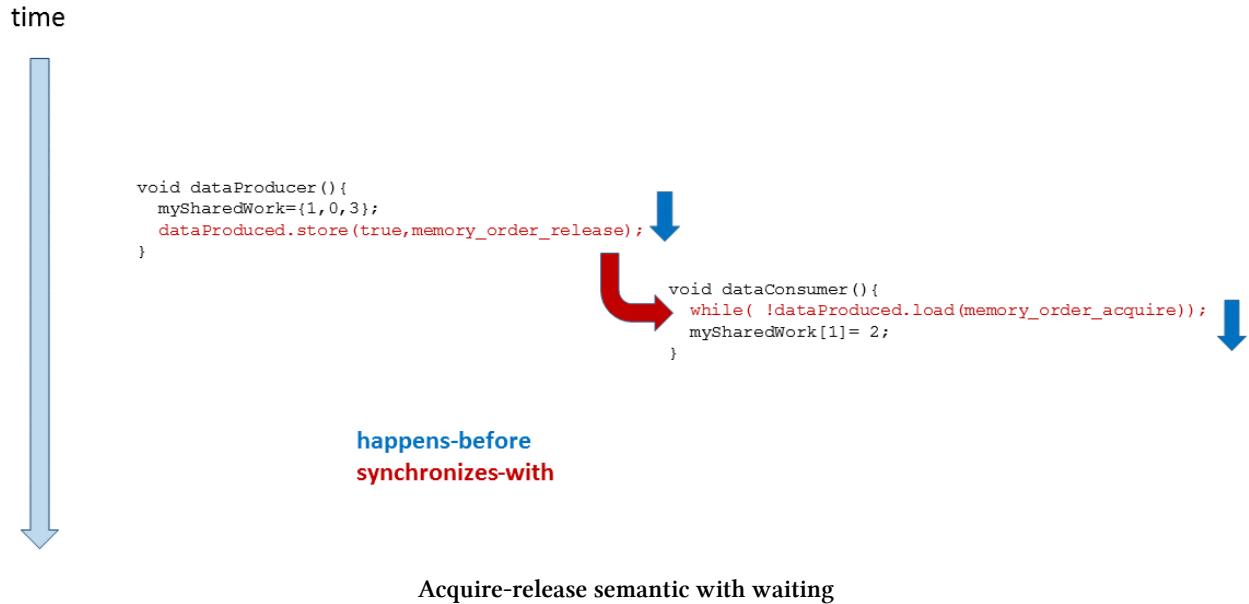
The consumer thread `t1` in line 17 waits until the consumer thread `t2` in line 13 sets `dataProduced` to `true`. `dataProduced` is the guard and it guarantees that access to the non-atomic variable `mySharedWork` is synchronised. This means that the producer thread `t2` initialises `mySharedWork`, then the consumer thread `t2` 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`.



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

If ...

What happens if the consumer thread t_1 in line 17 doesn't wait for the producer thread t_2 ?

Acquire-release without waiting

```

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 }
```

```

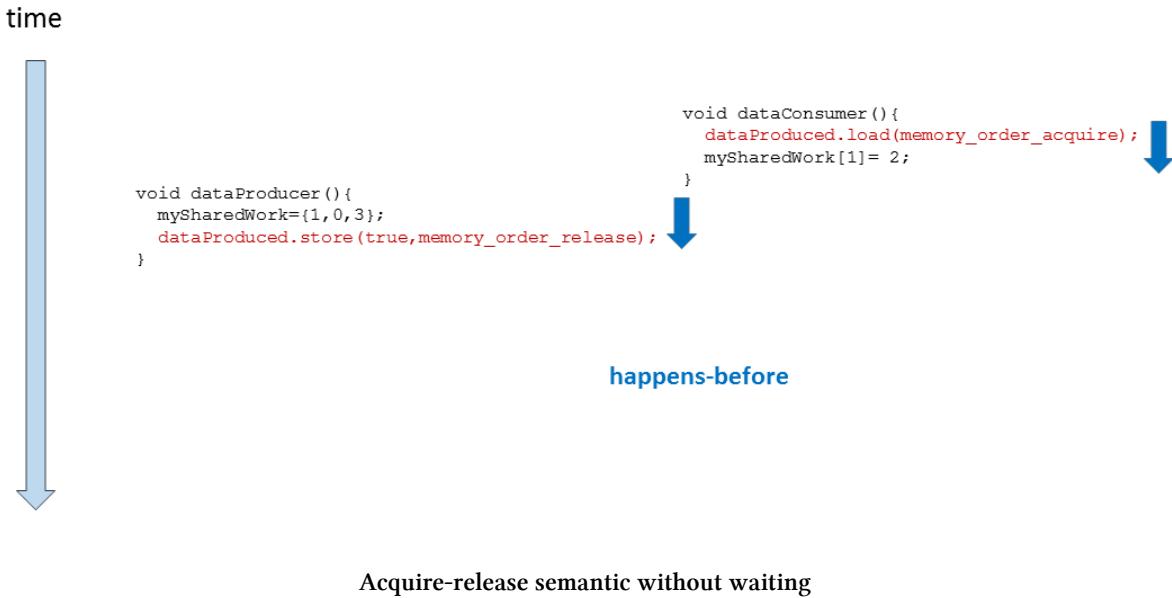
20
21 int main(){
22
23     std::cout << std::endl;
24
25     std::thread t1(dataConsumer);
26     std::thread t2(dataProducer);
27
28     t1.join();
29     t2.join();
30
31     for (auto v: mySharedWork){
32         std::cout << v << " ";
33     }
34
35     std::cout << "\n\n";
36
37 }
```

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

Undefined behaviour 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* relationship.

with relation.



Acquire-release semantic without waiting

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))`).

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)`.

Release Sequence

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

Acquire-release without waiting

```
1 // releaseSequence.cpp
2
3 #include <atomic>
4 #include <thread>
5 #include <iostream>
6 #include <mutex>
7
8 std::atomic<int> atom{0};
9 int somethingShared{0};
10
11 using namespace std::chrono_literals;
12
13 void writeShared(){
14     somethingShared = 2011;
15     atom.store(2, std::memory_order_release);
16 }
17
18 void readShared(){
19     while ( !(atom.fetch_sub(1, std::memory_order_acquire) > 0) ){
20         std::this_thread::sleep_for(100ms);
21     }
22
23     std::cout << "somethingShared: " << somethingShared << std::endl;
24 }
25
26 int main(){
27
28     std::cout << std::endl;
29
30     std::thread t1(writeShared);
31     std::thread t2(readShared);
32     // std::thread t3(readShared);
33
34     t1.join();
35     t2.join();
36     // t3.join();
37
38     std::cout << "atom: " << atom << std::endl;
39
40     std::cout << std::endl;
41
42 }
```

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. This 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.

```
rainer@linux:~> releaseSequence
somethingShared: 2011
somethingShared: 2011
atom: 1998
rainer@linux:~>
```

A release sequence

More formally the definition of a release sequence by the [N4659: Working Draft, Standard for Programming Language C++²³](#).



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.

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`.

²³<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf>

std::memory_order_consume

`std::memory_order_consume` is the most legendary of the six memory orderings. That is 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 is acceptable because both are load or acquire operations. `std::memory_order_consume` requires weaker synchronisation 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 get an understanding of 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 emphasise the strong relationship of `std::memory_order_consume` and `std::memory_order_acquire`.

Release-acquire ordering

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

Release-acquire ordering

```

1 // acquireRelease.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_acquire)));
24     cout << "*p2: " << *p2 << endl;
25     cout << "data: " << data << endl;
26     cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
27 }
28
29 int main(){
30
31     cout << endl;
32
33     thread t1(producer);
34     thread t2(consumer);
35
36     t1.join();
37     t2.join();
38
39     cout << endl;
40
41 }
```

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

Release-consume ordering

I replace the memory order `std::memory_order_acquire` in line 21 with `std::memory_order_consume`.

Release-acquire 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 << endl;
25     cout << "data: " << data << endl;
26     cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
27 }
28
29 int main(){
30
31     cout << endl;
32
33     thread t1(producer);
34     thread t2(consumer);
35
36     t1.join();
37     t2.join();
38
39     cout << endl;
40
41 }
```

Now the program has undefined behaviour. 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.

Release-acquire versus Release-consume ordering

The outputs of the programs are identical.

```
rainer@linux:~> acquireRelease
*p2: C++11
data: 2011
atoData: 2014
rainer@linux:~> acquireConsume
*p2: C++11
data: 2011
atoData: 2014
rainer@linux:~>
```

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 16 *synchronizes-with* the load operation in line 21. The reason is that the store operation uses `std::memory_order_release` and the load operation uses `std::memory_order_acquire`. This is the synchronisation. What are the ordering constraints for the release-acquire operations? The release-acquire ordering guarantees that the results of all operations before the store operation (line 16) are available after the load operation (line 21). So also, the release-acquire operation orders the 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`?

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 visualise them.

```

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

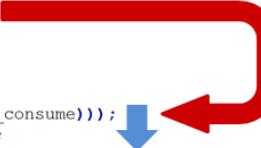
void producer(){
    std::string* p = new std::string("C++11");
    data = 2011;
    atoData.store(2014, std::memory_order_relaxed);
    ptr.store(p, std::memory_order_release);
}

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 << std::endl` 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 << std::endl`, 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 race condition 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 behaviour.

Finally, we have reached the relaxed semantic.

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 occur in some particular total order.

No synchronisation and ordering constraints

This is quite easy. If there are no rules, we cannot violate them. However, that is too easy; the program should have *well-defined* behaviour. This means in particular that you typically use synchronisation 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 synchronised.

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 in which order the different threads increment the counter; the critical observation for a counter is that all increments are atomic and that all threads' tasks are done at the end. Have a look at the following example.

A counter with relaxed semantic

```

1 // relaxed.cpp
2
3 #include <vector>
4 #include <iostream>
5 #include <thread>
6 #include <atomic>
7
8 std::atomic<int> count = {0};
9
10 void add()
11 {
12     for (int n = 0; n < 1000; ++n) {
13         count.fetch_add(1, std::memory_order_relaxed);
14     }
15 }
16
17 int main()
18 {
19     std::vector<std::thread> v;
20     for (int n = 0; n < 10; ++n) {
21         v.emplace_back(add);
22     }
23     for (auto& t : v) {
24         t.join();
25     }
26     std::cout << "Final counter value is " << count << '\n';
27 }
```

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

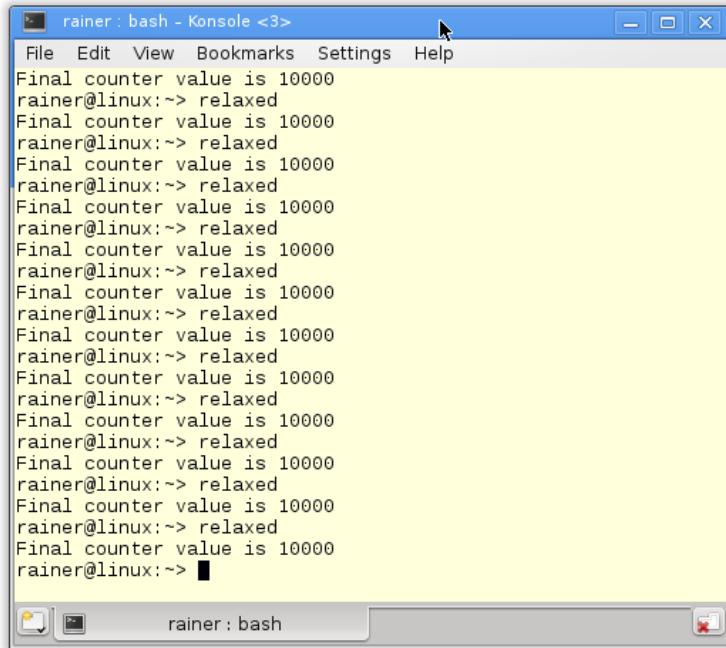
In line 13 the atomic number count is incremented using the relaxed semantic, so we have a guarantee that the operation is atomic. The `fetch_add` operation establishes an ordering on count. The function `add` (lines 10 - 15) is the work package of the threads. Each thread gets its work package on line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

Thread creation is one synchronisation point. The other one being `t.join()` on line 24.

The creator thread synchronises with all its children in line 24. It waits with the `t.join()` call until all its children are done. `t.join()` is the reason that 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`. This only holds for the increment operation. The key property for incrementing the reference counter is that the operation is atomic. The order of the increment operations does not matter. This 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 synchronisation 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 synchronisation and ordering constraints between threads without an atomic operation.

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`: synchronises between a signal handler and code running on the same thread.

`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 what a `std::atomic_thread_fence` is all about.

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.

The Three Fences

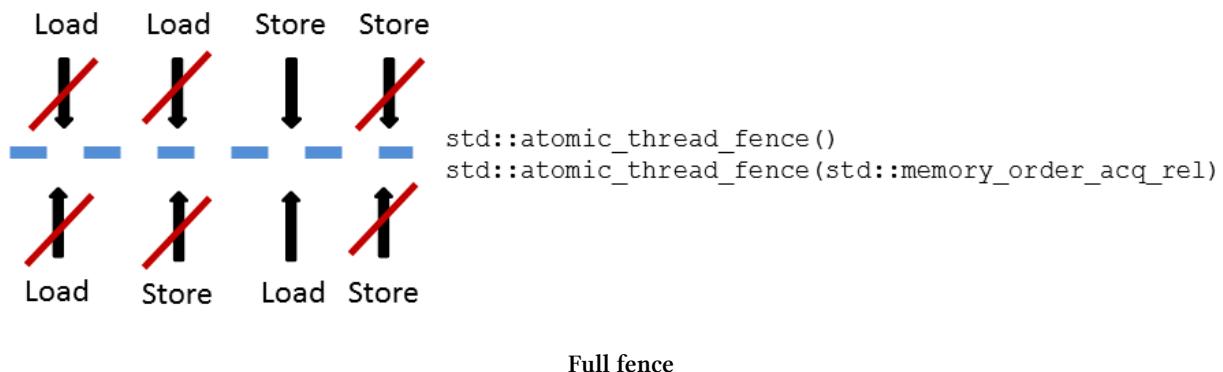
Typically, three kinds of fences are used. They are called full fence, acquire fence and release fence. As a reminder, acquire is a load, and release is a store operation. What happens if I place one of the three memory barriers between the four combinations of load and store operations?

- **Full fence:** A full fence `std::atomic_thread_fence()` between two arbitrary operations prevents the reordering of these operations, but that guarantee does not hold for StoreLoad operations. They 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 reordered with a read or write operation before a release fence.

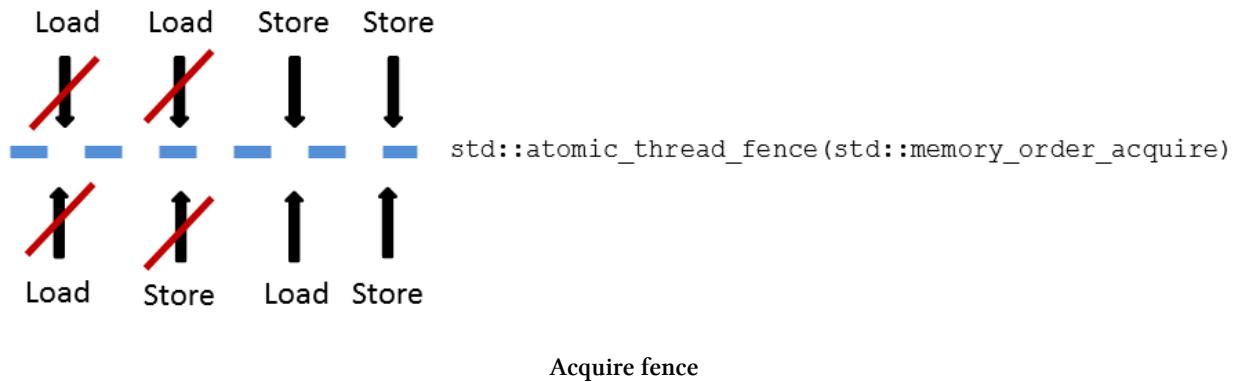
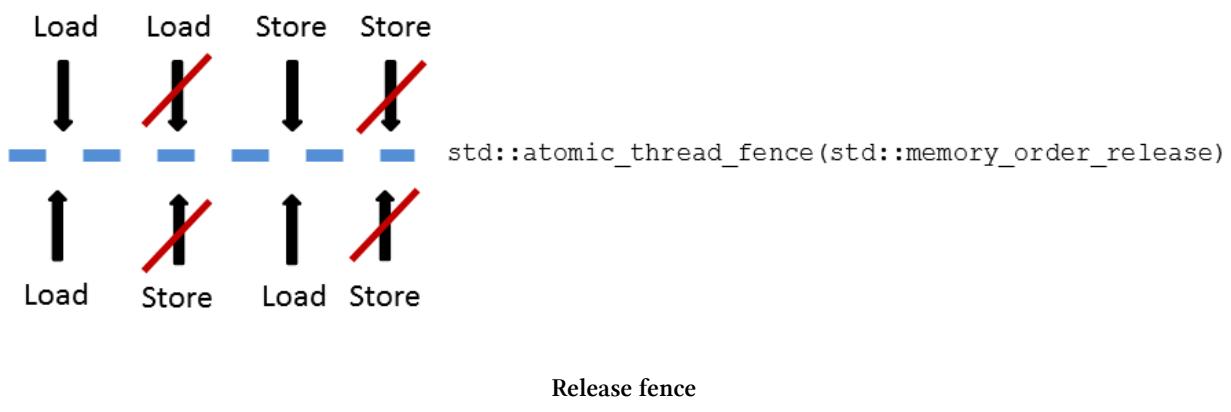
A lot of energy goes into getting the definitions of the acquire and release fence and their consequences for lock-free programming right. Especially challenging to understand are the subtle differences between the acquire-release semantic of atomic operations. Before I get to that point, I illustrate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

Full fence

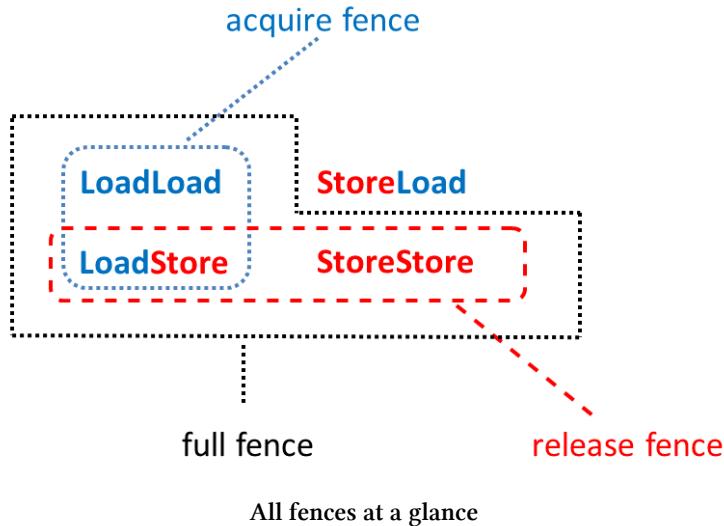


Of course, instead of writing `std::atomic_thread_fence()` you can explicitly write `std::atomic_thread_fence(std::memory_order_seq_cst)`. **Sequential consistency** is applied to fences by default. If you use sequential consistency for a full fence, the `std::atomic_thread_fence` follows a global order.

Acquire fence**Release fence**

The three memory barriers can be depicted even more concisely.

All Fences at a Glance



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

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.

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 synchronisation and ordering constraints between threads. These synchronisation 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.

Acquire Operation

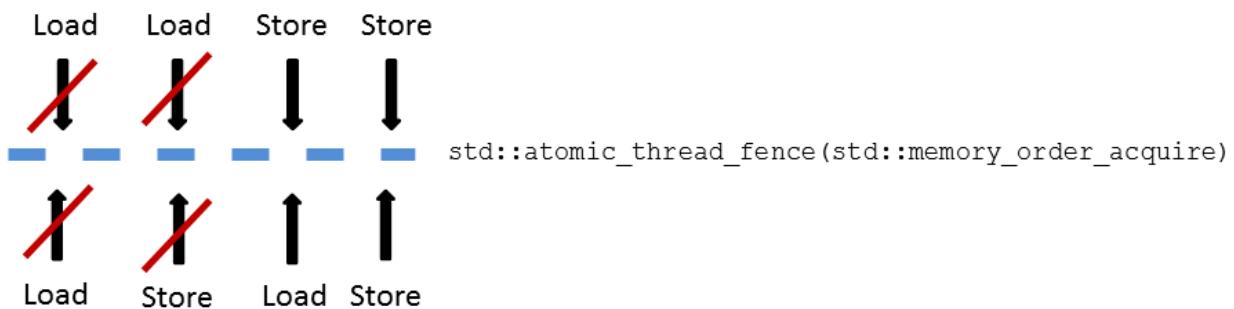
A load (read) operation on an atomic variable with the memory-ordering set to `std::memory_order_acquire` is an acquire operation.

```
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:



Fence with acquire semantic

This comparison emphasises two points:

1. A fence with acquire semantic establishes stronger ordering constraints. Although the acquire operation on an atomic and on a fence require that no read or write operation can 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::atomic_thread_fence(std::memory_order_acquire)`, this operation cannot be moved after the acquire fence.

Similar observations can be made for the release fence.

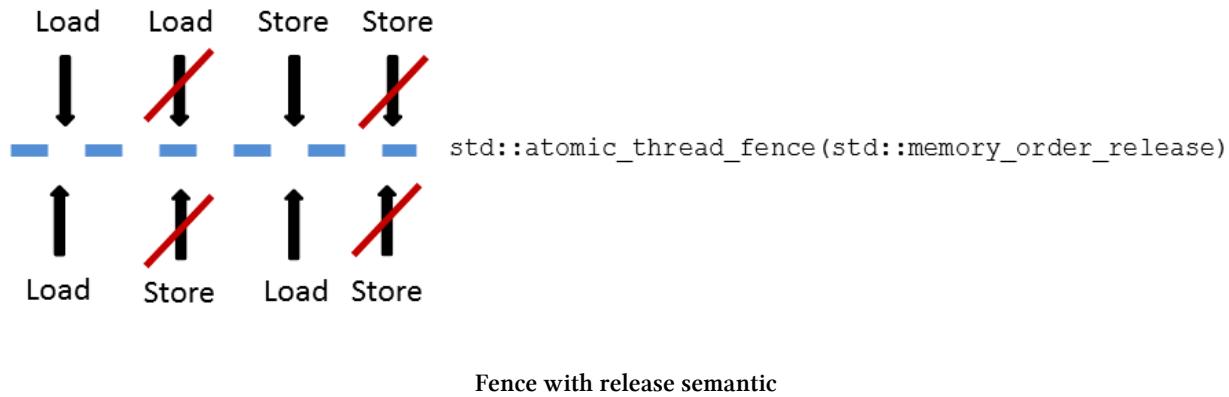
Release Operation

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

Also, the release fence.



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

Synchronisation 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.

Atomic Operations

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

Acquire-release ordering with atomics

```

1 // acquireRelease.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_acquire)));
24     cout << "*p2: " << *p2 << endl;
25     cout << "data: " << data << endl;
26     cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
27 }
28
29 int main(){
30
31     cout << endl;
32
33     thread t1(producer);
34     thread t2(consumer);
35
36     t1.join();
37     t2.join();
38
39     cout << endl;
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 emphasise 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. \Rightarrow 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.

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

```

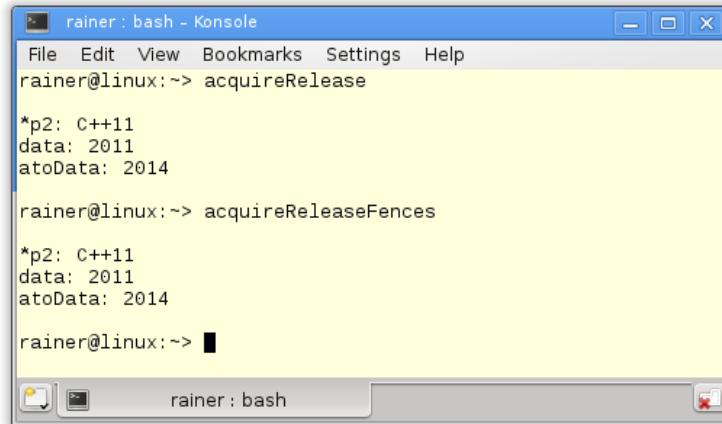
```

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 << endl;
27     cout << "data: " << data << endl;
28     cout << "atoData: " << atoData.load(memory_order_relaxed) << endl;
29 }
30
31 int main(){
32
33     cout << endl;
34
35     thread t1(producer);
36     thread t2(consumer);
37
38     t1.join();
39     t2.join();
40
41     delete ptr;
42
43     cout << endl;
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 key 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.



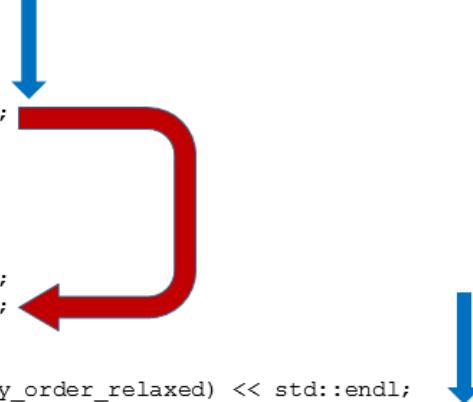
```
rainer@linux:~> acquireRelease
*p2: C++11
data: 2011
atoData: 2014

rainer@linux:~> acquireReleaseFences
*p2: C++11
data: 2011
atoData: 2014

rainer@linux:~> ■
```

Synchronisation with fences

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



```
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 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.



Synchronisation between the release fence and the acquire fence

The words from the [N4659: Working Draft, Standard for Programming Language C++²⁴](#) 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.

`std::atomic_signal_fence`

`std::atomic_signal_fence` establishes memory synchronisation 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`.

²⁴<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf>

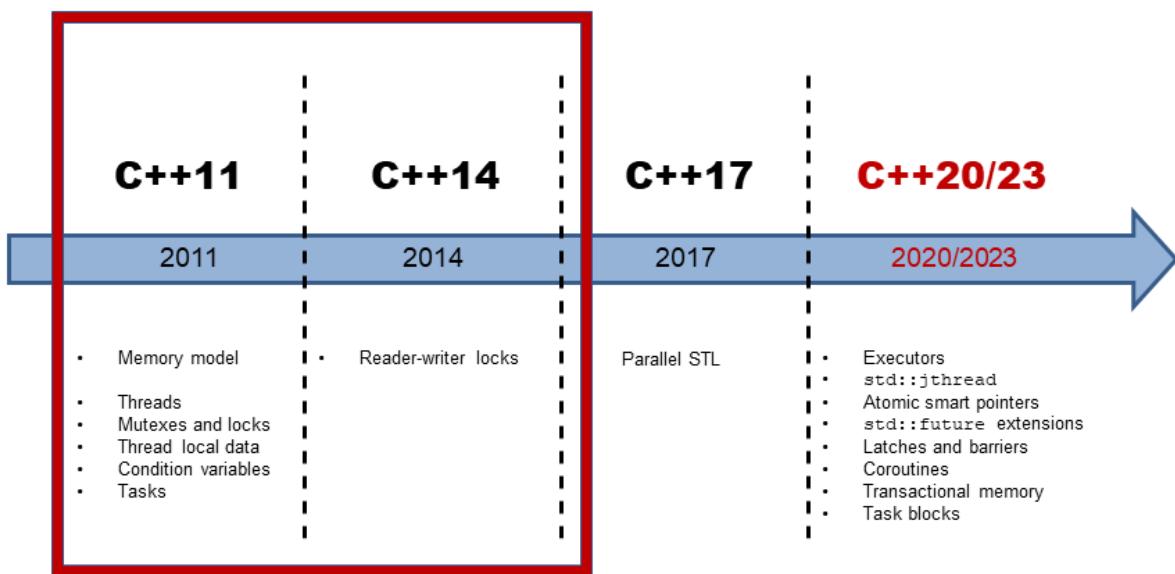
Synchronisation with a signal handler

```
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.

Multithreading

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



Multithreading in C++11 and C++14

Threads

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

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.

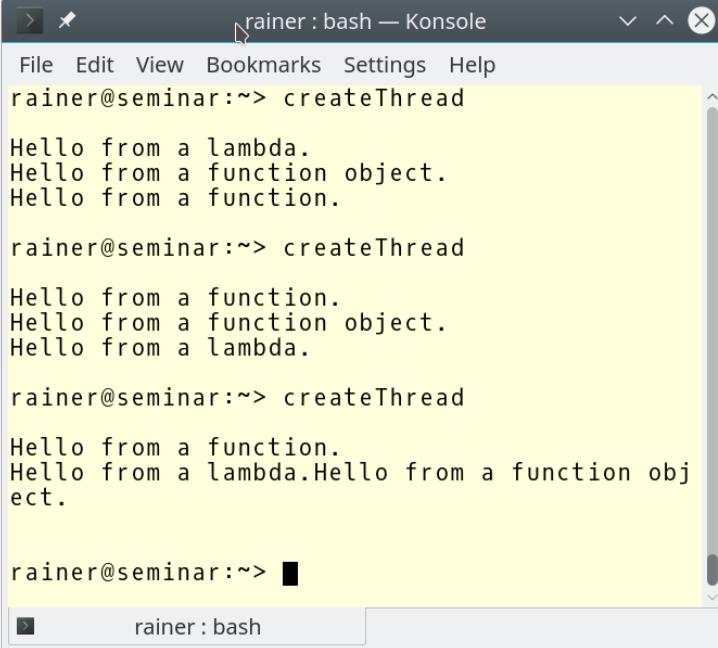
Creation of a thread with callable units

```
1 // createThread.cpp
2
3 #include <iostream>
4 #include <thread>
5
6 void helloFunction(){
7     std::cout << "Hello from a function." << std::endl;
8 }
9
10 class HelloFunctionObject{
11 public:
12     void operator()() const {
13         std::cout << "Hello from a function object." << std::endl;
14     }
15 };
16
17 int main(){
18
19     std::cout << std::endl;
20
21     std::thread t1(helloFunction);
22
23     HelloFunctionObject helloFunctionObject;
24     std::thread t2(helloFunctionObject);
25
26     std::thread t3([]{std::cout << "Hello from a lambda." << std::endl;});
27 }
```

```
28     t1.join();
29     t2.join();
30     t3.join();
31
32     std::cout << std::endl;
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.



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> createThread
Hello from a lambda.
Hello from a function object.
Hello from a function.

rainer@seminar:~> createThread
Hello from a function.
Hello from a function object.
Hello from a lambda.

rainer@seminar:~> createThread
Hello from a function.
Hello from a lambda.
Hello from a function object.

rainer@seminar:~> █
```

Creation of threads with various callables

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

The creator of the child - in our case this was the main thread - is responsible for the lifetime of the child.

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() << std::endl;});
9
10 }
```

But the program does not print the id.



Forget to join a thread

What's the reason for the exception?

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 subsequent 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.

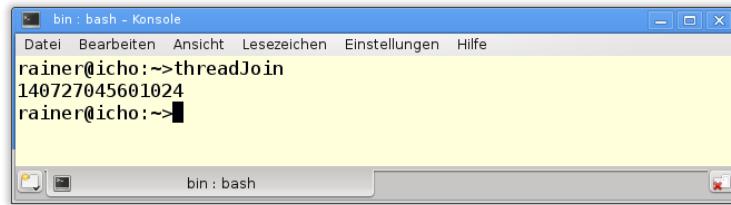
A thread `t` with a callable unit - you can create threads without a **callable unit** - is called joinable if neither a `t.join()` nor a `t.detach()` call happened. The destructor of a joinable thread throws the `std::terminate` exception. This was the reason the program execution of `threadWithoutJoin.cpp` 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

```
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() << std::endl;});
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 behaviour 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 manually take care of the lifetime of your threads `t`, you can encapsulate a `std::thread` in your wrapper class. This class should automatically call `t.join()` in its destructor if the thread `t` is still joinable. Of course, you can go the other way and call `t.detach()`; but you know, there is an issue with `detach`.

Anthony Williams created such a useful class and presented it in his excellent book [Concurrency in Action²⁵](#). 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.

```
// 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;
};
```

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 template²⁶](#). This means it can get an arbitrary number of arguments.

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

²⁵<https://www.manning.com/books/c-plus-plus-concurrency-in-action>

²⁶http://en.cppreference.com/w/cpp/language/parameter_pack

Copy or Reference

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

```
std::string s{"C++11"}  
  
std::thread t1([=]{ std::cout << s << std::endl;});  
t1.join();  
  
std::thread t2([&]{ std::cout << s << std::endl;});  
t2.detach();
```

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



Thread arguments by reference

To be honest, I cheated a little. Not 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 wrapper²⁷](#). This is quite straightforward with the helper function `std::ref`²⁸. `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 afterwards is detached from the lifetime of its creator. The lifetime of the string `s` is bound to the lifetime of the creator; the lifetime of the global object `std::cout` is bound to the lifetime of the main thread. So it is possible that the lifetime of `s` or the lifetime of `std::cout` is shorter than the lifetime of the thread t2. Now we are deep in the area of undefined behaviour.

Not convinced? Let's take a closer look at what undefined behaviour may look like.

²⁷http://en.cppreference.com/w/cpp/utility/functional/reference_wrapper

²⁸<http://en.cppreference.com/w/cpp/utility/functional/ref>

Passing the arguments to a thread by reference

```

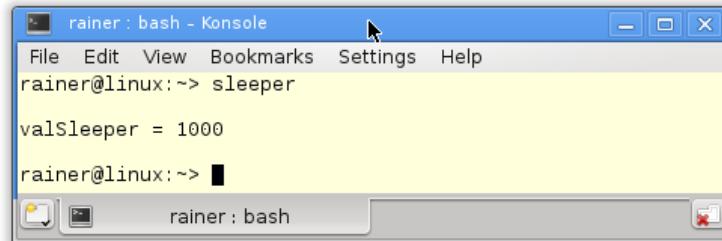
1 // threadArguments.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <thread>
6
7 class Sleeper{
8     public:
9         Sleeper(int& i_): i{i_}{};
10        void operator() (int k){
11            for (unsigned int j = 0; j <= 5; ++j){
12                std::this_thread::sleep_for(std::chrono::milliseconds(100));
13                i += k;
14            }
15            std::cout << std::this_thread::get_id() << std::endl;
16        }
17     private:
18         int& i;
19     };
20
21
22 int main(){
23
24     std::cout << std::endl;
25
26     int valSleeper = 1000;
27     std::thread t(Sleeper(valSleeper), 5);
28     t.detach();
29     std::cout << "valSleeper = " << valSleeper << std::endl;
30
31     std::cout << std::endl;
32
33 }
```

The question is: what value does `valSleeper` have in line 29? `valSleeper` is a global variable. 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 method 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**²⁹ (a German proverb), the result

²⁹https://de.wikipedia.org/wiki/Liste_gefl%C3%BCgelter_Worte/N#Nach_Adam_Riese

should be $1000 + 6 * 5 = 1030$.

However, what happened? Something is going very wrong.



Undefined behaviour 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 lifetime of the main thread ends before the child thread has performed its calculation or written its id to `std::cout`.

Both issues are [race conditions](#) because the result of the program depends on the interleaving of the operations. The race condition is the cause of the data race.

Fixing the data race is quite 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.

```
int main(){
    std::cout << std::endl;

    int valSleeper= 1000;
    std::thread t(Sleeper(valSleeper),5);
    t.join();
    std::cout << "valSleeper = " << valSleeper << std::endl;

    std::cout << std::endl;
}
```

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



```
rainer@linux:~> sleeper
139801113319168
valSleeper = 1030
rainer@linux:~> █
```

Fixed the undefined behaviour

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

Methods

Here is the interface of `std::thread` `t` in a concise table. For additional details, please refer to cppreference.com³⁰.

Methods of a thread	
Method	Description
<code>t.join()</code>	Waits until thread <code>t</code> has finished its executable unit.
<code>t.detach()</code>	Executes the created thread <code>t</code> independently of the creator.
<code>t.joinable()</code>	Returns true if thread <code>t</code> is still joinable.
<code>t.get_id()</code> and <code>std::this_thread::get_id()</code>	Returns the identity of the thread.
<code>std::thread::hardware_concurrency()</code>	Indicates the number of threads that can be run concurrently.
<code>std::this_thread::sleep_until(absTime)</code>	Puts thread <code>t</code> to sleep until the time point <code>absTime</code> .
<code>std::this_thread::sleep_for(relTime)</code>	Puts thread <code>t</code> to sleep for the time duration <code>relTime</code> .
<code>std::this_thread::yield()</code>	Enables the system to run another thread.
<code>t.swap(t2)</code> and <code>std::swap(t1, t2)</code>	Swaps the threads.

The static method `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 duration** as an argument.

³⁰<http://de.cppreference.com/w/cpp/thread/thread>



Access to the system-specific implementation

The threading interface is a wrapper around the underlying implementation. You can use the method `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` method performs a move when possible.

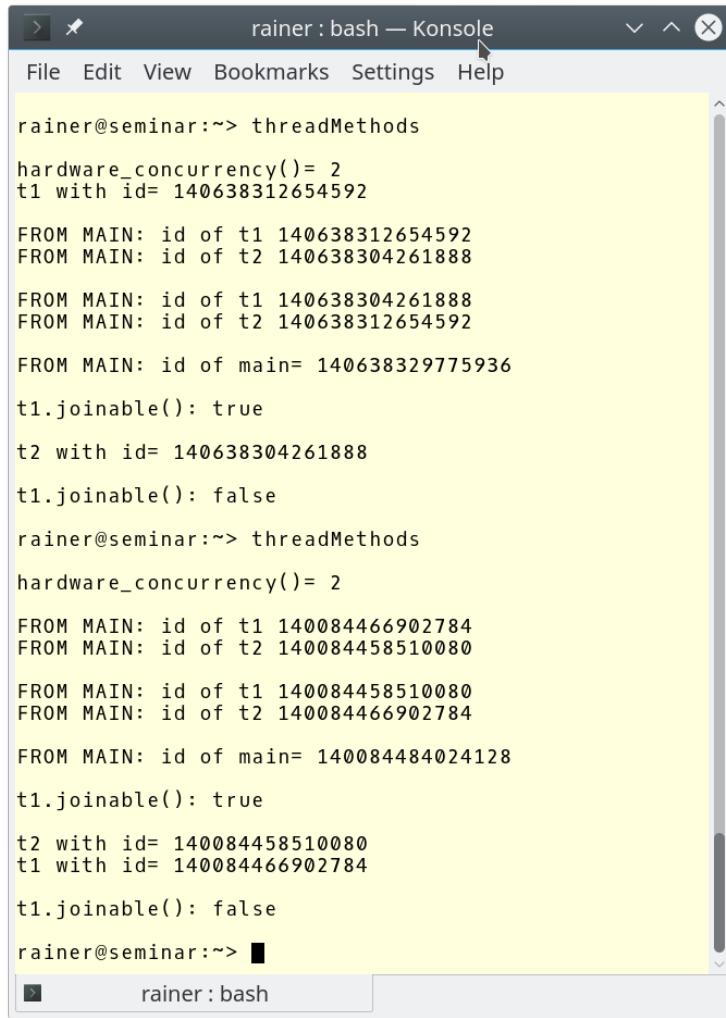
To conclude this subsection, here are a few of the mentioned methods in practice.

Methods of a thread

```
1 // threadMethods.cpp
2
3 #include <iostream>
4 #include <thread>
5
6 using namespace std;
7
8 int main(){
9
10    cout << boolalpha << endl;
11
12    cout << "hardware_concurrency()= " << thread::hardware_concurrency() << endl;
13
14    thread t1([]{cout << "t1 with id= " << this_thread::get_id() << endl;});
15    thread t2([]{cout << "t2 with id= " << this_thread::get_id() << endl;});
16
17    cout << endl;
18
19    cout << "FROM MAIN: id of t1 " << t1.get_id() << endl;
20    cout << "FROM MAIN: id of t2 " << t2.get_id() << endl;
21
22    cout << endl;
23    swap(t1,t2);
24
25    cout << "FROM MAIN: id of t1 " << t1.get_id() << endl;
26    cout << "FROM MAIN: id of t2 " << t2.get_id() << endl;
27
28    cout << endl;
29
30    cout << "FROM MAIN: id of main= " << this_thread::get_id() << endl;
31
32    cout << endl;
```

```
33
34     cout << "t1.joinable(): " << t1.joinable() << endl;
35
36     cout << endl;
37
38     t1.join();
39     t2.join();
40
41     cout << endl;
42
43     cout << "t1.joinable(): " << t1.joinable() << endl;
44
45     cout << endl;
46
47 }
```

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



```
rainer@seminar:~/threadMethods
hardware_concurrency()= 2
t1 with id= 140638312654592
FROM MAIN: id of t1 140638312654592
FROM MAIN: id of t2 140638304261888
FROM MAIN: id of t1 140638304261888
FROM MAIN: id of t2 140638312654592
FROM MAIN: id of main= 140638329775936
t1.joinable(): true
t2 with id= 140638304261888
t1.joinable(): false
rainer@seminar:~/threadMethods
hardware_concurrency()= 2
FROM MAIN: id of t1 140084466902784
FROM MAIN: id of t2 140084458510080
FROM MAIN: id of t1 140084458510080
FROM MAIN: id of t2 140084466902784
FROM MAIN: id of main= 140084484024128
t1.joinable(): true
t2 with id= 140084458510080
t1 with id= 140084466902784
t1.joinable(): false
rainer@seminar:~> █
```

Methods 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 guarantee 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.

Shared Data

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

The easiest way to visualise concurrent, unsynchronised read and write operations is to write something to `std::cout`.

Let's have a look.

Writing unsynchronised to `std::cout`

```
1 // coutUnsynchronised.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 !!!" << std::endl;
16        }
17    }
18 private:
19     std::string name;
20 };
21
22
23 int main(){
24
25     std::cout << std::endl;
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." << std::endl;
45
46     std::cout << std::endl;
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 receives notifications from all workers, he sends them home (line 43).

What a mess for such a simple workflow.

```
File Edit View Bookmarks Settings Help

rainer@suse:~> coutUnsynchronised
Boss: Let's start working.

Herb: Work      Bjarne Andrei1 done !!!
      Jenne: Work      Bart: : Work Work 1 done !!!1
      Scott done !!!: Work
1 done !!!
1 done !!!
: Work 1 done !!!
  Scott: Work 2 done !!!
  Bjarne: Work 2 done !!!
  Jenne: Work 2 done !!!
Andrei: Herb: Work Work 22 done !!! done !!!
  Bart: Work 2 done !!!

Andrei: Work 3 done !!!
  Bart:      Scott: Work Work 3 done !!!3 done !!!
  Bjarne: Work 3 done !!!
Herb: Work 3 done !!!
  Jenne: Work 3 done !!!
Boss: Let's go home.

rainer@suse:~> █
```

rainer : bash

Non-synchronised writing to std::cout

The most straightforward solution is to use a mutex.

Mutexes

Mutex stands for **mutual exclusion**. 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 a harmony.

Writing synchronised to std::cout

```
1 // coutSynchronised.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             coutMutex.lock();
20             std::cout << name << ":" << "Work " << i << " done !!" << std::endl;
21             coutMutex.unlock();
22         }
23     }
24 private:
25     std::string name;
26 };
27
28
29 int main(){
30
31     std::cout << std::endl;
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." << std::endl;
50
51     std::cout << std::endl;
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
Synchronised 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. It is possible that more output statements like those in the example 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

Method	mutex	recursive_mutex	timed_mutex	recursive_timed_mutex
<code>m.lock</code>	yes	yes	yes	yes
<code>m.try_lock</code>	yes	yes	yes	yes
<code>m.try_lock_for</code>			yes	yes
<code>m.try_lock_until</code>			yes	yes
<code>m.unlock</code>	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`³¹ 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

Method	shared_timed_mutex	shared_mutex
<code>m.lock</code>	yes	yes
<code>m.try_lock</code>	yes	yes
<code>m.try_lock_for</code>	yes	
<code>m.try_lock_until</code>	yes	
<code>m.unlock</code>	yes	yes
<code>m.lock_shared</code>	yes	yes
<code>m.try_lock_shared</code>	yes	yes
<code>m.try_lock_shared_for</code>	yes	
<code>m.try_lock_shared_until</code>	yes	
<code>m.unlock_shared</code>	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_mutex)` into a `std::shared_lock`. The methods `m.try_`

³¹http://en.cppreference.com/w/cpp/error/system_error

`m.lock_for(relTime)` and `m.try_lock_shared_for(relTime)`) need a relative **time duration**; the methods `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 methods `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.

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?

Exceptions and Unknown Code

The small code snippet has many issues.

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

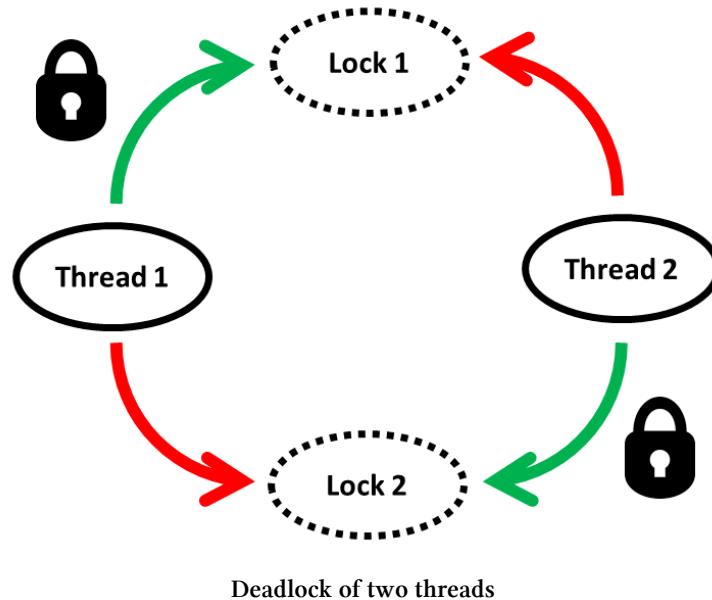
Here are the issues:

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 behaviour because `m` is not a recursive mutex. Most of the time, the undefined behaviour 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.

Lock Mutexes in Different Order

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



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 in such a way 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 to get the other mutex the other thread has to release it first. The expression "deadly embrace" describes this kind of deadlock very well.

Translating this picture into code is easy.

Locking mutexes in different order

```

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

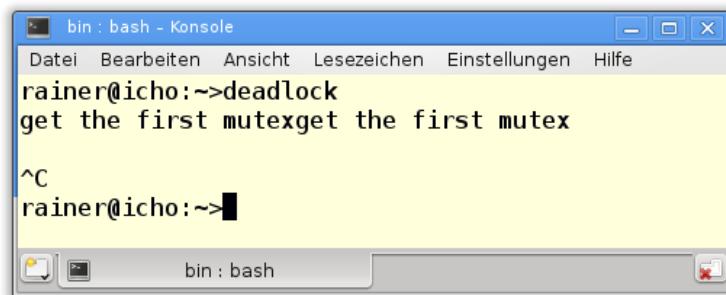
```

```

14     a.mut.lock();
15     std::cout << "get the first mutex" << std::endl;
16     std::this_thread::sleep_for(std::chrono::milliseconds(1));
17     b.mut.lock();
18     std::cout << "get the second mutex" << std::endl;
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 }
```

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 methods or members apart from a mutex).

A short sleep of about 1 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.

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

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.

```
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 maintains a kind of 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? The lifetime of `std::lock_guard` is limited by its scope and the scope is defined by the curly brackets³². 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. A function scope and loop scope also limit the lifetime of an object.

`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.

³²http://en.cppreference.com/w/cpp/language/scope#Block_scope

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 one of the current threads already owns the corresponding mutex and the mutex is not recursive, the behaviour 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.

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

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

The interface of `std::unique_lock`

Method	Description
<code>lk.lock()</code>	Locks the associated mutex.
<code>lk.try_lock()</code> and <code>lk.try_lock_for(relTime)</code> and <code>lk.try_lock_until(absTime)</code>	Tries to lock the associated mutex.
<code>lk.unlock()</code>	Unlocks the associated mutex.
<code>lk.release()</code>	Release the mutex. The mutex remains locked.
<code>lk.swap(lk2)</code> and <code>std::swap(lk, lk2)</code>	Swaps the locks.
<code>lk.mutex()</code>	Returns a pointer to the associated mutex.
<code>lk.owns_lock()</code> and operator <code>bool</code>	Checks if the lock <code>lk</code> 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 method `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 methods `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 methods `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`.

The method `lk.release()` returns the mutex; therefore, you have to unlock it manually.

Thanks to `std::unique_lock`, it is quite easy to lock many mutexes in one atomic step. Therefore you can overcome deadlocks by locking mutexes in a different order. Remember the deadlock from the subsection [Issues of Mutexes](#)?

Locking mutexes in different order

```
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" << std::endl;
16     std::this_thread::sleep_for(std::chrono::milliseconds(1));
17     b.mut.lock();
18     std::cout << "get the second mutex" << std::endl;
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

```
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" << endl;
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" << endl;
23
24     cout << "      Thread: " << this_thread::get_id() << " get both mutex" << endl;
25     lock(guard1,guard2);
26     // do something with a and b
27 }
28
```

```

29 int main(){
30
31     cout << endl;
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 << endl;
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 owner of the mutex. Thanks to the variadic template `std::lock`, the lock operation is performed in an atomic step (line 25).



Atomic locking with `std::lock`

`std::lock` can lock its mutexes in an atomic step. `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 mutexes are locked by an unspecified series of calls to `lock`, `try_lock`, and `unlock`. If a call to `lock` or `unlock` causes an exception, `unlock` is called for any locked mutex before rethrowing.

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`

```

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

```

`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 at the same time, 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 do help to minimise the bottleneck.

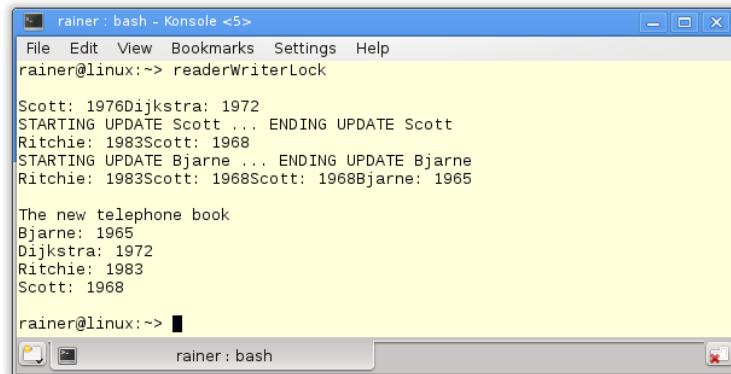
A telephone book is a typical example using a reader-writer lock. Usually, many people want to look up a telephone number, but only a few want to change them. Let's look at an example.

Reader-writer locks

```
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 << std::endl;
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 << std::endl;
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 << std::endl;
54
55     std::cout << "\nThe new telephone book" << std::endl;
56     for (auto teleIt: teleBook){
57         std::cout << teleIt.first << ":" << teleIt.second << std::endl;
58     }
59
60     std::cout << std::endl;
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 at the same time, 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. At the end, the program displays the updated telephone book (lines 55 - 58).



```
File Edit View Bookmarks Settings Help
rainer@linux:~> readerWriterLock
Scott: 1976Dijkstra: 1972
STARTING UPDATE Scott ... ENDING UPDATE Scott
Ritchie: 1983Scott: 1968
STARTING UPDATE Bjarne ... ENDING UPDATE Bjarne
Ritchie: 1983Scott: 1968Bjarne: 1965

The new telephone book
Bjarne: 1965
Dijkstra: 1972
Ritchie: 1983
Scott: 1968
rainer@linux:~>
```

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

The screenshot shows that the output of the reading threads 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 behaviour.

Undefined behaviour

The program has undefined behaviour. 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 at the same time and at least one of them is a writer. This exact scenario may occur during program execution. One of the features of the associative container 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

```
1 // readerWriterLocksResolved.cpp
2
3 ...
4
5 void printNumber(const std::string& na){
6     std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
7     auto searchEntry = teleBook.find(na);
8     if(searchEntry != teleBook.end()){
9         std::cout << searchEntry->first << ":" << searchEntry->second << std::endl;
10    }
11    else {
12        std::cout << na << " not found!" << std::endl;
13    }
14 }
```

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

```

STARTING UPDATE Scott ... ENDING UPDATE Scott
STARTING UPDATE Bjarne ... ENDING UPDATE Bjarne
Dijkstra: 1972
Ritchie: 1983
Scott: 1968
Scott: 1968

Bjarne: 1965
1968
Ritchie: 1983

The new telephone book
Bjarne: 1965
Dijkstra: 1972
Ritchie: 1983
Scott: 1968

rainer@suse:~> readerWriterLocksResolved
ScottRitchie: 1983
1976
Scott: 1976
Ritchie: 1983

DijkstraScott: 1976
1972
Scott: 1976
Bjarne not found!

STARTING UPDATE Scott ... ENDING UPDATE Scott
STARTING UPDATE Bjarne ... ENDING UPDATE Bjarne

The new telephone book
Bjarne: 1965
Dijkstra: 1972
Ritchie: 1983
Scott: 1968

rainer@suse:~> ■

```

Data race resolved

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

Thread-safe Initialisation

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

There are three ways in C++ to initialise 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 Initialisation in the main thread

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

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 initialised 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 be met to initialise it at compile time.

- They must not have virtual methods or a virtual base class.
- Their constructor must be empty and itself be a constant expression.
- Every base class and each non-static member must be initialized.
- Their methods, 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 }
```

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 above-mentioned execution 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.

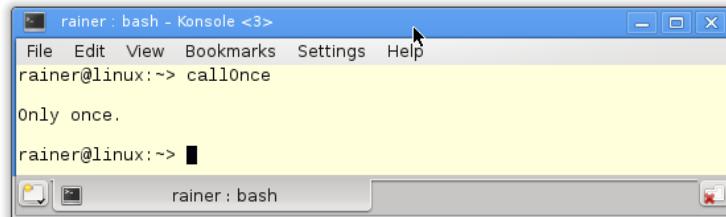
The short example demonstrates the application of `std::call_once` and the `std::once_flag`. Both of them are declared in the header `<mutex>`.

User defined constant expressions

```
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." << std::endl; });
11 }
12
13 void do_once2(){
14     std::call_once(onceFlag, [](){ std::cout << "Only once2." << std::endl; });
15 }
16
17 int main(){
18
19     std::cout << std::endl;
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 << std::endl;
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 initialised in a thread-safe way.

Singleton pattern with `std::call_once` and the `std::once_flag`

```
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 public:
17     MySingleton(const MySingleton&) = delete;
18     MySingleton& operator=(const MySingleton&) = delete;
19
20     static MySingleton* getInstance(){
21         call_once(initInstanceFlag, MySingleton::initSingleton);
22         return instance;
23     }
24
25     static void initSingleton(){
26         instance = new MySingleton();
27     }
28 };
29
30 MySingleton* MySingleton::instance = nullptr;
31 once_flag MySingleton::initInstanceFlag;
32
33 int main(){
34
35     cout << endl;
36
37     cout << "MySingleton::getInstance(): " << MySingleton::getInstance() << endl;
38     cout << "MySingleton::getInstance(): " << MySingleton::getInstance() << endl;
39
40     cout << endl;
41
42 }

```

Let's first review the static flag `initInstanceFlag`. It is declared in line 11 and initialised in line 31. The static method `getInstance` (lines 20 - 23) uses the flag `initInstanceFlag` to ensure that the static method `initSingleton` (line 25 - 27) is executed exactly once. The singleton is created in the body of the method.

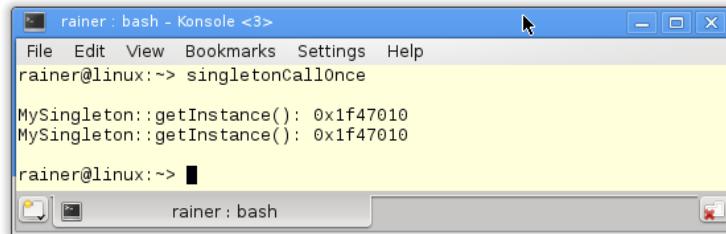
default and delete

You can request special methods from the compiler by using the keyword `default`. These methods are special because the compiler can create them for us.

The result of annotating a method with `delete` is that the compiler generated method 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`³³.

³³<https://isocpp.org/wiki/faq/cpp11-language-classes>

The `MySingleton::getInstance()` method displays the address of the singleton.



```
rainer : bash - Konsole <3>
File Edit View Bookmarks Settings Help
rainer@linux:~/> singletonCallOnce
MySingleton::getInstance(): 0x1f47010
MySingleton::getInstance(): 0x1f47010
rainer@linux:~/>
```

Thread-safe implementation of the singleton with `std::call_once` and `std::once_flag`

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 the usage. This characteristic is the basis of the so-called Meyers Singleton, named after [Scott Meyers](#)³⁴. 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 initialised in a thread-safe way.

Here is the thread-safe Meyers Singleton pattern.

The thread-safe Meyers Singleton pattern

```
1 // meyersSingleton.cpp
2
3 class MySingleton{
4 public:
5     static MySingleton& getInstance(){
6         static MySingleton instance;
7         return instance;
8     }
9 private:
10    MySingleton();
11    ~MySingleton();
12    MySingleton(const MySingleton&)= delete;
13    MySingleton& operator=(const MySingleton&)= delete;
14
15 };
16
17 MySingleton::MySingleton()= default;
18 MySingleton::~MySingleton()= default;
19
20 int main(){
21 }
```

³⁴https://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.

Thread-Local Data

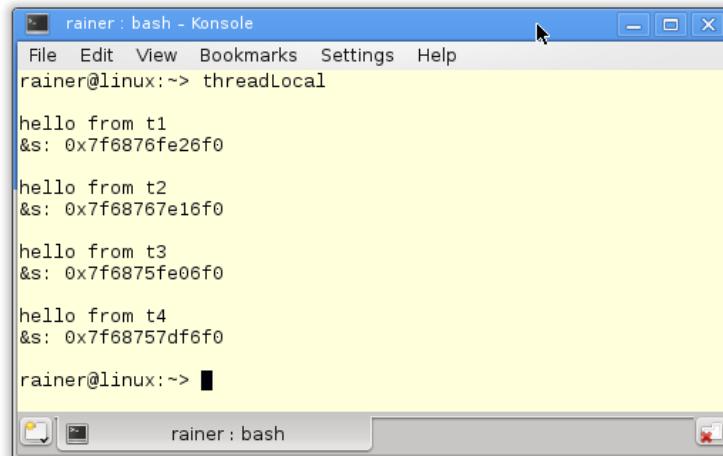
Thread-local data, also known as thread-local storage, is created for each thread separately. It behaves like static data because it's bound for the lifetime of the thread and is created at its first usage. Thread-local data belongs exclusively to the thread.

Thread-local data

```
1 // threadLocal.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 addThreadLocal(std::string const& s2){
13
14     s += s2;
15     // protect std::cout
16     std::lock_guard<std::mutex> guard(coutMutex);
17     std::cout << s << std::endl;
18     std::cout << "&s: " << &s << std::endl;
19     std::cout << std::endl;
20
21 }
22
23 int main(){
24
25     std::cout << std::endl;
26
27     std::thread t1(addThreadLocal, "t1");
28     std::thread t2(addThreadLocal, "t2");
29     std::thread t3(addThreadLocal, "t3");
30     std::thread t4(addThreadLocal, "t4");
31
32     t1.join();
33     t2.join();
34     t3.join();
35     t4.join();
```

```
36
37 }
```

By 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.



```
rainer : bash - Konsole
File Edit View Bookmarks Settings Help
rainer@linux:~> threadLocal
hello from t1
&s: 0x7f6876fe26f0

hello from t2
&s: 0x7f68767e16f0

hello from t3
&s: 0x7f6875fe06f0

hello from t4
&s: 0x7f68757df6f0
rainer@linux:~> █
```

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 string `s`. First, each output shows a new `thread-local` string, second, each string `s` has a different address.

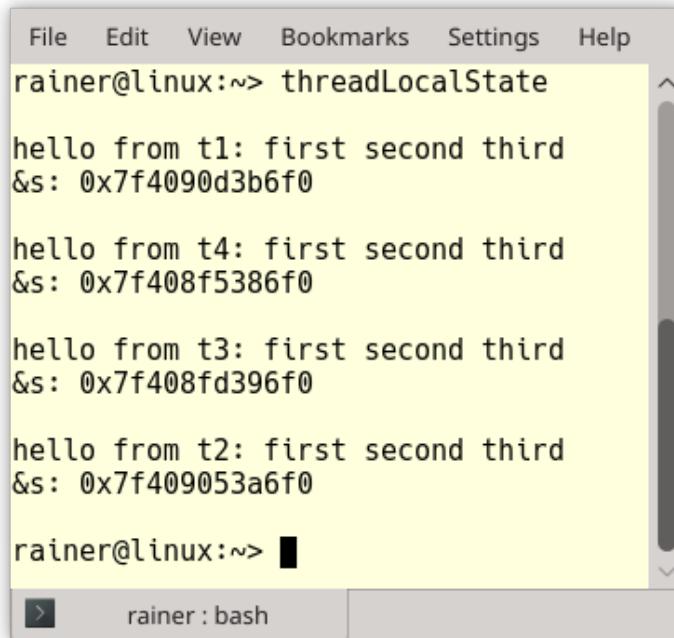
I often discuss in my seminars: What is the difference between a `static`, a `thread_local`, and a local variable? A `static` variable is bound to the lifetime of the main thread, a `thread_local` variable is bound to the lifetime of its thread, and a local variable is bound to the lifetime of the scope in which it was created. Here is a variation of the previous program `threadLocal.cpp` to make my point clear.

State between function calls

```
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 << std::endl;
34     std::cout << "&s: " << &s << std::endl;
35     std::cout << std::endl;
36
37 }
38
39 int main(){
40
41     std::cout << std::endl;
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 kind of thread-local state between the function calls `first`, `second`, and `third` (lines 28 - 30). The output shows that the strings are independent.



```
File Edit View Bookmarks Settings Help
rainer@linux:~> threadLocalStorage
hello from t1: first second third
&s: 0x7f4090d3b6f0

hello from t4: first second third
&s: 0x7f408f5386f0

hello from t3: first second third
&s: 0x7f408fd396f0

hello from t2: first second third
&s: 0x7f409053a6f0

rainer@linux:~> █
> rainer : bash
```

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 behaviour.

In contrast to thread-local data, condition variables are not easy to use.

Condition Variables

Condition variables enable threads to be synchronised 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 methods of the condition variable cv

Method	Description
<code>cv.notify_one()</code>	Notifies a waiting thread.
<code>cv.notify_all()</code>	Notifies all waiting threads.
<code>cv.wait(lock, ...)</code>	Waits for the notification while holding a <code>std::unique_lock</code> .
<code>cv.wait_for(lock, relTime, ...)</code>	Waits for a time duration for the notification while holding a <code>std::unique_lock</code> .
<code>cv.wait_until(lock, absTime, ...)</code>	Waits until a time point for the notification while holding a <code>std::unique_lock</code> .
<code>cv.native_handle()</code>	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 does stay in the wait state. Before we cover the gory details - which are the three dots in the wait operations - of condition variables, here is an example.

First usage of condition variables

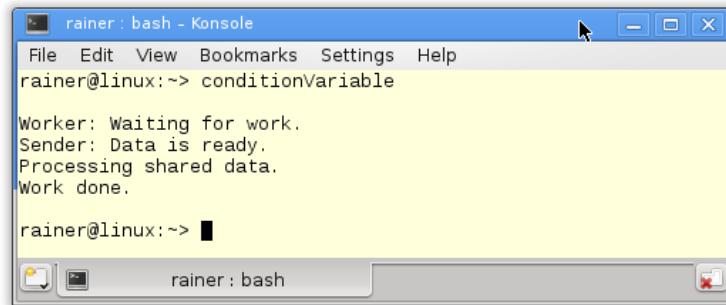
```

1 // conditionVariable.cpp
2
3 #include <iostream>
4 #include <condition_variable>
5 #include <mutex>
6 #include <thread>
7
8 std::mutex mutex_;
9 std::condition_variable condVar;
10
11 bool dataReady{false};
12
13 void doTheWork(){
14     std::cout << "Processing shared data." << std::endl;
15 }
```

```
16
17 void waitingForWork(){
18     std::cout << "Worker: Waiting for work." << std::endl;
19     std::unique_lock<std::mutex> lck(mutex_);
20     condVar.wait(lck, []{ return dataReady; });
21     doTheWork();
22     std::cout << "Work done." << std::endl;
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." << std::endl;
31 condVar.notify_one();
32 }
33
34 int main(){
35
36     std::cout << std::endl;
37
38     std::thread t1(waitingForWork);
39     std::thread t2(setDataReady);
40
41     t1.join();
42     t2.join();
43
44     std::cout << std::endl;
45
46 }
```

The program has two child threads: t1 and t2. They get their work package `waitingForWork` and `setDataRead` in lines 38 and 39. `setDataRead` 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 case of the receiver, a `std::unique_lock` is necessary because it frequently locks and unlocks its mutex.

Here is the output of the program.



```
rainer@linux:~> conditionVariable
Worker: Waiting for work.
Sender: Data is ready.
Processing shared data.
Work done.
rainer@linux:~> █
```

Sychronisation 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](#)³⁵. The generalised `std::condition_variable_any` supports the same interface such as `std::condition_variable`.

The Predicate

Maybe you are wondering 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

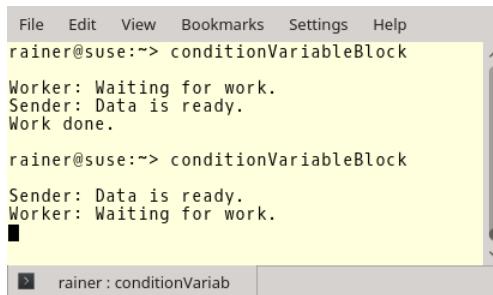
```
1 // conditionVariableBlock.cpp
2
3 #include <iostream>
4 #include <condition_variable>
5 #include <mutex>
6 #include <thread>
7
8 std::mutex mutex_;
9 std::condition_variable condVar;
10
11 void waitingForWork(){
12
13     std::cout << "Worker: Waiting for work." << std::endl;
14
15     std::unique_lock<std::mutex> lck(mutex_);
16     condVar.wait(lck);
```

³⁵<http://en.cppreference.com/w/cpp/concept/BasicLockable>

```

17     // do the work
18     std::cout << "Work done." << std::endl;
19
20 }
21
22 void setDataReady(){
23
24     std::cout << "Sender: Data is ready." << std::endl;
25     condVar.notify_one();
26
27 }
28
29 int main(){
30
31     std::cout << std::endl;
32
33     std::thread t1(setDataReady);
34     std::thread t2(waitingForWork);
35
36     t1.join();
37     t2.join();
38
39     std::cout << std::endl;
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).



```

File Edit View Bookmarks Settings Help
rainer@suse:~> conditionVariableBlock
Worker: Waiting for work.
Sender: Data is ready.
Work done.

rainer@suse:~> conditionVariableBlock
Sender: Data is ready.
Worker: Waiting for work.

```

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 victim to two known phenomena: lost wakeup and spurious wakeup.

Lost Wakeup and Spurious Wakeup

Lost Wakeup

The phenomenon of the lost wakeup 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 synchronisation mechanism: “The `condition_variable` class is a synchronisation 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

It may happen that the receiver wakes up, although no notification happened. At a minimum [POSIX Threads](#)³⁶ and the [Windows API](#)³⁷ 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.

The Wait Workflow

The waiting thread has quite a complicated workflow.

Here are the two key 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

³⁶https://en.wikipedia.org/wiki/POSIX_Threads

³⁷https://en.wikipedia.org/wiki/Windows_API

- `condVar.wait(lck)`: If the condition_variable `condVar` is in the waiting state 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 publish the modification to the waiting thread correctly.



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 synchronised. This [race condition](#) may cause a [deadlock](#). What does that mean: published, but not correctly synchronised. Let's have once more a closer look at the wait workflow and assume that `deadReady` is an 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 in the waiting state. This means execution of the thread is in the source snippet between line 2 and 4 (see the comment time window). The result is that the notification is lost. Afterwards the thread goes back in the waiting state and presumably sleeps forever.

This wouldn't have happened if `dataReady` had been protected by a mutex. Because of the synchronisation with the mutex, the notification would only be sent if the condition variable and, therefore, the receiver thread is in the waiting state.

In most of the use-cases, tasks are the less error-prone way to synchronise threads. I compare condition variables and tasks in the section [Returning a Notification](#).

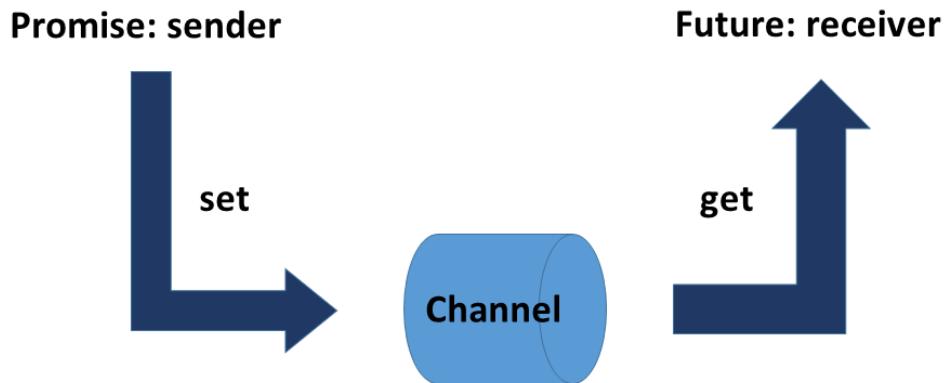
Tasks

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 the two associated components: a promise and a future. Both are connected via a data channel. The promise executes the work packages and puts the result in the data channel; the associated future picks up the result. Both communication endpoints can run in separate threads. It is special that the future can pick up the result at a later time; therefore, the calculation of the result by the promise is independent of the query of the result by the associated future.



Regard tasks as data channels 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 in different threads. The promise puts its result in the data channel. The future picks it up later.



Tasks as data channels between communication endpoints

Tasks versus Threads

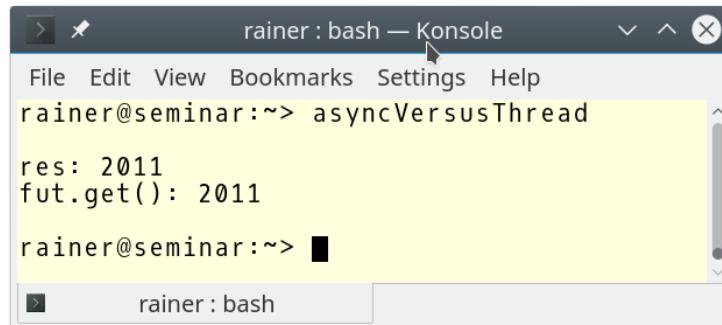
Tasks are very different threads.

std::async versus threads

```
1 // asyncVersusThread.cpp
2
3 #include <future>
4 #include <thread>
5 #include <iostream>
6
7 int main(){
8
9     std::cout << std::endl;
10
11    int res;
12    std::thread t([&]{ res = 2000 + 11; });
13    t.join();
14    std::cout << "res: " << res << std::endl;
15
16    auto fut= std::async([]{ return 2000 + 11; });
17    std::cout << "fut.get(): " << fut.get() << std::endl;
18
19    std::cout << std::endl;
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 result of the calculation. This `fut.get` call is blocking.

Here is the output of the program.



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> asyncVersusThread
res: 2011
fut.get(): 2011
rainer@seminar:~>
```

Tasks versus threads

Based on this program, I want to emphasise 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
Synchronisation	via <code>join()</code> (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, the communication of a task is more explicit. The future can request the result of the task only once (by calling `fut.get()`). Calling it more than once results in undefined behaviour. 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.

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 utilisation 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`.

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](#)³⁸, the expression is evaluated immediately - in the case of [lazy evaluation](#)³⁹ 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

```

1 // asyncLazy.cpp
2
3 #include <chrono>
4 #include <future>
5 #include <iostream>
6
7 int main(){
8
9     std::cout << std::endl;
10
11    auto begin= std::chrono::system_clock::now();
12
13    auto asyncLazy=std::async(std::launch::deferred,
14                             []{ return std::chrono::system_clock::now(); });

```

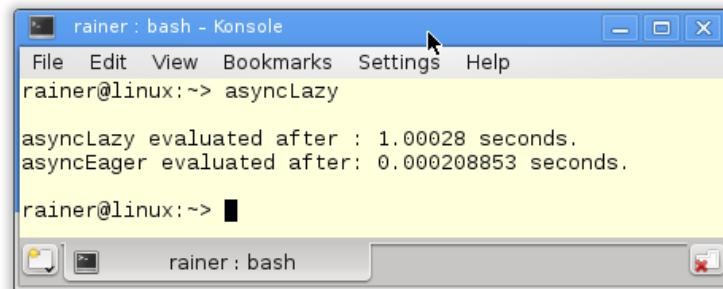
³⁸https://en.wikipedia.org/wiki/Eager_evaluation

³⁹https://en.wikipedia.org/wiki/Lazy_evaluation

```
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." << std::endl;
29     std::cout << "asyncEager evaluated after: " << eagerDuration
30             << " seconds." << std::endl;
31
32     std::cout << std::endl;
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.asyncEager.get()` 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.

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 so it can 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.

```
auto fut= std::async([]{ return 2011; });
std::cout << fut.get() << std::endl;

std::async(std::launch::async,
    []{ std::cout << "fire and forget" << std::endl; });
```

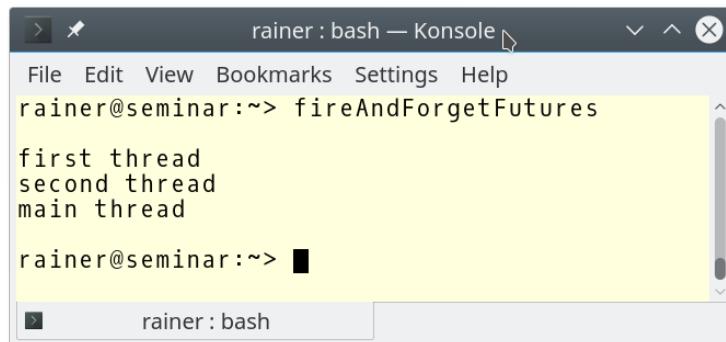
Fire and forget futures look very promising but have a big drawback. A future that is created by `std::async` waits on its destructor, until its promise is done. In this context, waiting is not very different from blocking. The future blocks the progress of the program in its destructor. This becomes more evident, when you use fire and forget futures. What seems to be concurrent actually runs sequentially.

Fire and forget futures

```
1 // fireAndForgetFutures.cpp
2
3 #include <chrono>
4 #include <future>
5 #include <iostream>
6 #include <thread>
7
8 int main(){
9
10    std::cout << std::endl;
11
12    std::async(std::launch::async, []{
13        std::this_thread::sleep_for(std::chrono::seconds(2));
14        std::cout << "first thread" << std::endl;
15    });
16
17    std::async(std::launch::async, []{
18        std::this_thread::sleep_for(std::chrono::seconds(1));
19        std::cout << "second thread" << std::endl;
20    });
21
22    std::cout << "main thread" << std::endl;
23}
```

```
24     std::cout << std::endl;
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 precisely 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.

Concurrent Calculation

The calculation of the scalar product can be spread across four asynchronous function calls.

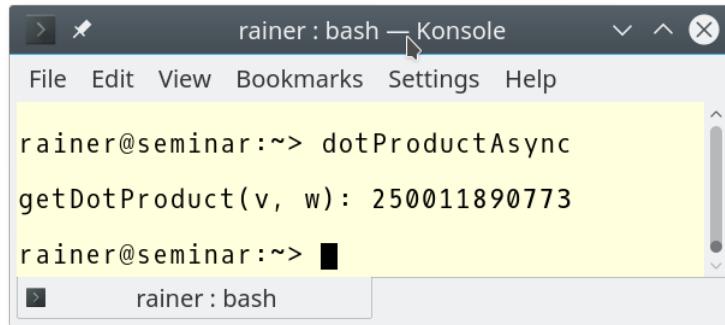
Dot product with four futures

```
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 << endl;
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) << endl;
```

```
59
60     cout << endl;
61
62 }
```

The program uses the functionality of the random and the time libraries. Both libraries are 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). Internally, `std::async` uses the standard template library algorithm `std::inner_product`. The return statement sums up the results of the futures.



```
rainer@seminar:~> dotProductAsync
getDotProduct(v, w): 250011890773
rainer@seminar:~> █
rainer@seminar:~> █
```

Scalar product with four asynchronous function calls

`std::packaged_task` is also usually used to perform a concurrent computation.

`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:

```
std::packaged_task<int(int, int)> sumTask([](int a, int b){ return a + b; });
```

II. Create a future:

```
std::future<int> sumResult= sumTask.get_future();
```

III. Perform the calculation:

```
    sumTask(2000, 11);
```

IV. Query the result:

```
    sumResult.get();
```

Here is an example showing the four steps.

Concurrency with `std::packaged_task`

```
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 << std::endl;
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 << std::endl;
65
66     std::cout << std::endl;
67
68 }

```

The purpose of the program is it 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 Summenformel⁴⁰](#). 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

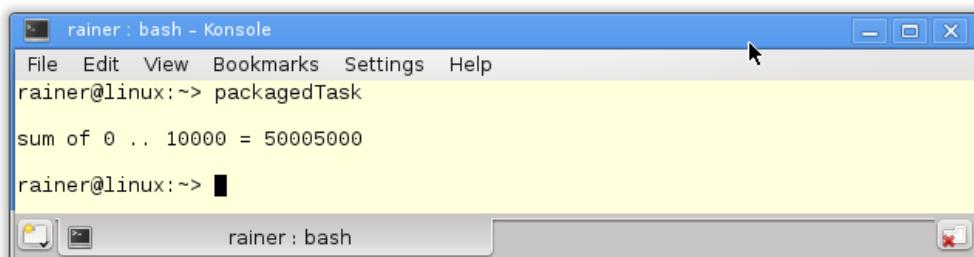
⁴⁰https://de.wikipedia.org/wiki/Gau%C3%9Fsche_Summenformel

- 15). The call operator sums up all numbers from `beg` to `end - 1` and returns the sum as the 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()`.

III. Perform the calculations: Now the calculation takes place. The `packaged_task` are moved onto the `std::deque`⁴¹ (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 for 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 `anstd::packaged_task` pack.

The methods of the `std::packaged_task`

Method	Description
<code>pack.swap(pack2)</code> and <code>std::swap(pack, pack2)</code>	Swaps the task objects.
<code>pack.valid()</code>	Checks if the task object has a valid function.
<code>pack.get_future()</code>	Returns the future.
<code>pack.make_ready_at_thread_exit(ex)</code>	Executes the function. The result is available if the current thread exists.
<code>pack.reset()</code>	Resets the state of the task. Abandons the stored results from previous executions.

⁴¹<http://en.cppreference.com/w/cpp/container/deque>

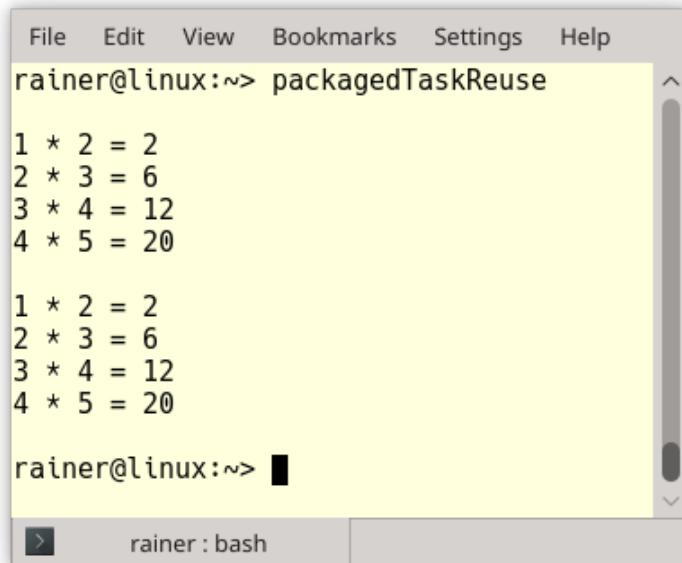
A `std::packaged_task` can be in contrast to a `std::async`, or a `std::promise` reset and reused. The following example shows this special use-case on a `std::packaged_task`.

Reuse of a `std::packaged_task`

```
1 // packagedTaskReuse.cpp
2
3 #include <functional>
4 #include <future>
5 #include <iostream>
6 #include <utility>
7 #include <vector>
8
9 void calcProducts(std::packaged_task<int(int, int)>& task,
10                   const std::vector<std::pair<int, int>>& pairs){
11     for (auto& pair: pairs){
12         auto fut = task.get_future();
13         task(pair.first, pair.second);
14         std::cout << pair.first << " * " << pair.second << " = " << fut.get()
15             << std::endl;
16         task.reset();
17     }
18 }
19
20 int main(){
21
22     std::cout << std::endl;
23
24     std::vector<std::pair<int, int>> allPairs;
25     allPairs.push_back(std::make_pair(1, 2));
26     allPairs.push_back(std::make_pair(2, 3));
27     allPairs.push_back(std::make_pair(3, 4));
28     allPairs.push_back(std::make_pair(4, 5));
29
30     std::packaged_task<int(int, int)> task{[](int fir, int sec){
31         return fir * sec; }
32     };
33
34     calcProducts(task, allPairs);
35
36     std::cout << std::endl;
37
38     std::thread t(calcProducts, std::ref(task), allPairs);
39     t.join();
```

```
40
41     std::cout << std::endl;
42
43 }
```

The function `calcProduct` (line 9) gets two arguments: the task and the vector of int-pairs. It uses the task to calculate for each int-pair the product (line 13). In the end, the task is reset in line 16. `calcProduct` runs in the main-thread (line 34) and in a separate thread (line 38). Here is the output of the program.



```
File Edit View Bookmarks Settings Help
rainer@linux:~> packagedTaskReuse
1 * 2 = 2
2 * 3 = 6
3 * 4 = 12
4 * 5 = 20

1 * 2 = 2
2 * 3 = 6
3 * 4 = 12
4 * 5 = 20

rainer@linux:~> █
```

Reuse of a `std::packaged_task`

`std::promise` and `std::future`

The class templates `std::promise` and `std::future` provide you with the 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++20/23 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

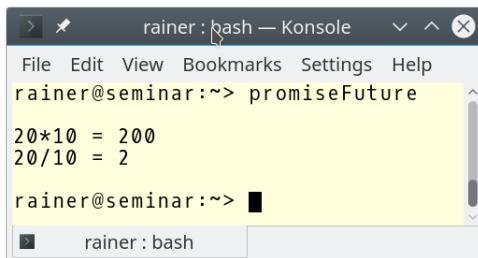
```
1 // promiseFuture.cpp
2
3 #include <future>
4 #include <iostream>
5 #include <thread>
6 #include <utility>
7
8 void product(std::promise<int>&& intPromise, int a, int b){
9     intPromise.set_value(a*b);
10 }
11
12 struct Div{
13
14     void operator() (std::promise<int>&& intPromise, int a, int 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 << std::endl;
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() << std::endl;
42     std::cout << "20/10 = " << divResult.get() << std::endl;
```

```

43
44     prodThread.join();
45
46     divThread.join();
47
48     std::cout << std::endl;
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 multiplikation with tasks

`std::promise`

`std::promise` enables you to set a value, a notification, or an exception. In addition the promise can provide its result in a delayed fashion.

The methods of the `std::promise` prom

Method	Description
<code>prom.swap(prom2)</code> and <code>std::swap(prom, prom2)</code>	Swaps the promises.
<code>prom.get_future()</code>	Returns the future.
<code>prom.set_value(val)</code>	Sets the value.
<code>prom.set_exception(ex)</code>	Sets the exception.

The methods of the `std::promise` `prom`

Method	Description
<code>prom.set_value_at_thread_exit(val)</code>	Stores the value and makes it ready if the promise exits.
<code>prom.set_exception_at_thread_exit(ex)</code>	Stores the exception and makes it ready if the promise exits.

If the promise sets the value or the exception more than once, a `std::future_error` exception is thrown.

`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 methods of the future `fut`

Method	Description
<code>fut.share()</code>	Returns a <code>std::shared_future</code> . Transfers the shared state.
<code>fut.get()</code>	Returns the result which can be a value or an exception.
<code>fut.valid()</code>	Checks if the shared state is available. After calling <code>fut.get()</code> it returns false.
<code>fut.wait()</code>	Waits for the result.
<code>fut.wait_for(relTime)</code>	Waits for the result, but not longer than for a <code>relTime</code> . Returns a <code>std::future_status</code> .
<code>fut.wait_until(abstime)</code>	Waits for the result, but not longer than until <code>abstime</code> . Returns a <code>std::future_status</code> .

In contrast to the `wait` call, the `wait_for` and `wait_until` variants return the status of the future.

`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

```
1 // waitFor.cpp
2
3 #include <iostream>
4 #include <future>
5 #include <thread>
6 #include <chrono>
7
8 using namespace std::literals::chrono_literals;
9
10 void getAnswer(std::promise<int> intPromise){
11     std::this_thread::sleep_for(3s);
12     intPromise.set_value(42);
13 }
14
15 int main(){
16     std::cout << std::endl;
17
18     std::promise<int> answerPromise;
19     auto fut = answerPromise.get_future();
20
21 }
```

```
22     std::thread prodThread(getAnswer, std::move(answerPromise));
23
24     std::future_status status{};
25     do{
26         status = fut.wait_for(0.2s);
27         std::cout << "... doing something else" << std::endl;
28     } while (status != std::future_status::ready);
29
30     std::cout << std::endl;
31
32     std::cout << "The Answer: " << fut.get() << '\n';
33
34     prodThread.join();
35
36     std::cout << std::endl;
37 }
```

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.

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`.

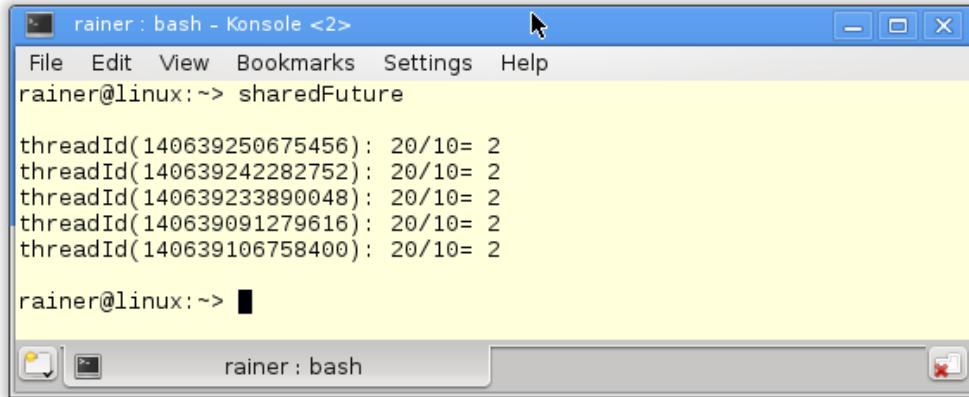
Taking a future with a `std::shared_future`

```
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() << std::endl;
29
30     }
31
32 };
33
34 int main(){
35
36     std::cout << std::endl;
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 separate 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 << std::endl;
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 57 - 61). It's important to emphasise 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.



```
rainer : bash - Konsole <2>
File Edit View Bookmarks Settings Help
rainer@linux:~> sharedFuture
threadId(140639250675456): 20/10= 2
threadId(140639242282752): 20/10= 2
threadId(140639233890048): 20/10= 2
threadId(140639091279616): 20/10= 2
threadId(140639106758400): 20/10= 2
rainer@linux:~> █
```

Taking a `std::shared_future`

As I previously mentioned it you can create a `std::shared_future` from a `std::future` by using its method `share`.

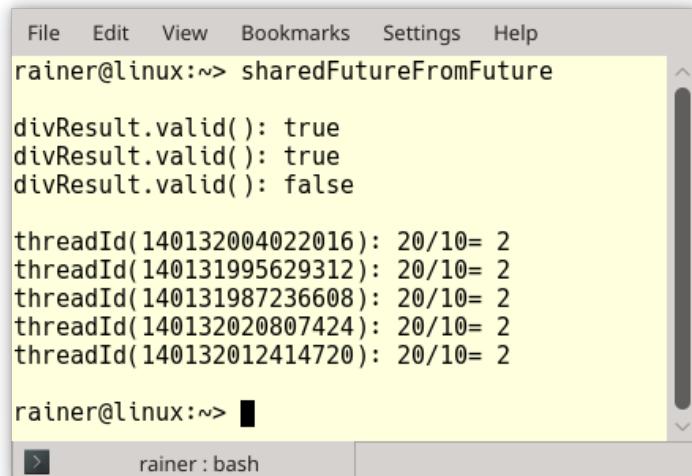
Creating a `std::shared_future` from a `std::future`

```
1 // sharedFutureFromFuture.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() << std::endl;
29
30 }
31
32 };
33
34 int main(){
35
36     std::cout << std::boolalpha << std::endl;
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() << std::endl;
45
46     // calculate the result in a separate thread
47     Div div;
48     std::thread divThread(div, std::move(divPromise), 20, 10);
49
50     std::cout << "divResult.valid(): " << divResult.valid() << std::endl;
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 << std::endl;
72
73 }
```

The first two calls of `divResult.valid()` on the `std::future` (lines 44 and 50) return true. This changes after the call `divResult.share()` in line 52 because this call transfers the shared state.



```
File Edit View Bookmarks Settings Help
rainer@linux:~> sharedFutureFromFuture
divResult.valid(): true
divResult.valid(): true
divResult.valid(): false
threadId(140132004022016): 20/10= 2
threadId(140131995629312): 20/10= 2
threadId(140131987236608): 20/10= 2
threadId(140132020807424): 20/10= 2
threadId(140132012414720): 20/10= 2
rainer@linux:~> ■
> rainer : bash
```

Creating a `std::shared_future`

Exceptions

If the callable used by `std::async` or by `std::packaged_task` throws an error, the exception is stored in the shared state. When the future `fut` then calls `fut.get()`, the exception is rethrown, and the future has to handle it.

A `std::promise` `prom` provides the same facility but has the method `prom.set_value(std::current_exception())` to set the exception as shared state.

Dividing a number by 0 is undefined behaviour. The function `executeDivision` displays the result of the calculation or the exception.

Returning an exception

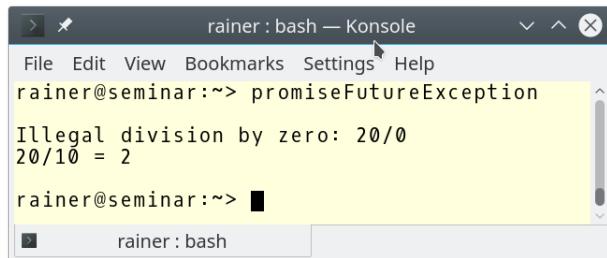
```
1 // promiseFutureException.cpp
2
3 #include <exception>
4 #include <future>
5 #include <iostream>
6 #include <thread>
7 #include <utility>
8
9 struct Div{
10     void operator()(std::promise<int>&& intPromise, int a, int b){
11         try{
12             if ( b==0 ){
13                 std::string 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() << std::endl;
35     }
36     catch (std::runtime_error& e){
37         std::cout << e.what() << std::endl;
38     }
39
40     divThread.join();
41 }
42 }
```

```

43 int main(){
44
45     std::cout << std::endl;
46
47     executeDivision(20, 0);
48     executeDivision(20, 10);
49
50     std::cout << std::endl;
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.



```

rainer : bash — Konssole
File Edit View Bookmarks Settings Help
rainer@seminar:~> promiseFutureException
Illegal division by zero: 20/0
20/10 = 2
rainer@seminar:~> 
```

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`⁴² 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`-method 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.

⁴²http://en.cppreference.com/w/cpp/error/current_exception

Notifications

Tasks are a save replacement for condition variables. If you use promises and futures to synchronise 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.

Condition variables versus tasks

Criteria	Condition Variables	Tasks
Multiple synchronisations	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 synchronise 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 functionality of a condition variable. If you use the condition variable for only one synchronisation, 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 synchronise two threads.

```

void waitingForWork(){
    std::cout << "Worker: Waiting for work." << std::endl;

    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
    doTheWork();
    std::cout << "Work done." << std::endl;
}

void setDataReady(){
    std::lock_guard<std::mutex> lck(mutex_);
    dataReady=true;
    std::cout << "Sender: Data is ready." << std::endl;
    condVar.notify_one();
}

```

The function `setDataReady` performs the notification part of the synchronisation - the function `waitForWork` the waiting part of the synchronisation.

Here is the same workflow with tasks.

Thread synchronisation with tasks

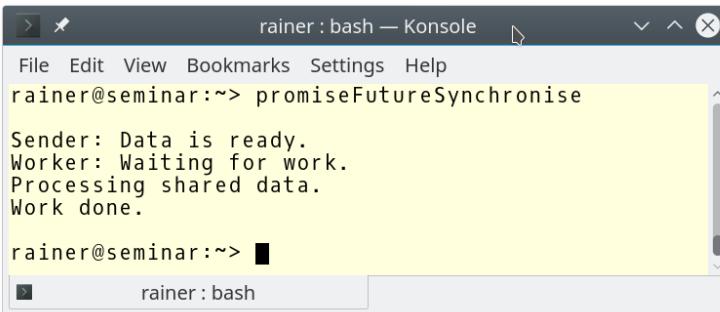
```
1 // promiseFutureSynchronise.cpp
2
3 #include <future>
4 #include <iostream>
5 #include <utility>
6
7
8 void doTheWork(){
9     std::cout << "Processing shared data." << std::endl;
10 }
11
12 void waitForWork(std::future<void>&& fut){
13
14     std::cout << "Worker: Waiting for work." << std::endl;
15     fut.wait();
16     doTheWork();
17     std::cout << "Work done." << std::endl;
18
19 }
20
21 void setDataReady(std::promise<void>&& prom){
22
23     std::cout << "Sender: Data is ready." << std::endl;
24     prom.set_value();
25
26 }
27
28 int main(){
29
30     std::cout << std::endl;
31
32     std::promise<void> sendReady;
33     auto fut = sendReady.get_future();
34
35     std::thread t1(waitForWork, std::move(fut));
36     std::thread t2(setDataReady, std::move(sendReady));
37
38     t1.join();
```

```
39     t2.join();
40
41     std::cout << std::endl;
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 structure and the output of the program matches the corresponding program in the section [condition variable](#).



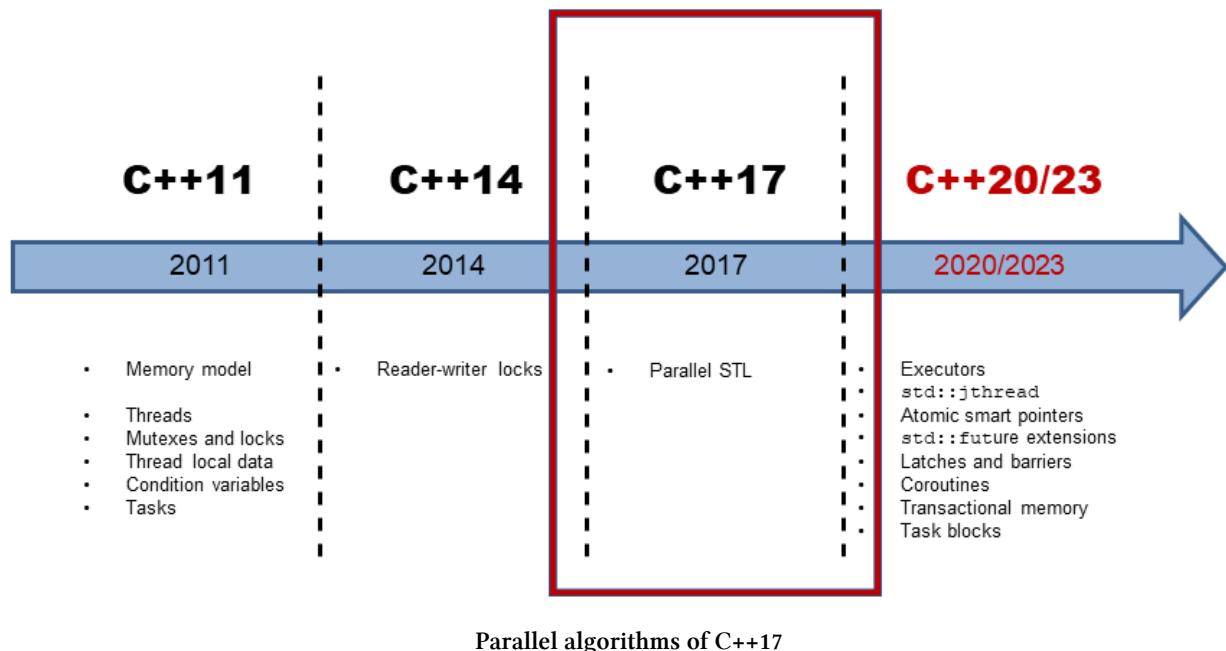
```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> promiseFutureSynchronise
Sender: Data is ready.
Worker: Waiting for work.
Processing shared data.
Work done.

rainer@seminar:~> █
█
```

std::promise and std::future as condition variable

Parallel Algorithms of the Standard Template Library

The Standard Template Library has more than 100 algorithms for searching, counting, and manipulation of ranges and their elements. With C++17, 69 of them are overloaded and eight new ones are added. The overloaded, and new algorithms can be invoked with a so-called execution policy.



By using an execution policy, you can specify whether the algorithm should run sequentially, in parallel, or in parallel with vectorisation. For using the execution policy, you have to include the header `<execution>`.

Execution Policies

The standard defines three execution policies:

- `std::execution::sequenced_policy`
- `std::execution::parallel_policy`
- `std::execution::parallel_unsequenced_policy`

The corresponding policy tag specifies whether a program should run sequentially, in parallel, or in parallel with vectorisation.

- `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 vectorised version with SIMD⁴³ (Single Instruction Multiple Data) extensions.

If an exception occurs during the usage of an algorithm with an execution policy, `std::terminate` is called. This is the difference between the invocation of an algorithm without an execution policy and an algorithm with a sequential `std::execution::seq` execution policy. The invocation of the algorithm without an execution policy just propagates the exception.

The usage of the execution policy `std::execution::par` or `std::execution::par_unseq` allows the algorithm to run parallel or parallel and vectorised. This is a permission and not a requirement.

The following code snippet applies all execution policies.

The execution policy

```

1 std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};
2
3 // standard sequential sort
4 std::sort(v.begin(), v.end());
5
6 // sequential execution
7 std::sort(std::execution::seq, v.begin(), v.end());
8
9 // permitting parallel execution
10 std::sort(std::execution::par, v.begin(), v.end());
11
12 // permitting parallel and vectorised execution
13 std::sort(std::execution::par_unseq, v.begin(), v.end());

```

⁴³<https://en.wikipedia.org/wiki/SIMD>

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 vectorised (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::parallel_unsequenced_policy`, or an implementation-defined execution policy type. Otherwise, `value` is equal to `false`.



Parallel and Vectorised Execution

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

The following example shows a simple loop for filling a vector.



Parallel and vectorised execution policy

```

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 key line in this small example. Thanks to the [compiler explorer](#)⁴⁴ we can have a closer look at the assembler instructions generated by clang 3.6.

Without Optimisation

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

```

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

Sequential execution

With maximum Optimisation

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

```

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

Vectorised execution

⁴⁴<https://godbolt.org/>



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

```
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

```
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 result in a deadlock.

Parallel and vectorised execution with a deadlock

```
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. This is undefined behaviour and gives most of the times a deadlock. You can avoid the deadlock by using an atomic.

Parallel and vectorised execution without a deadlock

```
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);`

An execution policy can parametrise 69 of the STL algorithms. Additionally, C++17 has 8 new algorithms.

Algorithms

Here are the 69 algorithms with parallelised 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.

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
<code>std::for_each</code>	Applies a unary callable to the range.
<code>std::for_each_n</code>	Applies a unary callable to the first n elements of the range.
<code>std::exclusive_scan</code>	Applies from the left a binary callable up to the ith (exclusive) element of the 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.

The new algorithms

Algorithm	Description
<code>std::inclusive_scan</code>	Similar to <code>std::partial_sum</code> ⁴⁵ Applies from the left a binary callable up to the <i>i</i> th (inclusive) element of the 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 <code>std::partial_sum</code>
<code>std::transform_exclusive_scan</code>	First applies a unary callable to the range and then applies <code>std::exclusive_scan</code> . If the binary callable is non- associative the result is non-deterministic.
<code>std::transform_inclusive_scan</code>	First applies a unary callable to the range and then applies <code>std::inclusive_scan</code> . If the binary callable is non- associative the result is non-deterministic.
<code>std::reduce</code>	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 <code>std::accumulate</code> ⁴⁶
<code>std::transform_reduce</code>	Applies first a unary callable to the range and then <code>std::reduce</code> . 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`. This 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 to the algorithms and then write about the functional heritage of these functions. 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 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; }`.

```
std::vector<int> v{1, 2, 3, 4};

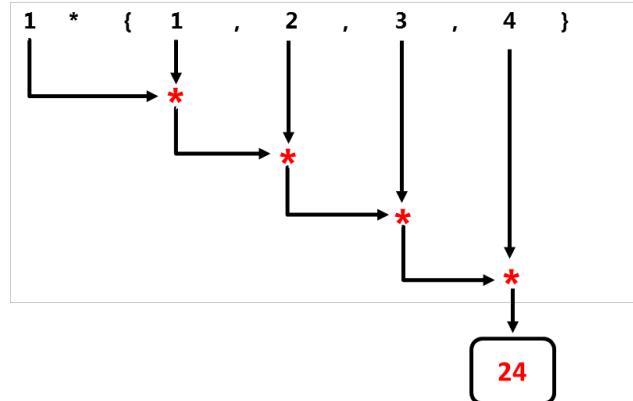
std::accumulate(v.begin(), v.end(), 1, [](int a, int b){ return a * b; });
std::reduce(std::execution::par, v.begin(), v.end(), 1,
           [](int a, int b){ return a * b; });
```

The two following graphs show the different fold strategies of `std::accumulate` and `std::reduce`.

⁴⁵http://en.cppreference.com/w/cpp/algorithm/partial_sum

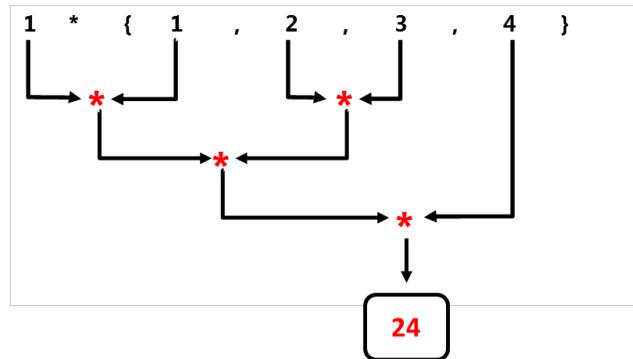
⁴⁶<http://en.cppreference.com/w/cpp/algorithm/accumulate>

`std::accumulate` starts at the left and successively applies the binary operator.



Fold strategy of `std::accumulate`

To 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 the commutativity, the intermediate results can be computed in an arbitrary order.



Available Implementations

Before I show you the source code of the example program, I have to make a general remark. As far as I know, at the time this book is being updated (September 2018) there is no fully standard-compliant implementation of the parallel STL available. MSVC 17.8 added support for about 30 algorithms.

Parallel Algorithms with MSVC 17.8	std::adjacent_find	std::all_of
std::adjacent_difference		
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		

I used the HPX implementation to produce the output. The [HPX \(High-Performance ParalleX\)⁴⁷](#) is a framework that is a general purpose C++ runtime system for parallel and distributed applications of any scale. HPX has already implemented the parallel STL in a different namespace.

For completeness, here are further (partial) implementations of the parallel STL:

- [Intel⁴⁸](#)
- [Thibaut Lutz⁴⁹](#)
- [Nvidia \(thrust\)⁵⁰](#)
- [Codeplay⁵¹](#)

⁴⁷<http://stellar.cct.lsu.edu/projects/hpx/>

⁴⁸<https://software.intel.com/en-us/get-started-with-psl>

⁴⁹<https://github.com/t-lutz/ParallelSTL>

⁵⁰https://thrust.github.io/doc/group__execution__policies.html

⁵¹<https://github.com/KhronosGroup/SyclParallelSTL>

The new algorithms

```
1 // newAlgorithm.cpp
2
3 #include <algorithm>
4 #include <execution>
5 #include <numeric>
6 #include <iostream>
7 #include <string>
8 #include <vector>
9
10
11 int main(){
12
13     std::cout << std::endl;
14
15     // for_each_n
16
17     std::vector<int> intVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
18     std::for_each_n(std::execution::par,
19                     intVec.begin(), 5, [](int& arg){ arg *= arg; });
20
21     std::cout << "for_each_n: ";
22     for (auto v: intVec) std::cout << v << " ";
23     std::cout << "\n\n";
24
25     // exclusive_scan and inclusive_scan
26     std::vector<int> resVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
27     std::exclusive_scan(std::execution::par,
28                         resVec.begin(), resVec.end(), resVec.begin(), 1,
29                         [](int fir, int sec){ return fir * sec; });
30
31     std::cout << "exclusive_scan: ";
32     for (auto v: resVec) std::cout << v << " ";
33     std::cout << std::endl;
34
35     std::vector<int> resVec2{1, 2, 3, 4, 5, 6, 7, 8, 9};
36
37     std::inclusive_scan(std::execution::par,
38                         resVec2.begin(), resVec2.end(), resVec2.begin(),
39                         [] (int fir, int sec){ return fir * sec; }, 1);
40
41     std::cout << "inclusive_scan: ";
42     for (auto v: resVec2) std::cout << v << " ";
```

```
43     std::cout << "\n\n";
44
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                             [] (int fir, int sec){ return fir + sec; },
52                             [] (int arg){ return arg *= arg; });
53
54 std::cout << "transform_exclusive_scan: ";
55 for (auto v: resVec4) std::cout << v << " ";
56 std::cout << std::endl;
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(), 0,
64                             [] (auto fir, auto sec){ return fir + sec; },
65                             [] (auto s){ return s.length(); });
66
67 std::cout << "transform_inclusive_scan: ";
68 for (auto v: resVec5) std::cout << v << " ";
69 std::cout << "\n\n";
70
71 // reduce and transform_reduce
72 std::vector<std::string> strVec2{"Only", "for", "testing", "purpose"};
73
74 std::string res = std::reduce(std::execution::par,
75                             strVec2.begin() + 1, strVec2.end(), strVec2[0],
76                             [] (auto fir, auto sec){ return fir + ":" + sec; });
77
78 std::cout << "reduce: " << res << std::endl;
79
80 std::size_t res7 = std::transform_reduce(std::execution::par,
81                             strVec2.begin(), strVec2.end(),
82                             [] (std::string s){ return s.length(); },
83                             0, [] (std::size_t a, std::size_t b){ return a + b; });
84
85
```

```

86     std::cout << "transform_reduce: " << res7 << std::endl;
87
88     std::cout << std::endl;
89
90 }
```

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 powers of 2.

`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 quite 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.

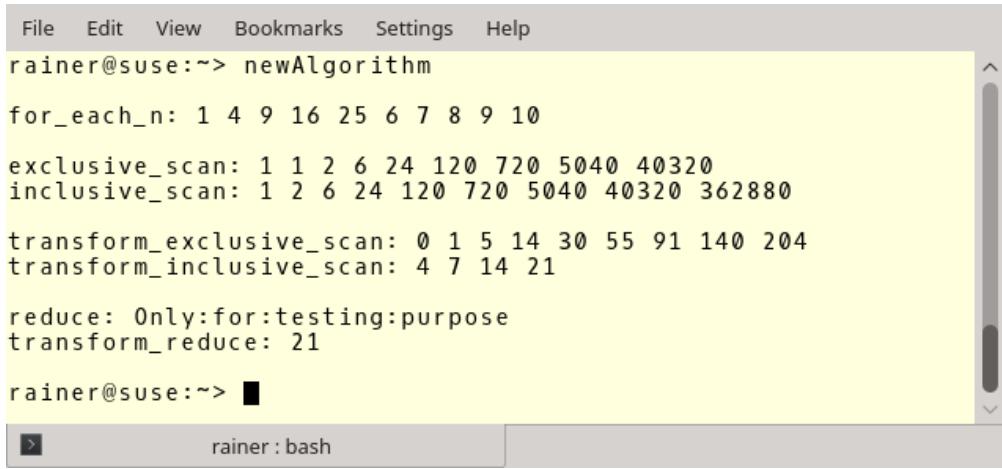


transform_reduce becomes map_reduce

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 understand it. `std::transform_reduce` is the well-known parallel [MapReduce⁵²](#) 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.

⁵²<https://en.wikipedia.org/wiki/MapReduce>



```

File Edit View Bookmarks Settings Help
rainer@suse:~> newAlgorithm
for_each_n: 1 4 9 16 25 6 7 8 9 10
exclusive_scan: 1 1 2 6 24 120 720 5040 40320
inclusive_scan: 1 2 6 24 120 720 5040 40320 362880
transform_exclusive_scan: 0 1 5 14 30 55 91 140 204
transform_inclusive_scan: 4 7 14 21
reduce: Only:for:testing:purpose
transform_reduce: 21
rainer@suse:~> █

```

The new algorithms

More overloads

All C++ variants of reduce and scan 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.

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` applies a binary operation to a list and reduces the list to a value. `foldl` needs in contrast to `foldl1` an 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> let ints = [1..9] (1)
Prelude> let strings= ["Only","for","testing","purpose"] (2)
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> ■

```

(1) and (2) define a list of integers and a list of strings. In (3), I apply the lambda function ($\lambda a \rightarrow a * a$) to the list of integers. (4) and (5) are more sophisticated. The expression (4) multiplies (*) all pairs of integers starting with the 1 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 case 0. Now expression (8) should be readable; it successively joins two strings with the ":" character.

Case Studies

After providing the theory on the [memory model](#) and the [multithreading interface](#), I now apply the theory in practice and provide you a few performance numbers.



The Reference PCs

I present the performance numbers of the programs with my Linux desktop (GCC 4.8.3) and my Windows laptop (cl.exe 19.00.23506). I always use full optimisation and create 64-bit executables. The Linux PC has four cores, while the Windows PC has two cores. Here are the details of both compilers.

```
rainer@linux:~> g++ -v
rainer@linux:~>
Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/usr/lib64/gcc/x86_64-suse-linux/4.8/lto-wrapper
Target: x86_64-suse-linux
Configured with: ../configure --prefix=/usr --infodir=/usr/share/info --mandir=/usr/share/man --libdir=/usr/lib64 --libexecdir=/usr/lib64 --enable-languages=c,c++,objc,fortran,obj-c++,java,ada --enable-checking=release --with-gxx-include-dir=/usr/include/c++/4.8 --enable-ssp --disable-libssp --disable-plugin --with-bugurl=http://bugs.opensuse.org/ --with-pkgversion='SUSE Linux' --disable-libgcj --disable-libmudflap --with-slibdir=/lib64 --with-system-zlib --enable_cxa_atexit --enable-libstdcxx-allocator=new --disable-libstdcxx-pch --enable-version-specific-runtime-libs --enable-linker-build-id --enable-linux-futex --program-suffix=-4.8 --without-system-libunwind --with-arch-32=i586 --with-tune=generic --build=x86_64-suse-linux --host=x86_64-suse-linux
Thread model: posix
gcc version 4.8.3 20140627 [gcc-4_8-branch revision 212064] (SUSE Linux)
rainer@linux:~>
```

Characteristic of the Linux PC

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0\cl.exe
Microsoft (R) C/C++-Optimierungscompiler Version 19.00.23506 für x86
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

Syntax: cl [ Option... ] Dateiname... [ /link Linkeroption... ]

C:\Program Files (x86)\Microsoft Visual Studio 14.0>
```

Characteristic of the Windows PC

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.

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]([https://en.wikipedia.org/wiki/Uniform_distribution_\(continuous\)](https://en.wikipedia.org/wiki/Uniform_distribution_(continuous))) 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.

Single Threaded addition of a Vector

The straightforward strategy is it to add the numbers in a range-based for loop.

Range-based for Loop

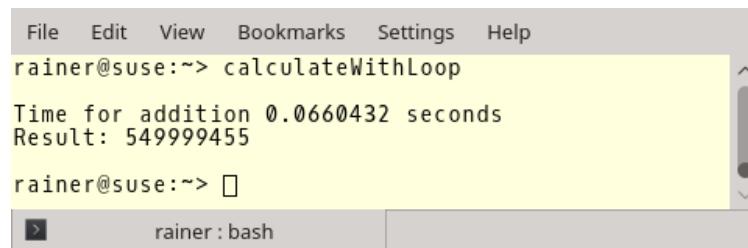
The summation takes place in line 27.

Summation of a vector in a range-based for loop

```
1 // calculateWithLoop.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <random>
6 #include <vector>
7
8 constexpr long long size = 100000000;
9
10 int main(){
11
12     std::cout << std::endl;
13
14     std::vector<int> randValues;
15     randValues.reserve(size);
16
17     // random values
18     std::random_device seed;
19     std::mt19937 engine(seed());
20     std::uniform_int_distribution<> uniformDist(1, 10);
21     for (long long i = 0 ; i < size ; ++i)
22         randValues.push_back(uniformDist(engine));
23 }
```

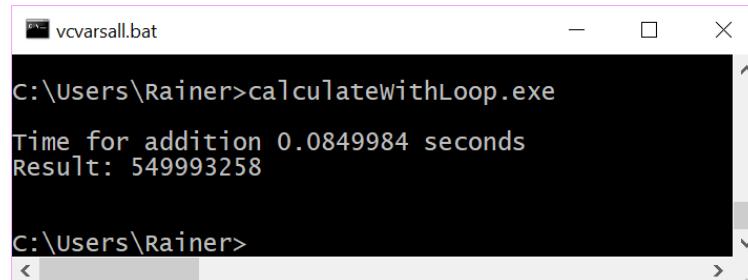
```
24 const auto sta = std::chrono::steady_clock::now();
25
26 unsigned long long sum = {};
27 for (auto n: randValues) sum += n;
28
29 const std::chrono::duration<double> dur =
30     std::chrono::steady_clock::now() - sta;
31
32 std::cout << "Time for mySumition " << dur.count()
33     << " seconds" << std::endl;
34 std::cout << "Result: " << sum << std::endl;
35
36 std::cout << std::endl;
37
38 }
```

How fast are my computers?



```
File Edit View Bookmarks Settings Help
rainer@suse:~> calculateWithLoop
Time for addition 0.0660432 seconds
Result: 549999455
rainer@suse:~> □
rainer : bash
```

Explicit summation on Linux



```
vcvarsall.bat
C:\Users\Rainer>calculateWithLoop.exe
Time for addition 0.0849984 seconds
Result: 549993258
C:\Users\Rainer>
```

Explicit summation on Windows

You should not use loops explicitly. Most of the time you can use an algorithm from the Standard Template Library.

Summation with `std::accumulate`

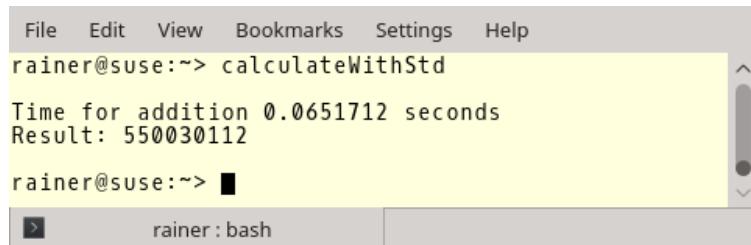
`std::accumulate` is the right way to calculate the sum of a vector. For the sake of simplicity, I show the application of `std::accumulate`. The entire source file can be found in the resources for this book.

Summation of a vector with std::accumulate*// calculateWithStd.cpp**...*

```
const unsigned long long sum = std::accumulate(randValues.begin(),
                                              randValues.end(), 0);
```

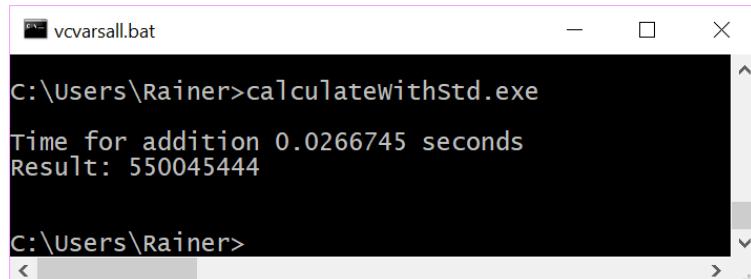
...

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, making its performance much better than Linux.



```
rainer@suse:~> calculateWithStd
Time for addition 0.0651712 seconds
Result: 550030112
rainer@suse:~>
```

Summation with std::accumulate on Linux



```
vcvarsall.bat
C:\Users\Rainer>calculatewithstd.exe
Time for addition 0.0266745 seconds
Result: 550045444
C:\Users\Rainer>
```

Summation with std::accumulate 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.

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 synchronisation 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.

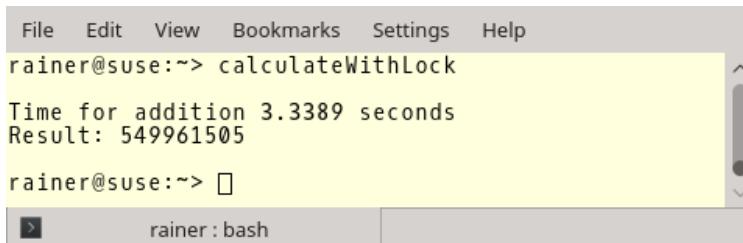
Summation of a vector by using a lock for the summation variable

```
// calculateWithLock.cpp

...
std::mutex myMutex;

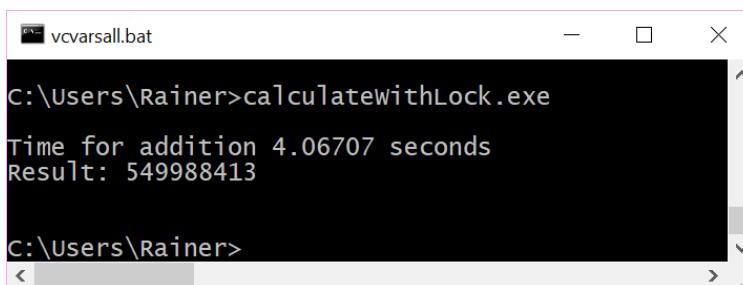
for (auto i: randValues){
    std::lock_guard<std::mutex> myLockGuard(myMutex);
    sum += i;
}
```

The execution times are as expected: the access to the protected variable `sum` is slower.



```
File Edit View Bookmarks Settings Help
rainer@suse:~> calculateWithLock
Time for addition 3.3389 seconds
Result: 549961505
rainer@suse:~> 
```

Single-threaded summation with `std::accumulate` on Linux using a lock



```
vcvarsall.bat
C:\Users\Rainer>calculateWithLock.exe
Time for addition 4.06707 seconds
Result: 549988413
C:\Users\Rainer> 
```

Single-threaded summation with `std::accumulate` 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.

Protection with Atomics

Accordingly, I have the same questions for atomics that I had for locks.

1. How expensive is the synchronisation 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?

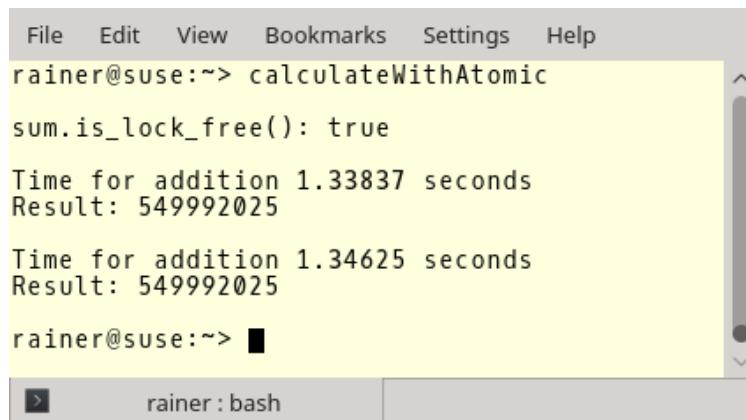
Summation of a vector by using an atomic as summation variable

```
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 << std::endl;
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() << std::endl;
29     std::cout << std::endl;
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     std::cout << "Time for addition " << dur.count()
```

```
39         << " seconds" << std::endl;
40     std::cout << "Result: " << sum << std::endl;
41
42     std::cout << std::endl;
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" << std::endl;
52     std::cout << "Result: " << sum << std::endl;
53
54     std::cout << std::endl;
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 45 the method `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.



```
File Edit View Bookmarks Settings Help
rainer@suse:~> calculateWithAtomic
sum.is_lock_free(): true
Time for addition 1.33837 seconds
Result: 549992025
Time for addition 1.34625 seconds
Result: 549992025
rainer@suse:~> █
```

Summation with an atomic on Linux

Summation with an atomic on Windows

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 synchronisation.
2. Atomics are 2 - 3 times faster on Linux and Windows than locks.
3. `std::accumulate` seems to be highly optimised on Windows.

Before we look at the multithreaded scenarios, here is a table summarising 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

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 synchronisation 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`?

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`

```
1 // synchronisationWithLock.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <mutex>
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 std::mutex myMutex;
19
20 void sumUp(unsigned long long& sum, const std::vector<int>& val,
21             unsigned long long beg, unsigned long long end){
22     for (auto it = beg; it < end; ++it){
23         std::lock_guard<std::mutex> myLock(myMutex);
24         sum += val[it];
25     }
26 }
27
28 int main(){
29
30     std::cout << std::endl;
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     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" << std::endl;
56     std::cout << "Result: " << sum << std::endl;
57
58     std::cout << std::endl;
59
60 }
```

The program is easy to explain. The function `sumUp` (lines 20 - 26) is the work package that each thread executes. `sumUp` gets the summation variable `sum` and the `std::vector val` by reference. `beg` and `end` specifies 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.

```

File Edit View Bookmarks Settings Help
rainer@suse:~> synchronisationWithLock
Time for addition 20.8111 seconds
Result: 549996948
rainer@suse:~> █

```

Summation with a shared variable on Linux

```
C:\Users\Rainer>synchronisationwithLock.exe
Time for addition 6.22507 seconds
Result: 550020076
```

Summation with a shared variable on Windows

The bottleneck of the program is the shared variable `sum` because a `std::lock_guard` heavily synchronises 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 full examples, please see the resources for this book.

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

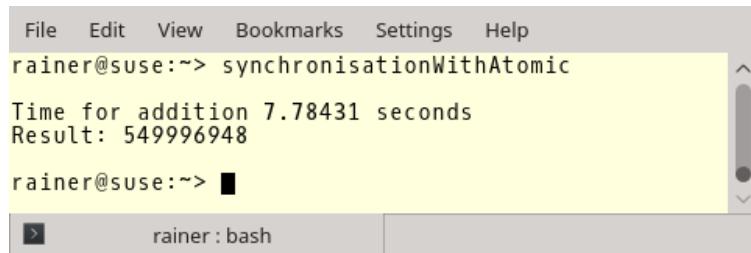
```
// synchronisationWithAtomic.cpp
```

...

```
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 quite weird on my Windows Laptop. The synchronisation with `std::lock_guard` is more than twice as fast as the atomic version.

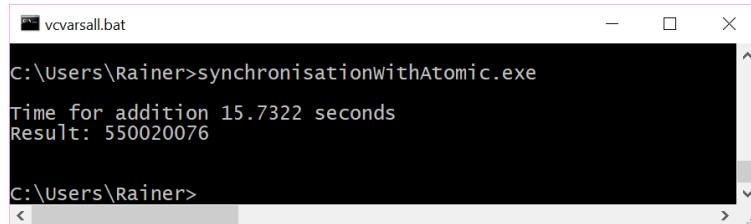


```

File Edit View Bookmarks Settings Help
rainer@suse:~> synchronisationWithAtomic
Time for addition 7.78431 seconds
Result: 549996948
rainer@suse:~> ■

```

Summation with an atomic on Linux



```

vcvarsall.bat
C:\Users\Rainer>synchronisationWithAtomic.exe
Time for addition 15.7322 seconds
Result: 550020076
C:\Users\Rainer>

```

Summation with an atomic on Windows

In addition to using the `+=` operator on an atomic, you can use the `fetch_add` method. Let's try it out.

Using the method `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 method `fetch_add`

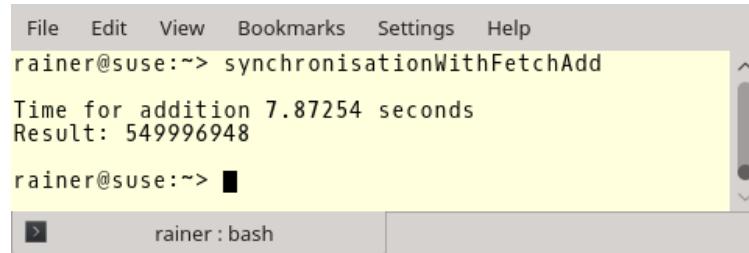
```

// synchronisationWithFetchAdd.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]);
    }
}

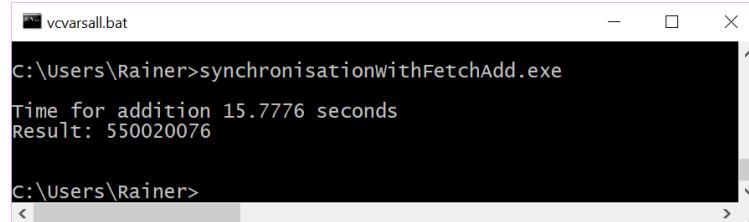
```

Now we have similar performance as the previous example; there is little difference between the operator `+=` and `fetch_add`.



```
File Edit View Bookmarks Settings Help
rainer@suse:~> synchronisationWithFetchAdd
Time for addition 7.87254 seconds
Result: 549996948
rainer@suse:~> ■
```

Summation with an atomic on Linux



```
vcvarsall.bat
C:\Users\Rainer>synchronisationWithFetchAdd.exe
Time for addition 15.7776 seconds
Result: 550020076
C:\Users\Rainer>
```

Summation with an atomic on Windows

Although there is no performance difference between the `+=` operation and the `fetch_add` method on an atomic, `fetch_add` has an advantage; it allows me to weaken the memory-ordering explicitly and to apply relaxed semantic.

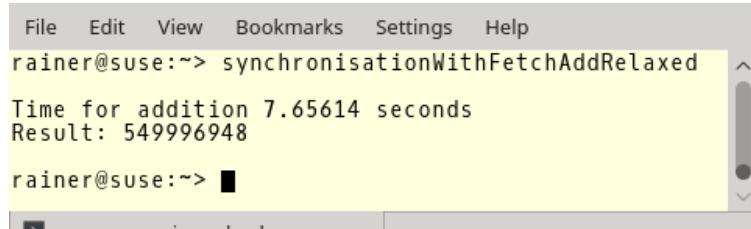
Using the method `fetch_add` with relaxed semantic

```
1 // synchronisationWithFetchAddRelaxed.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 behaviour for atomics is [sequential consistency](#). This statement is true for the addition and assignment of an atomic and of course for the `fetch_add` method, but we can optimise 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 optimisation.

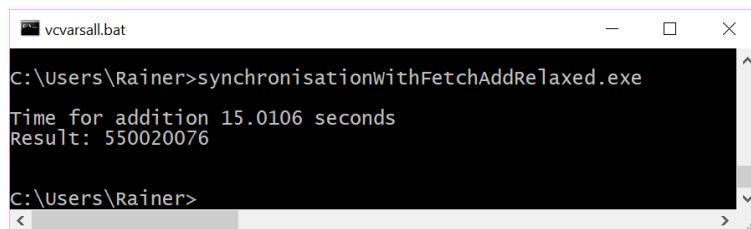
The relaxed semantic is fine in this use-case because we have two guarantees: each addition with `fetch_add` takes place atomically, and the threads synchronise with the join calls.

Because of the weakest memory model, we have the best performance.



```
File Edit View Bookmarks Settings Help
rainer@suse:~> synchronisationWithFetchAddRelaxed
Time for addition 7.65614 seconds
Result: 549996948
rainer@suse:~> ■
```

Summation with a relaxed atomic on Linux



```
vcvarsall.bat
C:\Users\Rainer>synchronisationWithFetchAddRelaxed.exe
Time for addition 15.0106 seconds
Result: 550020076
C:\Users\Rainer>
```

Summation with a relaxed atomic on Windows

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 the help of four threads 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 minimise the synchronisation between the threads.

Thread-Local Summation

There are different ways to minimise the synchronisation. I can use local variables, [thread-local data](#), and [tasks](#).

Using a Local Variable

Since each thread can use a local summation variable, it can do its job without synchronisation. The synchronisation is only necessary, to sum up the local variables. The summation of the local variables is the critical section that must be protected. This can be done in various ways. A quick remark: since only four additions take place, it doesn't matter from a performance perspective which synchronisation I use. I use a `std::lock_guard`, an atomic with sequential consistency and relaxed semantic for the summation.

`std::lock_guard`

Summation with minimal synchronisation using a `std::lock_guard`

```
1 // localVariable.cpp
2
3 #include <mutex>
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 std::mutex myMutex;
19
20 void sumUp(unsigned long long& sum, const std::vector<int>& val,
21             unsigned long long beg, unsigned long long end){
22     unsigned long long tmpSum{};
23     for (auto i = beg; i < end; ++i){
24         tmpSum += val[i];
25     }
26     std::lock_guard<std::mutex> lockGuard(myMutex);
27     sum += tmpSum;
28 }
29
30 int main(){
```

```
32     std::cout << std::endl;
33
34     std::vector<int> randValues;
35     randValues.reserve(size);
36
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" << std::endl;
61     std::cout << "Result: " << sum << std::endl;
62
63     std::cout << std::endl;
64
65 }
```

Lines 26 and 27 are the interesting lines. These are the lines where the local summation result `tmpSum` is added to the global summation variable `sum`.

```
File Edit View Bookmarks Settings Help
rainer@suse:~> localVariable
Time for addition 0.0284271 seconds
Result: 549996948
rainer@suse:~> ■
```

Summation with a local variable on Linux

```
vcvarsall.bat
c:\users\Rainer>localvariable.exe
Time for addition 0.0881557 seconds
Result: 550020076
c:\users\Rainer>
```

Summation with a local variable on Windows

In the next two variations using a local variable, only the function `sumUp` changes; therefore, I just display the function. For the entire program, please refer to the source files.

Using an Atomic Variable with Sequential Consistency

Let's replace the non-atomic global summation variable `sum` by an atomic.

Summation of a vector with minimal synchronisation and an atomic

```
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.

```
File Edit View Bookmarks Settings Help
rainer@suse:~> localVariableAtomic
Time for addition 0.0286921 seconds
Result: 549996948
rainer@suse:~> ■
```

Summation with a local variable on Linux

```
vcvarsall.bat
C:\Users\Rainer>localvariableAtomic.exe
Time for addition 0.100168 seconds
Result: 550020076
C:\Users\Rainer>
```

Summation with a local variable on Windows

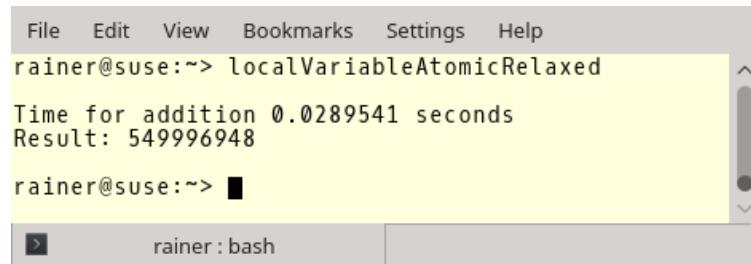
Using an Atomic Variable with Relaxed Semantic

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 take place and are atomic.

Summation of a vector with minimal synchronisation and an atomic using relaxed semantic

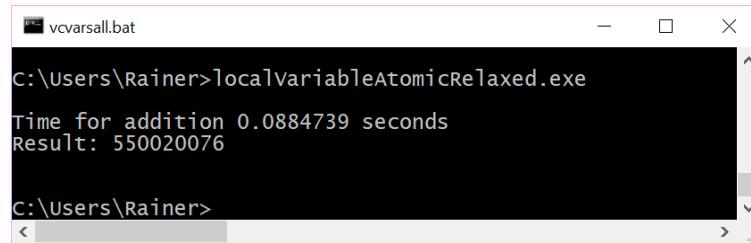
```
1 // localVariableAtomicRelaxed.cpp
2
3 void sumUp(std::atomic<unsigned long long>& sum, const std::vector<int>& val,
4             unsigned long long beg, unsigned long long end){
5     unsigned int long long tmpSum{};
6     for (auto i = beg; i < end; ++i){
7         tmpSum += val[i];
8     }
9     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.



```
File Edit View Bookmarks Settings Help
rainer@suse:~> localVariableAtomicRelaxed
Time for addition 0.0289541 seconds
Result: 549996948
rainer@suse:~> ■
```

Summation with a local variable on Linux



```
vcvarsall.bat
c:\users\rainer>localvariableAtomicRelaxed.exe
Time for addition 0.0884739 seconds
Result: 550020076
C:\users\rainer>
```

Summation with a local variable on Windows

Thread-local data is a particular kind of local data. Its 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.

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 synchronisation using thread-local data

```
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 << std::endl;
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" << std::endl;
58     std::cout << "Result: " << sum << std::endl;
59
60     std::cout << std::endl;
```

```
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.

```
File Edit View Bookmarks Settings Help  
rainer@suse:~> threadLocalSummation  
Time for addition 0.0369362 seconds  
Result: 549996948  
rainer@suse:~> █  
rainer : bash
```

Summation with a thread-local variable on Linux

```
vcvarsall.bat  
C:\Users\Rainer>threadLocalSummation.exe  
Time for addition 0.202901 seconds  
Result: 550020076  
C:\Users\Rainer>
```

Summation with a thread-local variable on Windows

In the last scenario, I use tasks.

Using Tasks

Using tasks, we can do the whole job without explicit synchronisation. 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 synchronisation using tasks

```
1 // tasksSummation.cpp  
2  
3 #include <chrono>  
4 #include <future>  
5 #include <iostream>  
6 #include <random>  
7 #include <thread>  
8 #include <utility>  
9 #include <vector>  
10
```

```
11 constexpr long long size = 1000000000;
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 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 << std::endl;
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" << std::endl;
61     std::cout << "Result: " << sum << std::endl;
62
63     t1.join();
64     t2.join();
65     t3.join();
66     t4.join();
67
68     std::cout << std::endl;
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 result of the summation by using the blocking `get` call.

```

File Edit View Bookmarks Settings Help
rainer@suse:~> taskssummation
Time for addition 0.0323008 seconds
Result: 549996948
rainer@suse:~> █

```

Summation with a local variable on Linux

```

vcvarsall.bat
C:\Users\Rainer>taskssummation.exe
Time for addition 0.100989 seconds
Result: 550020076
C:\Users\Rainer> █

```

Summation with a local variable on Windows

To conclude this section, there is an overview of all performance numbers.

All Numbers for the Thread-Local Summation

It does not make a big difference whether I use local variables or tasks for the calculation of the partial sum or if I use various synchronisation 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 optimised for 4 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. In the optimal case, I would expect a fourfold performance improvement because there is hardly a synchronisation necessary between the threads. What is the reason?

Summation of a Vector: The Conclusion

Single Threaded

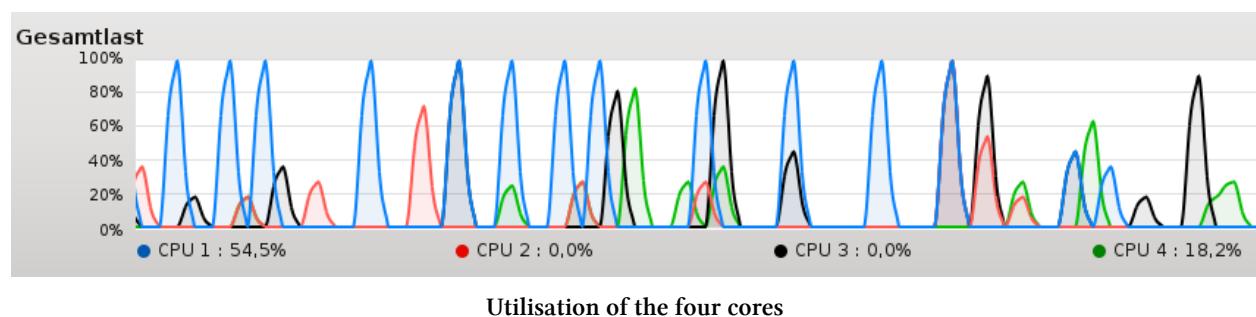
The range-based for loop and the STL algorithm `std::accumulate` are in the same performance range. In the optimised version, the compiler uses for the summation case the optimised version vectorised SIMD⁵³ instruction (SSE or AVX). Therefore, the loop counter is increased by 4 (SSE) or 8 (AVX).

Multithreading with a Shared Variable

The usage of a shared variable for the summation variable makes one point clear: synchronisation is very expensive 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, minimising expensive synchronisation has to be our first goal.

Thread-local Summation

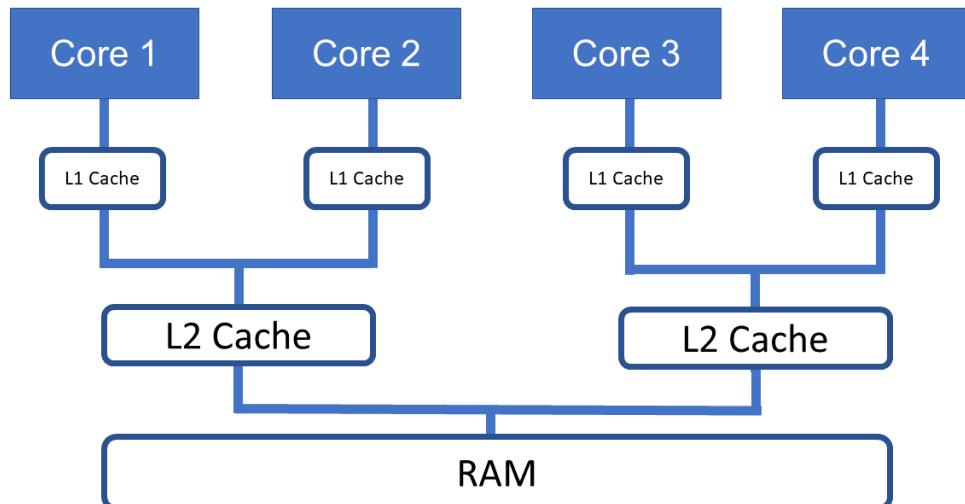
The thread-local summation is only two times faster than the single-threaded range-based for loop or `std::accumulate`. That holds even though each of the four threads can work independently. That surprised me because I was expecting a nearly fourfold improvement. What surprised me even more, was that my four cores are not fully utilised.



The reason is simple; the cores can't get the data fast enough from memory. The execution is **memory bound**⁵⁴. Or to say it the other way around, the memory slows down the cores. The following pictures show the bottleneck memory.

⁵³<https://en.wikipedia.org/wiki/SIMD>

⁵⁴https://en.wikipedia.org/wiki/Memory_bound_function



The bottleneck memory

The [Roofline model⁵⁵](#) is an intuitive performance model to provide performance estimates of applications running on multi-core, or many-core architecture. The model depends on the peak performance, peak bandwidth, and arithmetic intensity of the architecture.

⁵⁵https://en.wikipedia.org/wiki/Roofline_model

Thread-Safe Initialisation of a Singleton

Before I start with this case study, let me emphasise: I am not advocating the use of the singleton pattern. Therefore, let me start this chapter to the thread-safe initialisation 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 initialised in a thread-safe way. The singleton pattern has a few serious disadvantages. Let me give you a few ideas:

- A singleton is a global variable in disguise. Therefore, it makes it quite difficult to test your function because it depends on the global state.
- Typically, you use a singleton in a function by invoking the static method `MySingleton::getInstance()`. This means the interface of a function does not tell you that you use a singleton inside. You hide the dependency to 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 fiasco⁵⁶](#) because there is no guarantee which static is initialised 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 which you don't need any more, 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 pattern⁵⁷](#).

I want to start my discussion of the thread-safe initialisation of the singleton with a short detour.

Double-Checked Locking Pattern

The [double-checked locking⁵⁸](#) pattern is the classical way to initialise a singleton in a thread-safe way. What sounds like established best practice or as a pattern, is more a kind of [anti-pattern⁵⁹](#). 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 the creation of a singleton is an atomic operation; therefore, a solution that seems to be thread-safe is not thread-safe.

⁵⁶<https://isocpp.org/wiki/faq/ctors>

⁵⁷https://en.wikipedia.org/wiki/Singleton_pattern

⁵⁸<http://www.cs.wustl.edu/~schmidt/PDF/DC-Locking.pdf>

⁵⁹<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 initialisation of the singleton with a lock.

Thread-safe initialisation 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 initialises 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. Initialise the `MySingleton` object
3. Let `instance` refer to the fully initialised `MySingleton` object

The issue is that the C++ runtime provides no guarantee that the steps are performed in that sequence. For example, it is possible that 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-initialised 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-initialised singleton, and the program behaviour is undefined.

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 initialises 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 optimisation. 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` method is not thread-safe with the C++03 standard.

Single-threaded singleton implementation

```
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() << std::endl;
36
37 }
```

As the reference implementation, I use the so-called Meyers Singleton, named after [Scott Meyers](#)⁶⁰. The elegance of this implementation is that the singleton object in line 11 is a static variable with a

⁶⁰https://en.wikipedia.org/wiki/Scott_Meyers

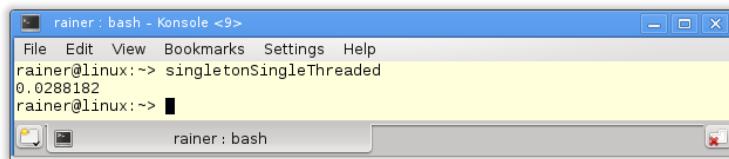
block scope; therefore, `instance` is initialised only once. This initialisation happens when the static method `getInstance` (lines 10 - 14) is executed the first time.



The volatile Variable dummy

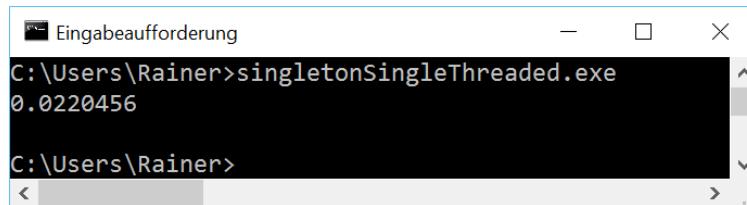
When I compiled the program with maximum optimisation, the compiler removed the call `MySingleton::getInstance()` in line 30 because the call has no effect; therefore, I got very fast execution, but wrong performance numbers. By using the `volatile` variable `dummy` (line 12), the compiler is not allowed to optimise away the `MySingleton::getInstance()` call in line 30.

Here are the reference numbers for the single threaded use-case.



```
rainer : bash - Konsole <9>
File Edit View Bookmarks Settings Help
rainer@linux:~/singletonSingleThreaded
0.0288182
rainer@linux:~>
```

Meyers Singelton on Linux (single theaded)



```
Eingabeaufforderung
C:\Users\Rainer>singletonSingleThreaded.exe
0.0220456
C:\Users\Rainer>
```

Meyers Singleton on Windows (single theaded)

The beauty of the Meyers Singleton is that it becomes thread-safe with C++11.

Thread-Safe Meyers Singleton

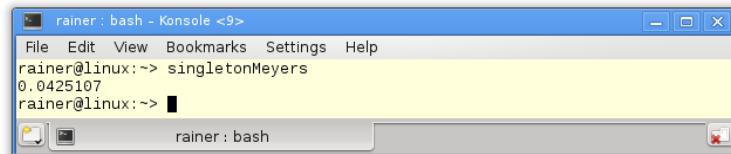
The C++11 standard guarantees that [static variables with block scope](#) are initialised in a thread-safe way. The Meyers Singleton uses a static variable with block scope, so we are done. The only work that is left to do is to rewrite the previously used [classical Meyers Singleton](#) for the multithreading use-case.

Meyers-Singleton in the multithreading use-case

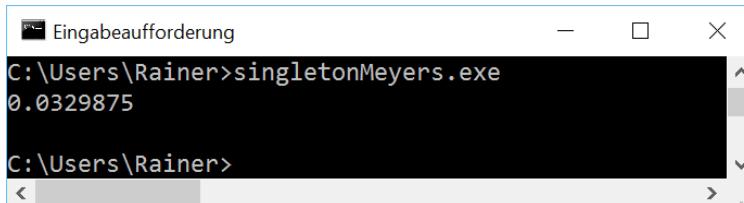
```
1 // singletonMeyers.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <future>
6
7 constexpr auto tenMill = 10000000;
8
9 class MySingleton{
10 public:
11     static MySingleton& getInstance(){
12         static MySingleton instance;
13         volatile int dummy{};
14         return instance;
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() << std::endl;
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.

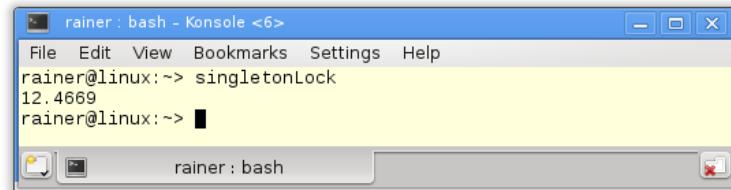
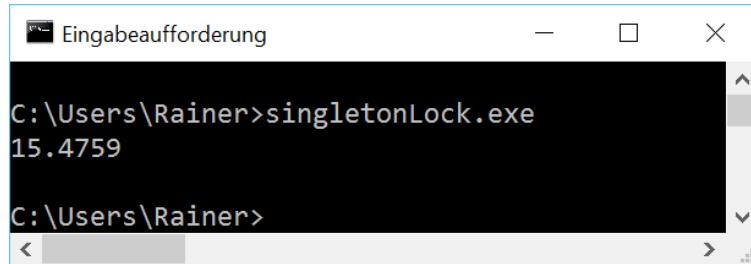
`std::lock_guard`

The mutex wrapped in a `std::lock_guard` guarantees that the singleton is initialised in a thread-safe way.

A thread-safe singleton using a lock

```
1 // singletonLock.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <future>
6 #include <mutex>
7
8 constexpr auto tenMill = 10000000;
9
10 std::mutex myMutex;
11
12 class MySingleton{
13 public:
14     static MySingleton& getInstance(){
15         std::lock_guard<std::mutex> myLock(myMutex);
16         if (!instance){
17             instance= new MySingleton();
18         }
19         volatile int dummy{};
20         return *instance;
21     }
22 private:
23     MySingleton() = default;
24     ~MySingleton() = default;
25     MySingleton(const MySingleton&) = delete;
26     MySingleton& operator=(const MySingleton&) = delete;
27
28     static MySingleton* instance;
29 };
30
31
32 MySingleton* MySingleton::instance = nullptr;
33
34 ...
```

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` in combination with the `std::once_flag`.

`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`

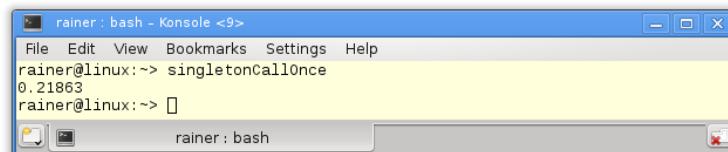
```
1 // singletonCallOnce.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <future>
6 #include <mutex>
7 #include <thread>
8
9 constexpr auto tenMill = 10000000;
10
11 class MySingleton{
12 public:
13     static MySingleton& getInstance(){
14         std::call_once(initInstanceFlag, &MySingleton::initSingleton);
15         volatile int dummy{};
16         return *instance;
17     }
18 private:
```

```

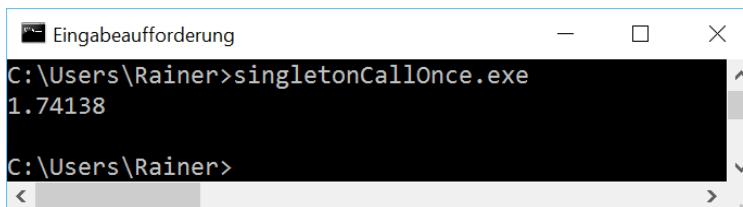
19  MySingleton() = default;
20  ~MySingleton() = default;
21  MySingleton(const MySingleton&) = delete;
22  MySingleton& operator=(const MySingleton&) = delete;
23
24  static MySingleton* instance;
25  static std::once_flag initInstanceFlag;
26
27  static void initSingleton(){
28      instance= new 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.

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.

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

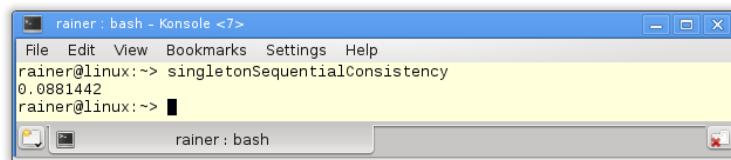
```
1 // singletonSequentialConsistency.cpp
2
3 #include <atomic>
4 #include <iostream>
5 #include <future>
6 #include <mutex>
7 #include <thread>
8
9 constexpr auto tenMill = 10000000;
10
11 class MySingleton{
12 public:
13     static MySingleton* getInstance(){
14         MySingleton* sin = instance.load();
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;
```

```

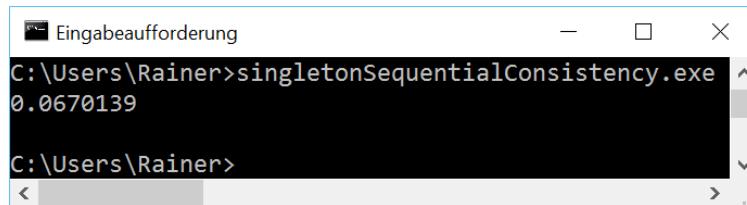
38 std::mutex MySingleton::myMutex;
39
40 ...

```

In contrast to the double-checked locking pattern, I now have the 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 optimise the program even more.

Acquire-release semantic

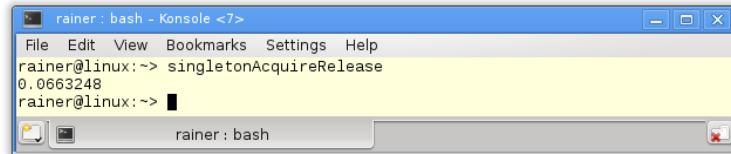
Let's have a closer look at the previous thread-safe implementation of the singleton pattern using atomics. The loading (or reading) of the singleton in line 14 is an acquire operation, the storing (or writing) in line 20 a release operation. Both operations take place on the same atomic. Therefore sequential consistency is overkill. The C++11 standard guarantees that a release operation synchronises 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, and 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.

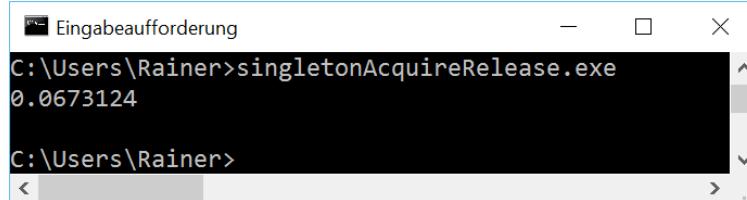
A thread-safe singleton using atomics with acquire-release semantic

```
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 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 [ARMv7⁶¹](#) or [PowerPC⁶²](#) architecture. You can read the details Jeff Preshings blog [Preshing on Programming⁶³](#).

At the end here is an overview of all performance numbers.

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 synchronisation 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 Consistency	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 threads. That means that we get optimal concurrency with the Meyers Singleton because the

⁶¹https://en.wikipedia.org/wiki/ARM_architecture

⁶²<https://en.wikipedia.org/wiki/PowerPC>

⁶³<http://preshing.com/>

Meyers Singleton is nearly as fast as the single threaded implementation.

Ongoing Optimisation with CppMem

I start with a small program and improve it successively. I verify each step of my process with CppMem. CppMem⁶⁴ is an interactive tool for exploring the behaviour of small code snippets using the C++ memory model.

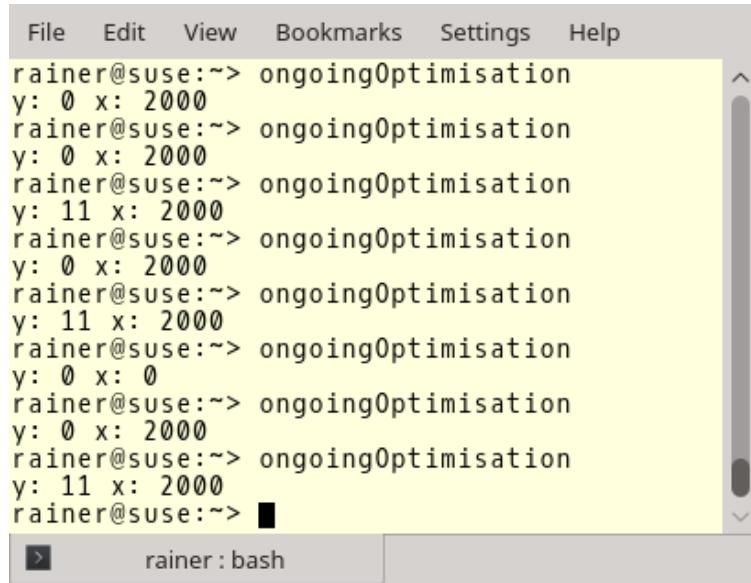
First, here is the small program.

The reference program for the ongoing optimisation

```
1 // ongoingOptimisation.cpp
2
3 #include <iostream>
4 #include <thread>
5
6 int x = 0;
7 int y = 0;
8
9 void writing(){
10    x = 2000;
11    y = 11;
12 }
13
14 void reading(){
15    std::cout << "y: " << y << " ";
16    std::cout << "x: " << x << std::endl;
17 }
18
19 int main(){
20    std::thread thread1(writing);
21    std::thread thread2(reading);
22    thread1.join();
23    thread2.join();
24 }
```

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 sounds straightforward, but even in this simple program we get three different results:

⁶⁴<http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/>



```
File Edit View Bookmarks Settings Help
rainer@suse:~> ongoingOptimisation
y: 0 x: 2000
rainer@suse:~> ongoingOptimisation
y: 0 x: 2000
rainer@suse:~> ongoingOptimisation
y: 11 x: 2000
rainer@suse:~> ongoingOptimisation
y: 0 x: 2000
rainer@suse:~> ongoingOptimisation
y: 11 x: 2000
rainer@suse:~> ongoingOptimisation
y: 0 x: 0
rainer@suse:~> ongoingOptimisation
y: 0 x: 2000
rainer@suse:~> ongoingOptimisation
y: 11 x: 2000
rainer@suse:~>
```

The reference program

I have two questions in mind for my process of ongoing optimisation.

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. I also present the possible values for x and y in a table.

However, I haven't explained what I mean by ongoing optimisation. It's pretty simple; I successively optimise the program by weakening the C++ memory-ordering. These are my optimisation 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 optimisation, you should have a basic understanding of CppMem. The chapter [CppMem](#) gives you a simplified introduction.

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` are protected. As a result, the program has undefined behaviour. 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 of the 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 emphasise 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.

This was my reasoning. Now we should have a look at what CppMem reports about the undefined behaviour of the program.

CppMem allows me to reduce the program to its bare minimum.

CppMem: unsynchronised 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.

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 analyse the annotated graph (3).

CppMem: Interactive C/C++ memory model

Model

standard preferred release_acquire tot relaxed_only

Program

examples/Paper

C Execution

```
int main(){
    int x= 0;
    int y= 0;
    {{{
        x= 2000;
        y= 11;
    }}{
        y;
        x;
    }}}

```

1 **4 executions; 1 consistent, not race free**

No next consistent.

Display Relations

sb asw dd cd
 rf mo sc lo
 hb vse lthb sw rs hrs dob cad
 unsequenced_races data_races

Display Layout

dot neato_par neato_par_Init neato_downwards
 tex

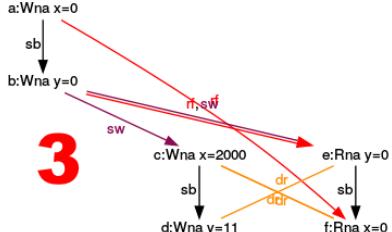
Execution candidate no. 1 of 4

2

Model Predicates

consistent_race_free_execution = **false**

consistent_execution = **true**
 assumptions = **true**
 well_formed_threads = **true**
 well_formed_rf = **true**
 locks_only_consistent_locks = **true**
 locks_only_consistent_lo = **true**
 consistent_mo = **true**
 sc_accesses_consistent_sc = **true**
 sc_fenced_sc_fences_heeded = **true**
 consistent_hb = **true**
 consistent_rf = **true**
 det_read = **true**
 consistent_non_atomic_rf = **true**
 consistent_atomic_rf = **true**
 coherent_memory_use = **true**
 rmw_atomicity = **true**
 sc_accesses_sc_reads_restricted = **true**
 unsequenced_races are **absent**
 data_races are **present**
 indeterminate_reads are **absent**
 locks_only_bad_mutexes are **absent**

3 

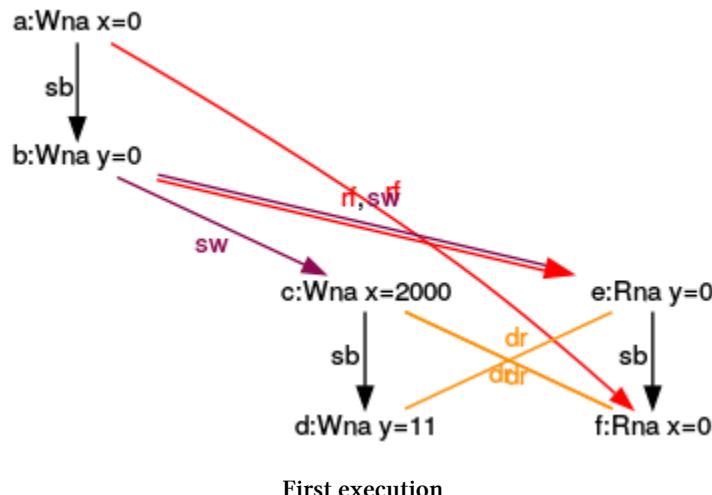
Files: [out.exc](#), [out.dot](#), [out.dsp](#), [out.tex](#)

A data race with non-atomics

You get the most out of CppMem by analysing the various graphs.

First Execution

What conclusions can we derive from the following graph?



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 holds 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 true because the expressions (f) takes place in a separate thread. Creation of a thread is a synchronisation point. Everything that happens before the thread creation is visible in the thread. For symmetry reasons, the same argument holds true 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 behaviour.

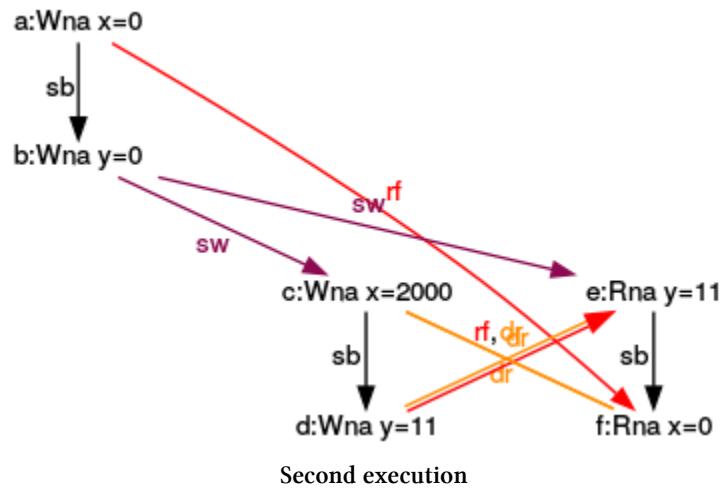


Why is the execution consistent?

The execution is consistent because the values x and y are initialised from the values in the main thread (a) and (b). The initialisation of the values x and y from the expressions (c) and (d) is not consistent with the memory model.

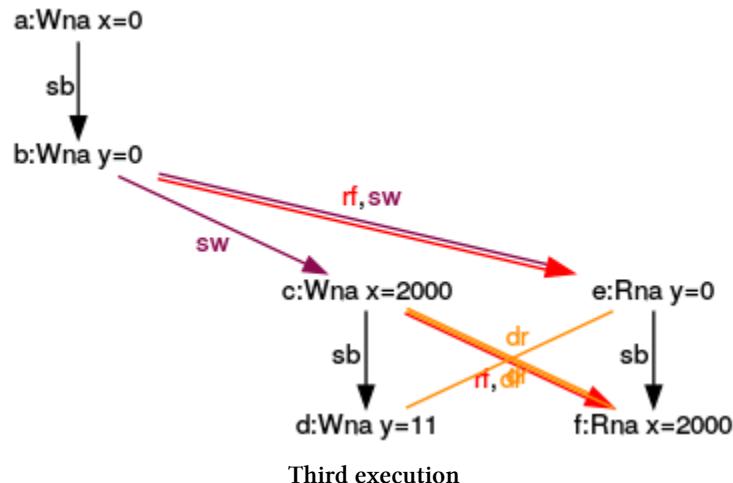
The next three executions are not consistent.

Second Execution



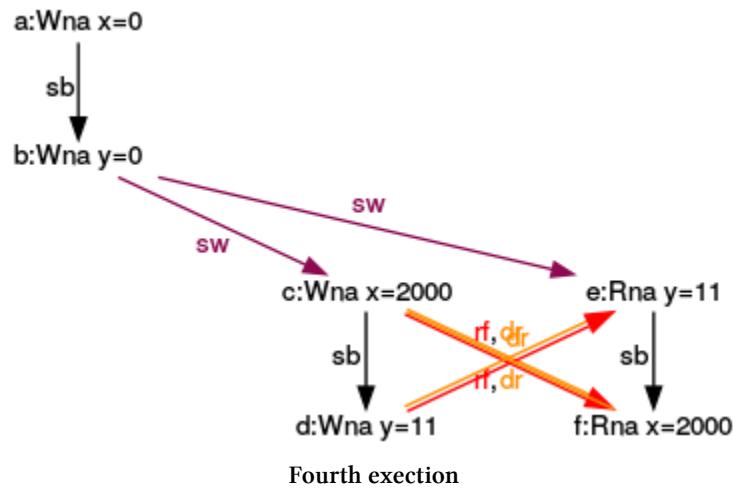
The expression (e) reads in this non-consistent execution the value of y from the expression (d). The writing of (d) happens concurrently with the reading of (e).

Third Execution



This execution is symmetric to the previous execution. The expression (f) reads from expression (c) concurrently.

Fourth Execution



Now everything goes wrong. The expressions (e) and (f) read from the expressions (d) and (c) concurrently.

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 behaviour.
- 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-synchronised access to `x` and `y`.

CppMem: unsynchronised 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 the 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 synchronised, and we got a data race; therefore, undefined behaviour. The most obvious way for synchronisation is to use locks.

CppMem: Locks

Both threads `thread1` and `thread2` use the same mutex, wrapped in a `std::lock_guard`.

Ongoing optimisations with locks

```

1 // ongoingOptimisationLock.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 << std::endl;
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 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 synchronisation is often too heavyweight. I now switch to a more lightweight strategy and use atomics.

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 optimised version of the program using atomics.

Ongoing optimisations with atomics (sequential consistency)

```

1 // ongoingOptimisationSequentialConsistency.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() << std::endl;
18 }
19
20 int main(){
21     std::thread thread1(writing);
22     std::thread thread2(reading);
23     thread1.join();
24     thread2.join();
25 }

```

Let's analyse 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`? The 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() << std::endl;`

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.

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.

CppMem: Interactive C/C++ memory model 2

Model

standard preferred release_acquire tot relaxed_only

Program

examples/LB_load_buffering (LB+acq_rel+acq_rel.c)

C Execution

```
int main(){
atomic_int x=0;
atomic_int y=0;
{{{
    x.store(2000);
    y.store(11);
}
|||
    y.load();
    x.load();
}}}
```

1 **384 executions; 6 consistent, all race free**

Model Predicates

- consistent_race_free_execution = true
- consistent_execution = true
- assumptions = true
- well_formed_threads = true
- well_formed_rf = true
- locks_only_consistent_locks = true
- locks_only_consistent_lo = true
- consistent_mo = true
- sc_accesses_consistent_sc = true
- sc_fenced_sc_fences_heeded = true
- consistent_hb = true
- consistent_rf = true
- det_read = true
- consistent_non_atomic_rf = true
- consistent_atomic_rf = true
- coherent_memory_use = true
- rmw_atomicity = true
- sc_accesses_sc_reads_restricted = true
- unsequenced_races are absent
- data_races are absent
- ineterminate_reads are absent
- locks_only_bad_mutexes are absent

Computed executions

Display Relations

- sb asw dd cd
- rf mo sc lo
- hb vse lthb sw rs hrs dob cad
- unsequenced_races data_races

Display Layout

- dot neato_par neato_par_init neato_downwards
- tex

Execution candidate no. 24 of 384

Model Predicates

- consistent_race_free_execution = true
- consistent_execution = true
- assumptions = true
- well_formed_threads = true
- well_formed_rf = true
- locks_only_consistent_locks = true
- locks_only_consistent_lo = true
- consistent_mo = true
- sc_accesses_consistent_sc = true
- sc_fenced_sc_fences_heeded = true
- consistent_hb = true
- consistent_rf = true
- det_read = true
- consistent_non_atomic_rf = true
- consistent_atomic_rf = true
- coherent_memory_use = true
- rmw_atomicity = true
- sc_accesses_sc_reads_restricted = true
- unsequenced_races are absent
- data_races are absent
- ineterminate_reads are absent
- locks_only_bad_mutexes are absent

Diagram

Files: [out.exc](#) [out.dot](#) [out.dsp](#) [out.tex](#)

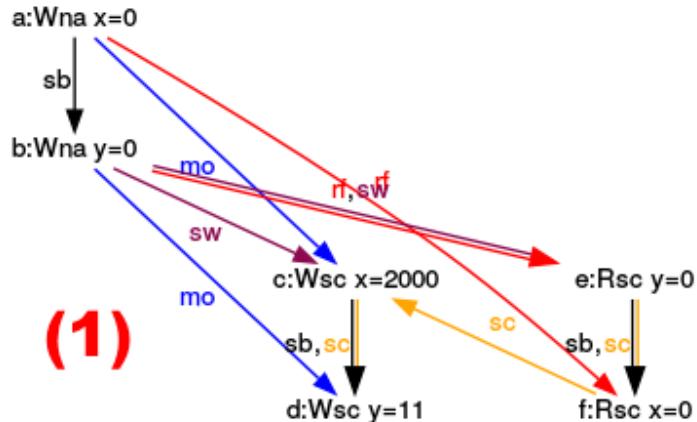
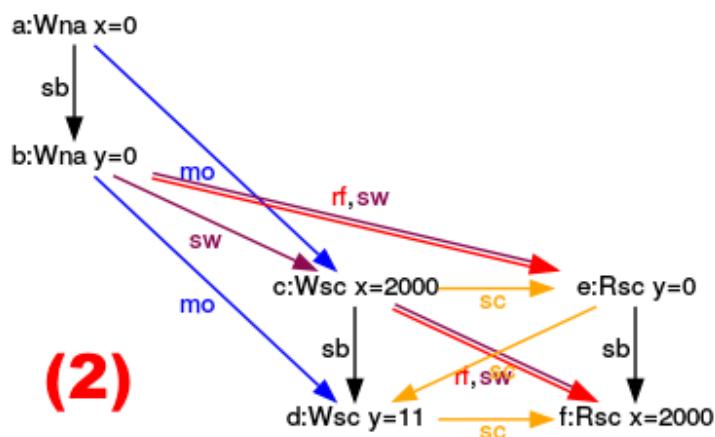
Ongoing Optimisation (sequential consistency)

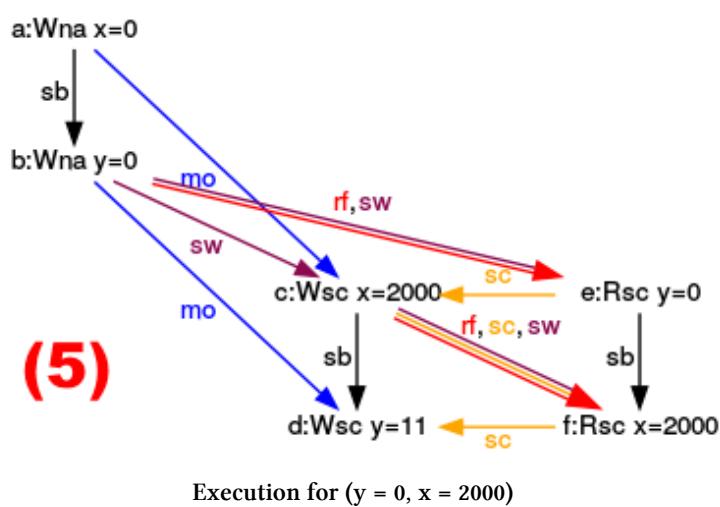
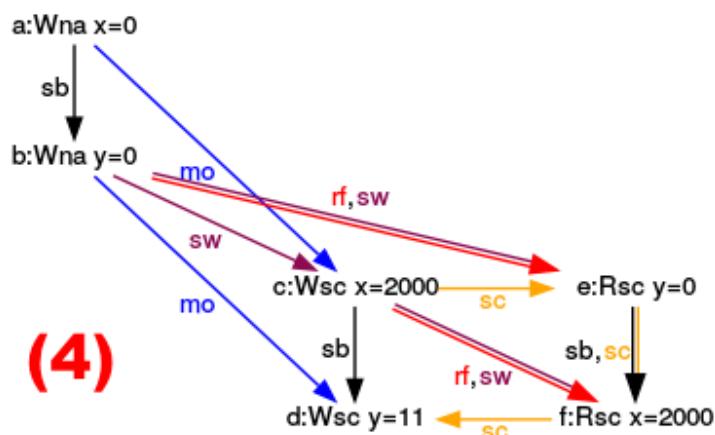
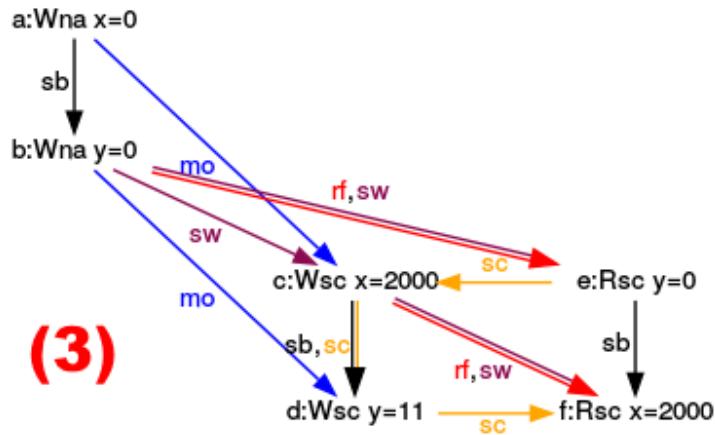
There are 384 (1) possible execution candidates, only 6 of them are consistent. No candidate has a **data race**. I'm only interested in the six consistent executions and ignore the other 378 non-consistent executions. Non-consistent means, for example, that they do not respect the **modification order** of the **memory model**.

I use the interface (2) to get the six annotated graphs.

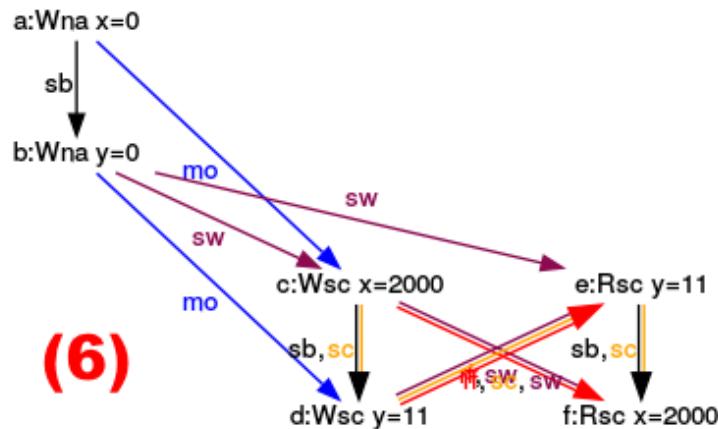
We already know that all values for **x** and **y** are possible except for **y = 11** and **x = 0**. This is because of the sequential consistency. Now I'm curious, which interleaving of threads produces which values for **x** and **y**?

Execution for (y = 0, x = 0)

Execution for $(y = 0, x = 0)$ Executions for $(y = 0, x = 2000)$ Execution for $(y = 0, x = 2000)$



Execution for (y = 11, x = 2000)

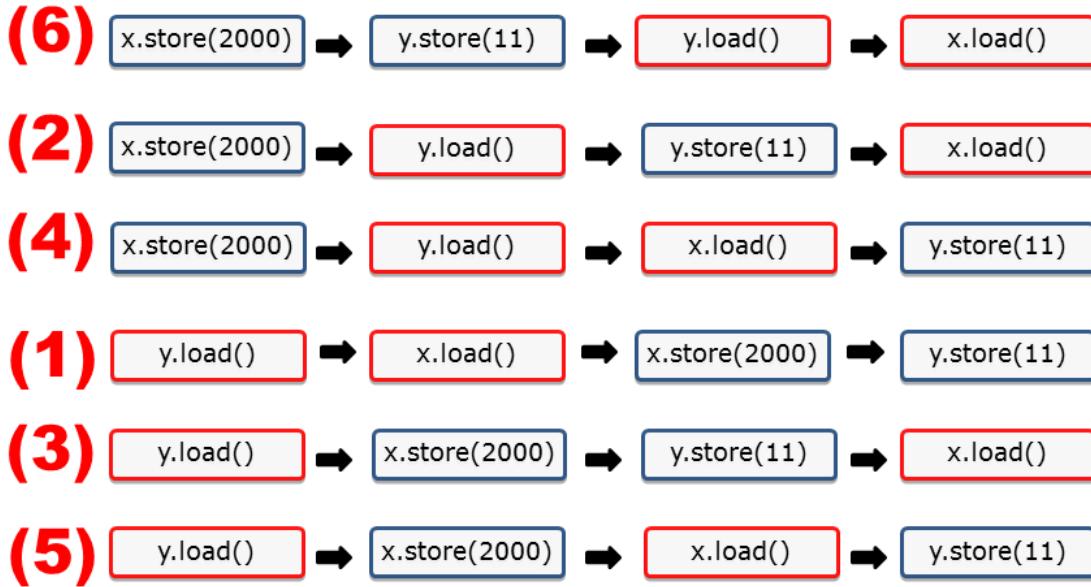


Execution for (y = 11, x = 2000)

I'm not done with my analysis. I'm interested in the answer to the question: Which sequence of instructions corresponds to which of the six graphs?

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 $x = 2000 \Rightarrow$ read $y = 0 \Rightarrow$ write $y = 11 \Rightarrow$ 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.

CppMem: Atomics with Acquire-Release Semantic

The synchronisation in the [acquire-release semantic](#) takes place between atomic operations on the same atomic. This is in contrast to the sequential consistency where we have synchronisation 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 optimisations with atomics (acquire-release semantic)

```

1 // ongoingOptimisationAcquireRelease.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, std::memory_order_relaxed);
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.load(std::memory_order_relaxed) << std::endl;
18 }
19
20 int main(){
21     std::thread thread1(writing);
22     std::thread thread2(reading);
23     thread1.join();
24     thread2.join();
25 }
```

On 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 synchronisation 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 synchronises 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 very impressive 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 case of the acquire-release semantic 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)

<code>y</code>	<code>x</code>	Values possible?
0	0	Yes
11	0	
0	2000	Yes
11	2000	Yes

Once more. Let's verify our thinking with CppMem.

CppMem

Here is the corresponding program.

CppMem: atomics(acquire-release semantic)

```

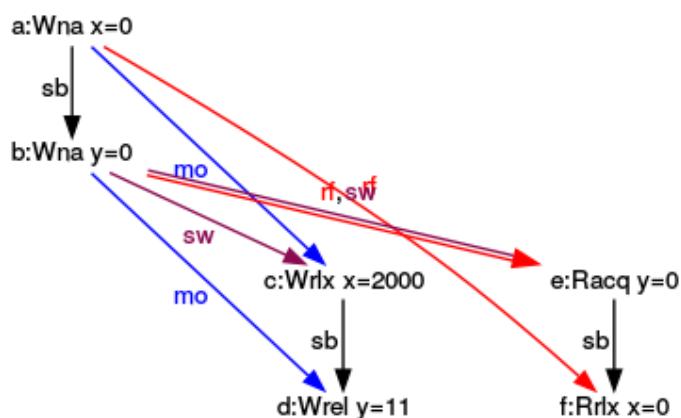
1 int main(){
2     atomic_int x = 0;
3     atomic_int y = 0;
4     {{{ {
5         x.store(2000, memory_order_relaxed);
6         y.store(11, memory_order_release);
7     }
8     ||| {
9         y.load(memory_order_acquire);
10        x.load(memory_order_relaxed);
11    }
12 }}}
13 }
```

We already know that all results are possible except for (y = 11, x = 0).

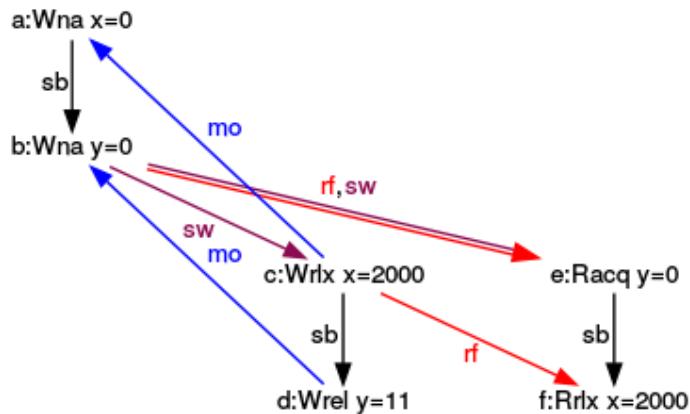
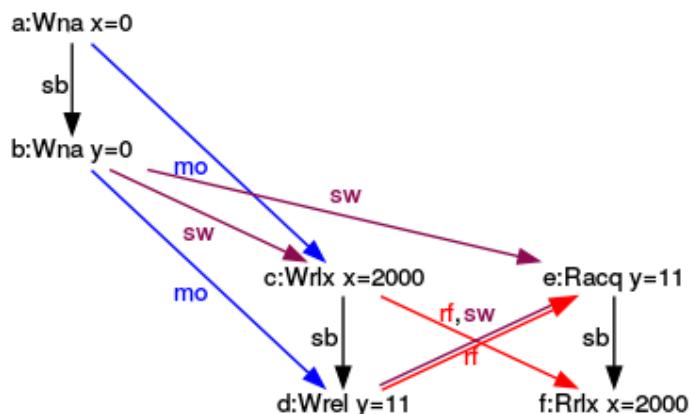
Possible executions

I only refer to the three graphs with consistent execution. The graphs show that there is 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 places in the main thread or a separate thread. The graphs show the *synchronizes-with* relation using a sw annotated arrow.

Execution for (y = 0, x = 0)



Execution for (y = 0, x = 0)

Execution for (y = 0, x = 2000)Execution for ($y = 0, x = 2000$)**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.

CppMem: Atomics with Non-atomics

A typical misunderstanding in the application of 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 have to be an atomic variable and we can further optimise the program.

Ongoing optimisations with atomics with non-atomics

```
1 // ongoingOptimisationAcquireReleaseBroken.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 << std::endl;
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 behaviour. 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` in line 17. If not the reading of `x` is executed at the same time 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.

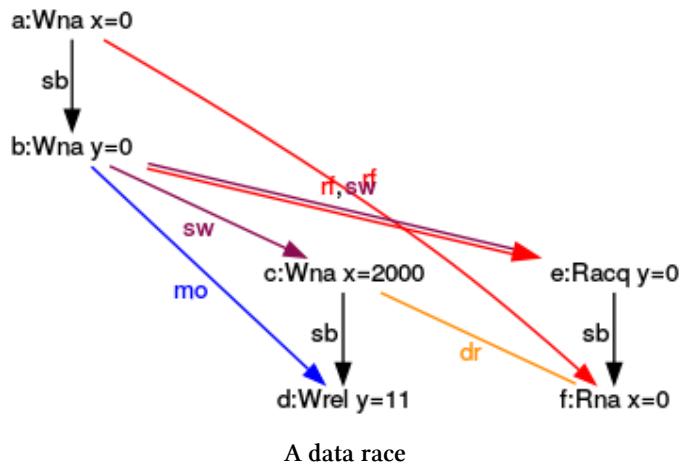
CppMem

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.



The final step in the process of ongoing optimisation is still missing: relaxed semantic.

CppMem: Atomics with Relaxed Semantic

With the relaxed semantic, we have no synchronisation and ordering constraints on atomic operations, only the atomicity of the operations is guaranteed.

Ongoing optimisation with atomics (relaxed semantic)

```
1 // ongoingOptimisationRelaxedSemantic.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, std::memory_order_relaxed);
12     y.store(11, std::memory_order_relaxed);
13 }
14
15 void reading(){
16     std::cout << y.load(std::memory_order_relaxed) << " ";
17     std::cout << x.load(std::memory_order_relaxed) << std::endl;
18 }
19
20 int main(){
21     std::thread thread1(writing);
22     std::thread thread2(reading);
23     thread1.join();
24     thread2.join();
25 }
```

For the relaxed semantic, my fundamental questions are straightforward to answer. These are my questions:

1. Does the program have well-defined behaviour?
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 optimisations 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?

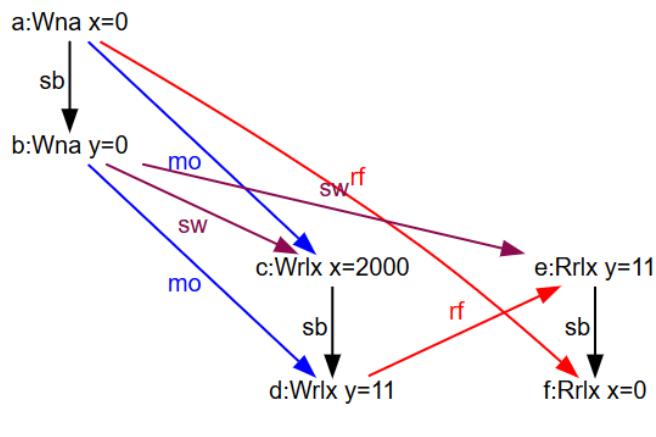
CppMem

CppMem: atomics (relaxed 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_relaxed);
7     }
8     ||| {
9         y.load(memory_order_relaxed);
10        x.load(memory_order_relaxed);
11    }
12 }}}
13 }
```

That was the CppMem program. Now to the graph that produces the counter-intuitive behaviour.



`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).

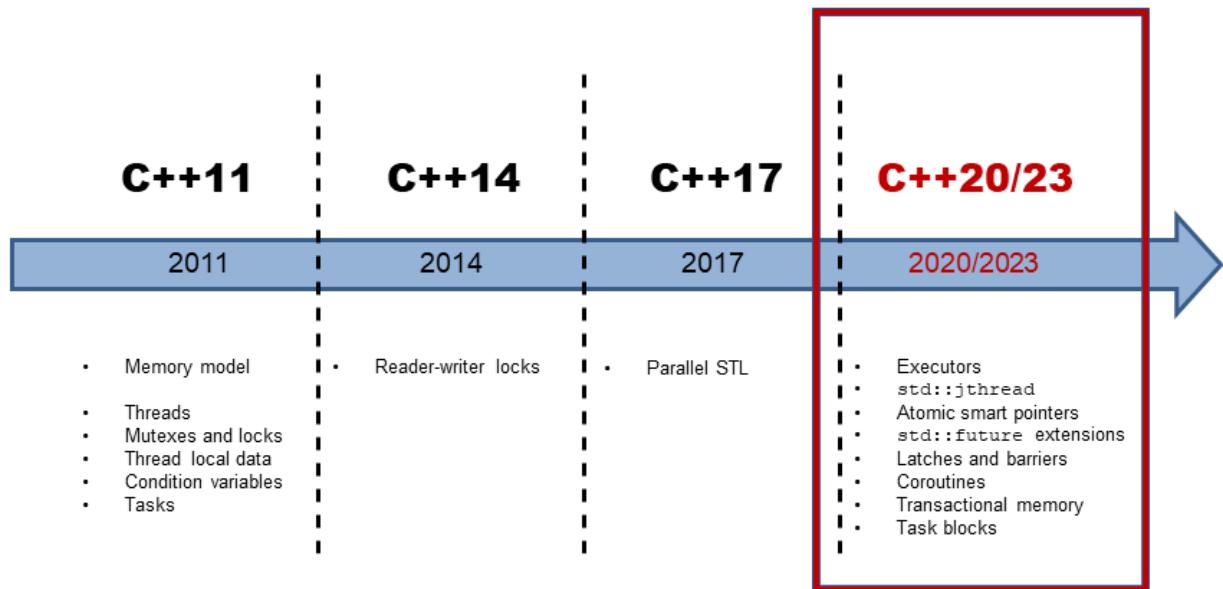
Conclusion

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.

The Future: C++20/23

This chapter is about the future of C++. 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++20/23 standard. Second, if a feature makes it into the C++20/23 standards, the interface of that feature changes very likely. This hold in particular true for the executors. I will update this book on a regular basis and will, therefore, reflect the updated and new proposals in this chapter.

My goal in this chapter is quite simple: give you an idea about the upcoming concurrency features in C++.



Concurrency in C++20/23

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++. At 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 standardised with C++20. This chapter is mainly based on the proposals to the design of executors [P0761⁶⁵](#), and to their formal description [P0443⁶⁶](#). This chapter also refers to the relatively new “Modest Executor Proposal” [P1055⁶⁷](#).

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 just be scheduled.
- **How:** The callable may run on a CPU or GPU or even be executed in a vectorised way.

More formally, each executor has properties associated with the performed [execution function](#). You can specify these properties.

1. **Directionality:** The execution function can be of kind “fire and forget”, return a future, or return a continuation.
2. **Cardinality:** The execution function can create one or multiple execution agents.
3. **Blocking:** The function may block or not.
4. **Continuations:** The task may be performed on the client’s calling thread.
5. **Future task submission:** Specify if outstanding work remains.
6. **Bulk forward progress guarantees:** Specifies the forward guarantees in contrast to other execution agents.
7. **Thread execution mapping guarantees:** Should each execution agent be mapped to a separated thread.
8. **Allocators:** Associates an allocator with an executor.

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 [N4734⁶⁸](#), but also for 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.

⁶⁵<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0761r2.pdf>

⁶⁶<http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0443r7.html>

⁶⁷<http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1055r0.pdf>

⁶⁸<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4734.pdf>

First Examples

Using an Executor

Here are a few code snippets showing the usage of executors:

- The promise `std::async`

Executing an `std::async`

```
// 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!" << std::endl;
});
```

- The STL algorithm `std::for_each`

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);
```

- Network 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)
    {
        ...
    }
);
```

- Network 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 [N4734⁶⁹](#) allows 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.

Obtaining an Executor

There are various ways to obtain an executor.

- From the execution context `static_thread_pool`

⁶⁹<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4734.pdf>

An executor from an execution context

```
// create a thread pool with 4 threads
static_thread_pool pool(4);

// get an executor from the thread pool
auto exec = pool.executor();

// use the executor on some long-running task
auto task1 = long_running_task(exec);
```

- From the [execution policy](#) `std::execution::par`

An executor from an execution policy

```
// 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);
```

- From the system executor

This is the default executor that usually uses a thread for the execution. It is used if not another one is specified.

- From an executor adapter

An executor from an executor adaptor

```
// 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.

Goals of an Executor Concept

What are the goals of an executor concept according to the proposal [P1055⁷⁰](#)?

1. **Batchable**: control the trade-off between the cost of the transition of the callable and the size of it.
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 [LIFO⁷¹](#) (Last In, First Out), [FIFO⁷²](#) (First In, First Out) execution, priority or time constraints, or even sequential execution.
4. **Controllable**: the callable has to be targetable to a specific compute resource, deferred, or even cancelled.
5. **Continuable**: to control an 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 should also be possible.
6. **Layerable**: hierarchies allow capabilities to be added without increasing the complexity of the simpler use-cases.
7. **Usable**: ease of use for the implementer and the user should be the main 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.

Terminology

The Proposal [P0761⁷³](#) defines a few new terms for the execution of a callable:

- **Execution resource**: is an instance of a hardware and/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**: is a program object that represents a specific collection of execution resources and the execution agents that exist 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 an 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.

⁷⁰<http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1055r0.pdf>

⁷¹[https://en.wikipedia.org/wiki/Stack_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Stack_(abstract_data_type))

⁷²[https://en.wikipedia.org/wiki/FIFO_\(computing_and_electronics\)](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics))

⁷³<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0761r2.pdf>

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
 - 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.

First, I refer to the single cardinality case. A oneway execution function is a fire and forget job. It's quite similar to a [fire and forget future](#), but it does not automatically block in the destructor of the future. A twoway execution function returns you a future which you can use to pick up the result. This behaves similarly to a [std::promise](#) that give 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.

Second, the bulk cardinality case is more complicated. These functions create a group of execution agents, and each of these execution agents calls the given callable. They return the result of a factory and not the result of a single callable `f` invoked by the execution agents. The client is responsible for disambiguating the right result via this factory.

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

```
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

```
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);
}
```

A Cooperatively Interruptible Joining Thread

`std::jthread` stands for joining thread. In addition to `std::thread` from C++11, `std::jthread` can automatically join the started thread and signal an interrupt. Its characteristics are described in the proposal [P0660R5⁷⁴](#): *A Cooperatively Interruptible Joining Thread*.

Automatically Joining

Here is the *non-intuitive* behaviour 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`

```
// threadJoinable.cpp

#include <iostream>
#include <thread>

int main(){
    std::cout << std::endl;
    std::cout << std::boolalpha;

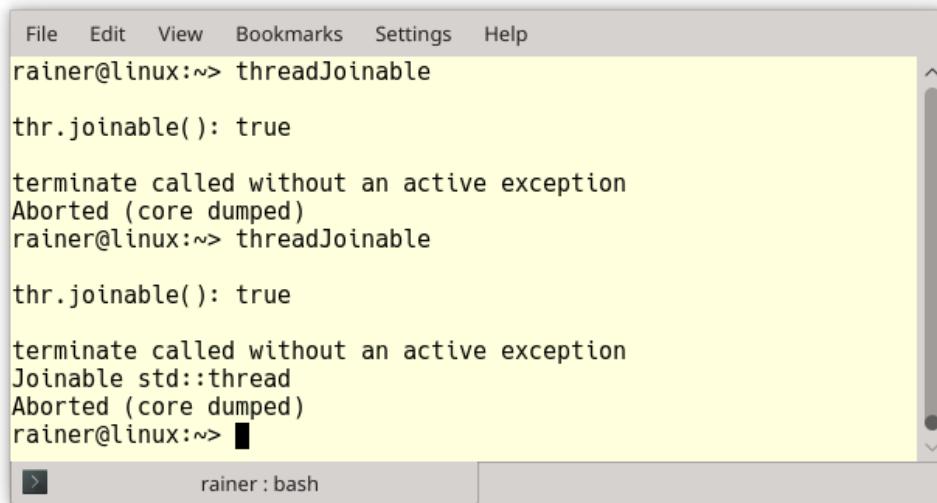
    std::thread thr{}; std::cout << "Joinable std::thread" << std::endl; };

    std::cout << "thr.joinable(): " << thr.joinable() << std::endl;

    std::cout << std::endl;
}
```

When executed, the program terminates.

⁷⁴<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0660r5.pdf>



```

File Edit View Bookmarks Settings Help
rainer@linux:~/threadJoinable
thr.joinable(): true
terminate called without an active exception
Aborted (core dumped)
rainer@linux:~/threadJoinable
thr.joinable(): true
terminate called without an active exception
Joinable std::thread
Aborted (core dumped)
rainer@linux:~/

```

Terminating a joinable std::jthread

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

In the next example, I replace the header `<thread>` with `“jthread.hpp”` and use `std::jthread` from the upcoming C++ standard.

Terminating a still joinable std::jthread

```

// jthreadJoinable.cpp

#include <iostream>
#include "jthread.hpp"

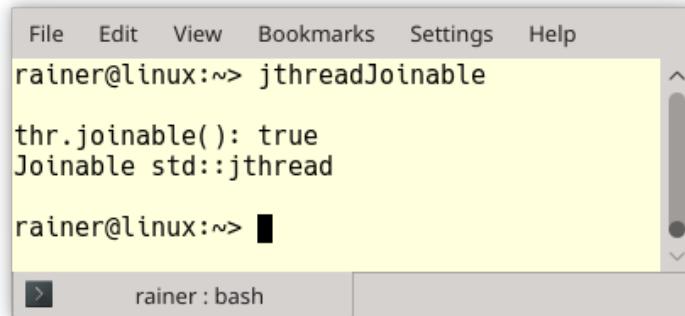
int main(){
    std::cout << std::endl;
    std::cout << std::boolalpha;

    std::jthread thr{[]{ std::cout << "Joinable std::thread" << std::endl; }};
    std::cout << "thr.joinable(): " << thr.joinable() << std::endl;

    std::cout << std::endl;
}

```

Now, the thread `thr` automatically joins in its destructor such as in this case if still joinable.



```

File Edit View Bookmarks Settings Help
rainer@linux:~> jthreadJoinable
thr.joinable(): true
Joinable std::jthread
rainer@linux:~> █
rainer:bash

```

Using a `std::jthread` which automatically joins

Interrupt a `std::jthread`

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

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

```

// interruptJthread.cpp

#include "jthread.hpp"
#include <chrono>
#include <iostream>

using namespace::std::literals;

int main(){
    std::cout << std::endl;

    std::jthread nonInterruptable([]{
        int counter{0};
        while (counter < 10){
            std::this_thread::sleep_for(0.2s);
            std::cerr << "nonInterruptable: " << counter << std::endl;
            ++counter;
        }
    });

    std::jthread interruptable([](std::interrupt_token itoken){
        int counter{0};
        while (counter < 10){
            std::this_thread::sleep_for(0.2s);

```

```
    if (itoken.is_interrupted()) return;
    std::cerr << "interruptable: " << counter << std::endl;
    ++counter;
}
});

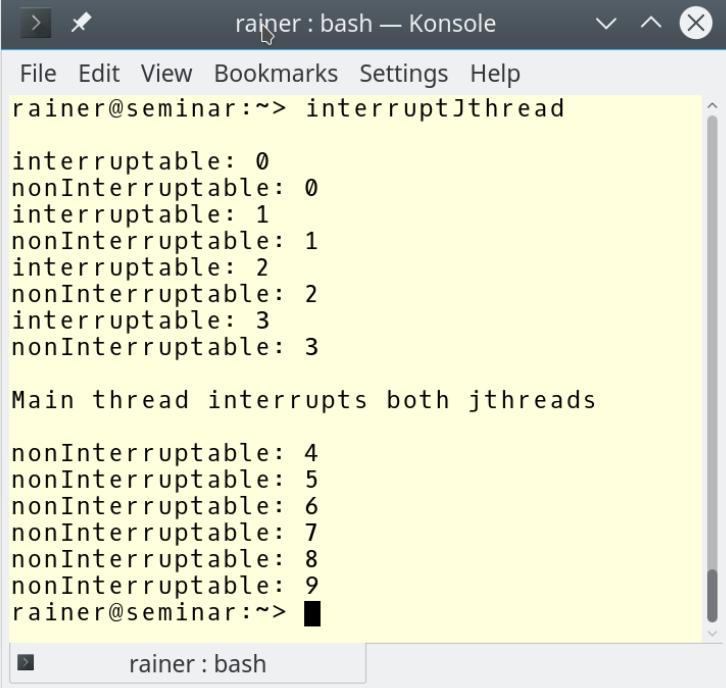
std::this_thread::sleep_for(1s);

std::cerr << std::endl;
std::cerr << "Main thread interrupts both jthreads" << std::endl;
nonInterruptable.interrupt();
interruptable.interrupt();

std::cout << std::endl;

}
```

I started in the main program the two threads `nonInterruptable` and `interruptable` (lines 13 and 22). In contrast to the thread `nonInterruptable`, the thread `interruptable` gets a `std::interrupt_token` and uses it in line 26 to check if it was interrupted: `itoken.is_interrupted()`. In case of an interrupt the lambda function returns and, therefore, the thread ends. The call `interruptable.interrupt()` (line 37) triggers the end of the thread. This does not hold for the previous call `nonInterruptable.interrupt()`, which does not have an effect.



```

rainer@seminar:~> interruptJthread
interruptable: 0
nonInterruptable: 0
interruptable: 1
nonInterruptable: 1
interruptable: 2
nonInterruptable: 2
interruptable: 3
nonInterruptable: 3

Main thread interrupts both jthreads

nonInterruptable: 4
nonInterruptable: 5
nonInterruptable: 6
nonInterruptable: 7
nonInterruptable: 8
nonInterruptable: 9
rainer@seminar:~>

```

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

Here are more details to interrupt tokens, the joining threads, and condition variables.

Interrupt Tokens

An interrupt token `std::interrupt_token` models shared ownership and can be used to signal once if the token is valid. It provides the three methods `valid`, `is_interrupted`, and `interrupt`.

The methods of the `std::interrupt_token` itoken

Method	Description
<code>itoken.value()</code>	true if the interrupt token can be used to signal interrupts.
<code>itoken.is_interrupted()</code>	true if initialized with true or initialised with false and <code>interrupt()</code> was called by one of the owners.
<code>itoken.interrupt()</code>	If <code>!valid()</code> or <code>is_interrupted()</code> the call has no effect. Otherwise, signals an interrupt so that <code>is_interrupted() == true</code> . Returns the value of <code>is_interrupted()</code> prior to the call.

If the interrupt token should be temporarily disabled, you can replace it with a default constructed token. A default constructed token is not valid. The following code snippet shows how to disable and enable the capability of a thread to accept signals.

Temporarily disable a interrupt token

```

1 std::jthread jthr([](std::interrupt_token itoken){
2     ...
3     std::interrupt_token interruptDisabled;
4     std::swap(itoken, interruptDisabled);
5     ...
6     std::swap(itoken, interruptDisabled);
7     ...
8 }
```

`std::interrupt_token interruptDisabled` is not valid. This means, from line 4 to 5 the interrupt token is disabled, from line 6 on enabled.

Joining Threads

A `std::jhread` is a `std::thread` with the additional functionality to signal an interrupt and to automatically `join()`. To support this functionality it has a `std::interrupt_token`.

The methods of the `std::jhread` `jthr`

Method	Description
<code>jthr.get_original_interrupt_token()</code>	Returns the <code>itoken</code> .
<code>jthr.interrupt()</code>	Returns <code>itoken.interrupt()</code> .

New Wait Overloads for Condition Variables

The three wait variations to `wait_for` and `wait_until` of the `std::condition_variable` get new overloads. They take a `std::interrupt_token`.

Three new wait overloads

```

template <class Predicate>
bool wait_until(unique_lock<mutex>& lock,
               Predicate pred,
               interrupt_token itoken);

template <class Rep, class Period, class Predicate>
bool wait_for(unique_lock<mutex>& lock,
              const chrono::duration<Rep, Period>& rel_time,
              Predicate pred,
              interrupt_token itoken);
```

```
template <class Clock, class Duration, class Predicate>
bool wait_until(unique_lock<mutex>& lock,
                const chrono::time_point<Clock, Duration>& abs_time,
                Predicate pred,
                interrupt_token itoken);
```

This new overloads require a predicate. The versions ensure to get notified if an interrupt is signaled for the passed `std::interrupt_token` `itoken`. After the wait calls, you can check if an interrupt occurred.

Handle interrupts with `wait`

```
cv.wait_until(lock, predicate, itoken);
if (itoken.is_interrupted()){
    // interrupt occurred
}
```

Atomic Smart Pointers

A `std::shared_ptr` consists of a control block and its resource. The control block is thread-safe, but the access to the resource is not. 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

Before I start, I want to make a short detour. This detour should only emphasise how important it is that the `std::shared_ptr` has well-defined multithreading semantic. 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 [data races](#) and hence for undefined behaviour. On the other hand, there is the guideline in modern C++: **Don't touch memory**. This means use smart pointers in multithreading programs.

The proposal [N4162⁷⁵](#) for atomic smart pointers directly addresses the deficiencies of the current implementation. The deficiencies boil down to these three points: consistency, correctness, and performance. I provide an overview of these three points. See the proposal N4162 for details.

- **Consistency:** the atomic operations for `std::shared_ptr` are the only atomic operations for a non-atomic data type.
- **Correctness:** the usage of the global atomic operations is quite error-prone because the right 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 behaviour because of a data race. If we used an atomic smart pointer instead, the type-system would not allow it.
- **Performance:** the atomic smart pointers have a big advantage compared to the free `atomic_*` functions. The atomic versions are designed for the special use case and can internally have a `std::atomic_flag` as a kind of cheap [spinlock](#). Designing the non-atomic versions of the pointer functions to be thread safe would be overkill if they are used in a single-threaded scenario. They would have a performance penalty.

For me, the correctness argument is 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.

⁷⁵<http://wg21.link/n4162>

A thread-safe singly linked list

```

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

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

```

A thread-safe singly linked list

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

The proposal [N4162⁷⁶](#) 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 specialisation of `std::atomic<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.

⁷⁶<http://wg21.link/n4162>

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.

Concurrency TS v1

`std::future`

The name extended futures is quite easy to explain. First, the interface of the C++11 `std::future` 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 methods:

- 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 method `then` that attaches a continuation to a future.

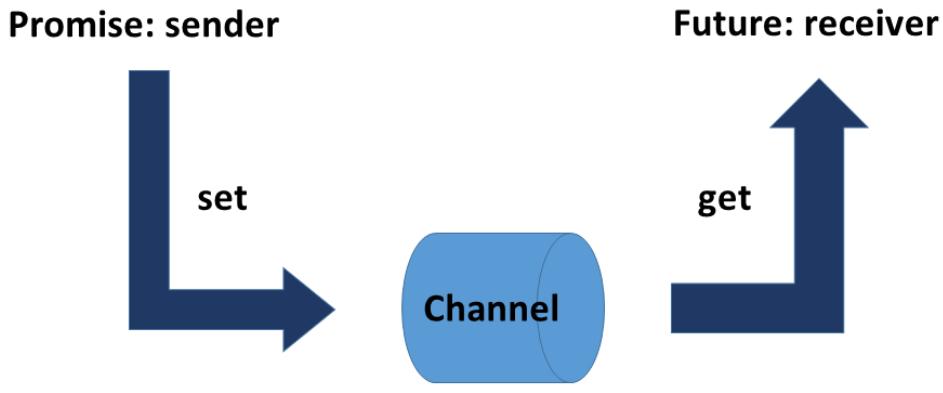
At first, the state of a future can be `valid` or `ready`.

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 to put it 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.



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 method `then` for the continuation of a future.

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 [N3721⁷⁷](#). 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.

⁷⁷<https://isocpp.org/files/papers/N3721.pdf>

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/asynchronous and the second call is blocking/synchronous. 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.

`std::async`, `std::packaged_task`, and `std::promise`

There is not much to say about the extensions of `std::async`, `std::packaged_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.

Creating new Futures

C++20 gets four new functions for creating special futures. These functions are `std::make_ready_future`, `std::make_exceptional_future`, `std::when_all`, and `std::when_any`. First, let's look at the functions `std::make_ready_future`, and `std::make_exceptional_future`.

`std::make_ready_future` and `std::make_exceptional_future`

Both functions create a future that is immediately ready. 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

```
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.

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

```
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 output of the function 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 next two examples show the usage of `std::when_all` and `std::when_any`.

`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.

`std::when_any`

Future composition with std::when_any

```

1 #include <future>
2 #include <vector>
3
4 using namespace std;
5
6 int main(){
7
8     vector<future<int>> v{ ... };
9     auto future_any = when_any(v.begin(), v.end());
10
11    when_any_result<vector<future<int>>> result = future_any.get();
12
13    future<int>& ready_future = result.futures[result.index];
14
15    auto myResult = ready_future.get();
16
17 }
```

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 it is 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 standardised futures nor the [concurrency TS v1 futures](#)⁷⁸ are “not as generic, expressive or powerful as they should be” [P0701r1](#)⁷⁹. Additionally, the `executors` as basic building block for executing something have to be unified with the new futures.

Unified Futures

What are the disadvantages of the already standardised and the futures from the concurrency TS 1?

Disadvantages

The already mentioned document gives an excellent description to the deficiencies of the futures.

⁷⁸<http://en.cppreference.com/w/cpp/experimental/concurrency>

⁷⁹<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0701r1.html>

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, or in a thread pool, or just sequentially.

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 P0701r1⁸⁰:

Executor propagation

```
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.
```

⁸⁰<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0701r1.html>

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`

```
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

```
std::future<int> f1 = std::async([]() { return 123; });
std::future<std::string> f2 = f1.then(std::to_string);
```

`when_all` and `when_any` Return Types are Unwieldy

The chapter to [std::when_all](#) and [std::when_any](#) shows their quite complicated usage.

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 the document P0701r1, this is not acceptable and error-prone.

I describe the peculiar behaviour of [Fire and Forget Futures](#) in the referenced chapter.

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` concurrency 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

```
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.

Five New Concepts

Their are five new concepts for futures and promises in the Proposal [1054R0](#)⁸¹.

- **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 which 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 of which is associated with a future and make the future ready with either a value or an exception.

The paper also provides the declaration of this new concepts.

⁸¹<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1054r0.html>

The five new concepts for futures and promises

```
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 method `via` that accepts 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` method for continuation. The `then` method 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.

Future Work

The Proposal [1054R0⁸²](#) left a few questions open.

- Forward progress guarantees for futures and promises.
- Requirements on synchronization for 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`.

⁸²<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1054r0.html>

Latches and Barriers

Latches and barriers are simple thread synchronisation mechanisms which 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++20/23 we get presumably latches and barriers in three variations: `std::latch`, `std::barrier`, and `std::flex_barrier`.

First, there are two questions:

1. What are the differences between these three mechanisms to synchronise threads? You can use a `std::latch` only once, but you can use a `std::barrier` and a `std::flex_barrier` more than once. Additionally, a `std::flex_barrier` enables you to execute a function 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 in combination 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.

Now, I have a closer look at these three coordination mechanisms.

`std::latch`

`std::latch` `latch` is a countdown counter. Its value is set in the constructor. A latch can decrement the counter by using the method `latch.count_down_and_wait` and blocks the thread until the counter becomes zero. In addition the method `latch.count_down` only decrements the counter by 1 without blocking. `std::latch` also has the method `latch.is_ready` that can be used to test if the counter is zero, and the method `latch.wait` to block the thread until the counter becomes zero. You cannot increment or reset the counter of a `std::latch`, hence you cannot reuse it.

Here is a short code snippet from the [N4204⁸³](#) proposal.

`std::latch`

```

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

```

⁸³<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4204.html>

```

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

I set the `std::latch completion_latch` in its constructor to `NTASKS` (line 2). The thread pool executes `NTASKS` (lines 4 - 8). At the end of each task (line 7), the counter is decremented. Line 11 is the barrier for the thread running the function `DoWork` and hence for the small workflow. This thread is blocked until all tasks have been finished.

A `std::barrier` is quite similar to a `std::latch`.

std::barrier

The subtle difference between `std::latch` and `std::barrier` is that you can use `std::barrier` more than once because its counter is reset to its previous value. Immediately after the counter becomes zero the so-called completion phase starts. `std::barrier` has an empty completion phase. This changes with `std::flex_barrier`. `std::barrier` has two interesting methods: `std::arrive_and_wait` and `std::arrive_and_drop`. While `std::arrive_and_wait` causes the thread to block at the synchronization point, `std::arrive_and_drop` removes the thread itself from the set of participating threads. It is unspecified whether this function blocks until the completion phase has ended.



The proposal N4204

The proposal uses a `vector<thread*>` and pushes the dynamically allocated threads onto the vector: `workers.push_back(new thread([&]{ ... })`. This is a memory leak. Instead you should put the threads into a `std::unique_ptr` or create them directly in the vector: `workers.emplace_back(&){ ... }`. This observation holds for the example with `std::barrier` and `std::flex_barrier`. The names in the example with `std::flex_barrier` are a little bit confusing. For example `std::flex_barrier` is called `notifying_barrier`. I changed the name to `flex_barrier`. Additionally, the variable `n_threads`, representing the number of threads, was not initialised. I initialised it to `NTASKS`.

Before I take a closer look at `std::flex_barrier` and the completion phase, in particular, I give a short example demonstrating the usage of `std::barrier`.

std::barrier

```

1  void DoWork() {
2      Tasks& tasks;
3      int n_threads{NTASKS};
4      vector<thread*> workers;
5
6      barrier task_barrier(n_threads);
7
8      for (int i = 0; i < n_threads; ++i) {
9          workers.push_back(new thread([&] {
10             bool active = true;
11             while(active) {
12                 Task task = tasks.get();
13                 // perform task
14                 ...
15                 task_barrier.arrive_and_wait();
16             }
17         });
18     }
19     // Read each stage of the task until all stages are complete.
20     while (!finished()) {
21         GetNextStage(tasks);
22     }
23 }
```

The `barrier` in line 6 is used to coordinate a number of threads that perform their task multiple times. The number of threads is `n_threads` (line 3). Each thread takes its task at line 12 via `tasks.get()`, performs it and blocks - once it is done with its task (line 15) - until all threads are done with their tasks. After that, it takes a new task in line 12 while `active` returns `true` in line 11.

In contrast to `std::barrier`, `std::flex_barrier` has an additional constructor.

std::flex_barrier

This additional constructor takes a `callable unit` that is invoked in the completion phase. The callable has to return a number. This number sets the value of the counter. A return of -1 means that the counter keeps the same counter value in the next iteration. Numbers smaller than -1 are not allowed.

The completion phase performs the following steps:

1. All threads are blocked.
2. An arbitrary thread is unblocked and executes the callable unit.

3. If the completion phase is done, all threads are unblocked.

The code snippet shows the usage of a `std::flex_barrier`.

`std::flex_barrier`

```

1  void DoWork() {
2      Tasks& tasks;
3      int initial_threads;
4      int n_threads{NTASKS};
5      atomic<int> current_threads(initial_threads);
6      vector<thread*> workers;
7
8      // Create a flex_barrier, and set a lambda that will be
9      // invoked every time the barrier counts down. If one or more
10     // active threads have completed, reduce the number of threads.
11     std::function rf = [&] { return current_threads; };
12     flex_barrier task_barrier(n_threads, rf);
13
14     for (int i = 0; i < n_threads; ++i) {
15         workers.push_back(new thread([&] {
16             bool active = true;
17             while(active) {
18                 Task task = tasks.get();
19                 // perform task
20                 ...
21                 if (finished(task)) {
22                     current_threads--;
23                     active = false;
24                 }
25                 task_barrier.arrive_and_wait();
26             }
27         }));
28     }
29
30     // Read each stage of the task until all stages are complete.
31     while (!finished()) {
32         GetNextStage(tasks);
33     }
34 }
```

The example follows a similar strategy as the previous example with `std::barrier`. The difference is that this time the `std::flex_barrier` counter is adjusted at runtime. Hence, the `std::flex_barrier`

`task_barrier` in line 11 gets a lambda function. This lambda function captures its variable `current_thread` by reference: `[&] { return current_threads; }`. The variable is decremented in line 21 and `active` is set to `false` if the thread has completed its task. Therefore the counter is decremented in the completion phase.

`std::flex_barrier` can increase the counter in contrast with `std::barrier` or `std::latch`.

You can read further details of `std::latch`⁸⁴, `std::barrier`⁸⁵, `std::flex_barrier`⁸⁶ at [cppreference.com](http://en.cppreference.com).

⁸⁴<http://en.cppreference.com/w/cpp/experimental/latch>

⁸⁵<http://en.cppreference.com/w/cpp/experimental/barrier>

⁸⁶http://en.cppreference.com/w/cpp/experimental/flex_barrier

Coroutines

Coroutines are functions that can suspend and resume their execution while keeping their state. The evolution of functions goes in C++ one step further. Coroutines are with high probability part of C++20.

What I present in this section as a new idea in C++20 is quite old. The term coroutine was coined by [Melvin Conway](#)⁸⁷. He used it in his publication on compiler construction in 1963. [Donald Knuth](#)⁸⁸ called procedures a special case of coroutines. Sometimes, it just takes a while to get your ideas accepted.

With the new keywords `co_await` and `co_yield`, C++20 extends the execution of C++ functions with two new concepts.

Thanks to `co_await` expression it is possible to suspend and resume the execution of the expression. If you use `co_await` expression in a function `func`, the call `auto getResult = func()` does not block if the result of the function is not available. Instead of resource-consuming blocking, you have resource-friendly waiting.

`co_yield` expression allows it to write a generator function. The generator function returns a new value each time. A generator function is a kind of data stream from which you can pick values. The data stream can be infinite. Therefore, we are in the centre of lazy evaluation with C++.

A Generator Function

The following program is as simple as possible. The function `getNumbers` returns all integers from `begin` to `end` incremented by `inc`. `begin` has to be smaller than `end` and `inc` has to be positive.

A greedy generator function

```

1 // greedyGenerator.cpp
2
3 #include <iostream>
4 #include <vector>
5
6 std::vector<int> getNumbers(int begin, int end, int inc = 1){
7
8     std::vector<int> numbers;
9     for (int i = begin; i < end; i += inc){
10         numbers.push_back(i);
11     }
12
13     return numbers;

```

⁸⁷https://en.wikipedia.org/wiki/Melvin_Conway

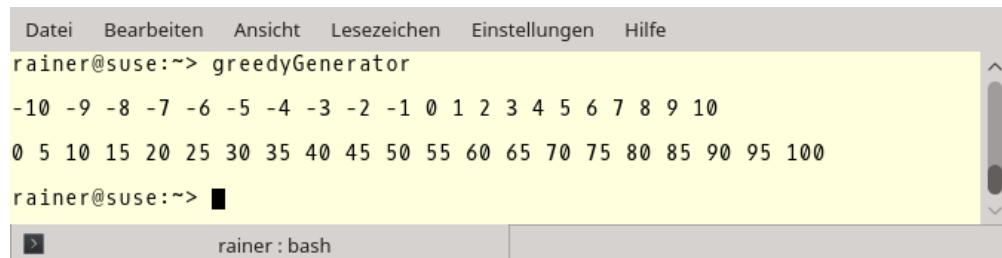
⁸⁸https://en.wikipedia.org/wiki/Donald_Knuth

```

14
15 }
16
17 int main(){
18
19     std::cout << std::endl;
20
21     const auto numbers= getNumbers(-10, 11);
22
23     for (auto n: numbers) std::cout << n << " ";
24
25     std::cout << "\n\n";
26
27     for (auto n: getNumbers(0, 101, 5)) std::cout << n << " ";
28
29     std::cout << "\n\n";
30
31 }
```

Of course, I am reinventing the wheel with `getNumbers` because that job could be done with `std::iota`⁸⁹ since C++11.

For completeness, here is the output.



```

Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe
rainer@suse:~> greedyGenerator
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
rainer@suse:~> █
█          rainer : bash
```

A generator function

Two observations about the program 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.

⁸⁹<http://en.cppreference.com/w/cpp/algorithm/iota>

A lazy generator function

```

1 // lazyGenerator.cpp
2
3 #include <iostream>
4 #include <vector>
5
6 generator<int> generatorForNumbers(int begin, int inc = 1){
7
8     for (int i = begin;; i += inc){
9         co_yield i;
10    }
11 }
12 }
13
14 int main(){
15
16     std::cout << std::endl;
17
18     const auto numbers= generatorForNumbers(-10);
19
20     for (int i= 1; i <= 20; ++i) std::cout << numbers << " ";
21
22     std::cout << "\n\n";
23
24     for (auto n: generatorForNumbers(0, 5)) std::cout << n << " ";
25
26     std::cout << "\n\n";
27
28 }
```

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 18 or `generatorForNumbers(0, 5)` in line 24 returns a new number on request. The range-based for-loop triggers the query. To be more precise, 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 24 is a just-in-place usage 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 24 since there is no end condition. Therefore, the expression runs *forever*.

Because coroutines are a new concept to C++, I want to provide a few details about them.

Details

Typical Use-Cases

Coroutines are the usual way to write [event-driven applications](#)⁹⁰. This can be simulations, games, servers, user interfaces, or even algorithms. Coroutines are typically used for [cooperative multitasking](#)⁹¹. The key to cooperative multitasking is that each task takes as much time as it needs. This is in contrast to pre-emptive multitasking, for which we have a schedule that decides how long each task gets the CPU.

There are different kinds of coroutines.

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. This means that you can use them as an argument to or return value from a function, or store them in a variable.

A **stackless** coroutine enables it to suspend and resume the top-level coroutine, but this coroutine can not invoke another coroutine. Stackless coroutines are often called resumable functions.

Design Goals

Gor Nishanov describes 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](#)⁹², tasks and more.
- usable in environments where exceptions are forbidden or not available.

⁹⁰https://en.wikipedia.org/wiki/Event-driven_programming

⁹¹<https://de.wikipedia.org/wiki/Multitasking>

⁹²<https://tour.golang.org/concurrency/1>

Due to the design goals of scalability and seamless interaction with existing facilities the coroutines are stackless. In contrast, a stackful coroutine reserve a default stack for 1MB on Windows, and 2MB on Linux.

There are four ways for a function to become a coroutine.

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.

This explanation was from proposal N4628.

Finally, I discuss the new keywords `co_return`, `co_yield`, and `co_await`.

`co_return`, `co_yield`, and `co_await`

`co_return`: a coroutine uses `co_return` as its return statement.

`co_yield`: thanks to `co_yield` you can implement a generator. This means you can create 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>`. `generator<int>` internally holds a special promise `p` such that a call `co_yield i` is equivalent to a call `co_await p.yield_value(i)`. `co_yield i` can be called an arbitrary number of times. Immediately after the call, the execution of the coroutine is suspended.

`co_await`: `co_await` eventually causes the execution of the coroutine to be suspended and resumed. The expression `exp` in `co_await exp` has to be a so-called awaitable expression. `exp` has to implement a specific interface. This interface, consisting of the three functions `await_ready`, `await_suspend`, and `wait_resume`.

A typical use case for `co_await` is a server that waits for events.

A blocking server

```
1 Acceptor acceptor{443};  
2 while (true){  
3     Socket socket= acceptor.accept();           // blocking  
4     auto request= socket.read();                // blocking  
5     auto response= handleRequest(request);  
6     socket.write(response);                   // blocking  
7 }
```

The server is quite simple because it sequentially answers each request in the same thread. The server listens on port 443 (line 1), accepts its connections (line 3), reads the incoming data from the client (line 4), and writes its answer to the client (line 6). All 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 }
```

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 Atomicity, Consistency, Isolation, and Durability (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.

ACI(D)

What do atomicity, consistency, and isolation mean for an atomic block consisting of some statements?

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 synchronisation. 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 initial state of the transaction 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 synchronisation. It is only 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 flavours: synchronised blocks and atomic blocks.

Synchronized and Atomic Blocks

So far I only wrote about transactions. Now I write about synchronised blocks and atomic blocks. Both can be encapsulated in each other. To be more specific synchronised blocks are not transactions because they can execute transaction-unsafe. An example for transaction-unsafe code would be code like the output to the console which can not be undone. For this reason, synchronized blocks are often called relaxed blocks.

Synchronized Blocks

Synchronized blocks behave like a global lock protects them. This means that all synchronised blocks follow a total order and in particular: all changes to a synchronised block are available in the next synchronized block. There is a *synchronizes-with* relation between the synchronised blocks because the commit of the transaction *synchronizes-with* the next start of a transaction. Synchronised 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 synchronised block protects the total program.

This is the reason the following program is *well-defined*:

A synchronized block

```

1 // synchronized.cpp
2
3 #include <iostream>
4 #include <vector>
5 #include <thread>
6
7 int i = 0;
8
9 void increment(){
10     synchronized{
11         std::cout << ++i << " ,";
12     }
13 }
14
15 int main(){
16
17     std::cout << std::endl;
18
19     std::vector<std::thread> vecSyn(10);
20     for(auto& thr: vecSyn)
21         thr = std::thread([]{ for(int n = 0; n < 10; ++n) increment(); });
22     for(auto& thr: vecSyn) thr.join();
23 }
```

```
24     std::cout << "\n\n";
25
26 }
```

Although the variable `i` in line 7 is a global variable and the operations in the synchronised block 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. The access to `i` and `std::cout` happens in total order. This is the characteristic of the synchronised 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.



```
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 ,
```

Incrementing with synchronised blocks

What about data races? You can have them with synchronised blocks. A small modification of the source code is sufficient to introduce a [data race](#).

A data race with a synchronized block

```
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 << endl;
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 }
```

To observe the data race, I let the synchronised block sleep for a nanosecond (line 16). At the same time, I access the output stream `std::cout` without a synchronised block (line 30). In total 20 threads increment the global variable `i`, half of them without synchronisation. The output shows the issue.

A data race with synchronised blocks

I put red circles around the issues in the output. These are the locations where `std::cout` is written by at least two threads 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 `i` is written by at least two threads. This is a data race; hence, the program has undefined behaviour. 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 synchronised blocks also holds for atomic blocks.

Atomic Blocks

You can execute transaction-unsafe code in a synchronised 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 cancelled, 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`⁹³, `std::bad_array_length`⁹⁴, `std::bad_array_new_length`⁹⁵, `std::bad_cast`⁹⁶, `std::bad_typeid`⁹⁷, `std::bad_exception`⁹⁸, `std::exception`⁹⁹, and all exceptions that are derived from one of these.

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`

```
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 [N4265¹⁰⁰](#), a function that has a `transaction_safe` definition. This holds true 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 [N4265¹⁰¹](#) for the details.

⁹³http://en.cppreference.com/w/cpp/memory/new/bad_alloc

⁹⁴https://www.cs.helsinki.fi/group/boi2016/doc/cppreference/reference/en.cppreference.com/w/cpp/memory/new/bad_array_length.html

⁹⁵http://en.cppreference.com/w/cpp/memory/new/bad_array_new_length

⁹⁶http://en.cppreference.com/w/cpp/types/bad_cast

⁹⁷http://en.cppreference.com/w/cpp/types/bad_typeid

⁹⁸http://en.cppreference.com/w/cpp/error/bad_exception

⁹⁹<http://en.cppreference.com/w/cpp/error/exception>

¹⁰⁰<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4265.html>

¹⁰¹<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4265.html>

Task Blocks

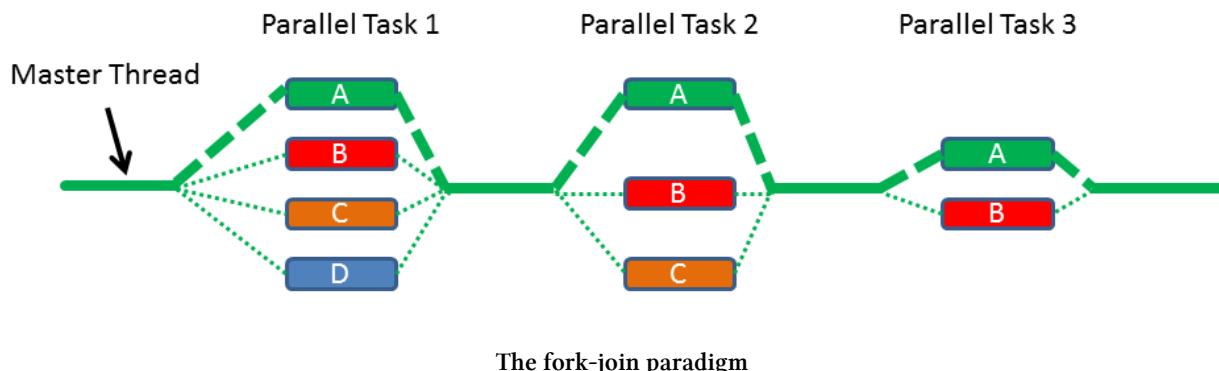
Task blocks use the well-known fork-join paradigm for the parallel execution of tasks. They are already part of the [Technical Specification for C++ Extension Parallelism Version 2](#)¹⁰²; therefore, it's quite probable that we get them with C++20.

Who invented it in C++? Both Microsoft with its [Parallel Patterns Library \(PPL\)](#)¹⁰³ and Intel with its [Threading Building Blocks \(TBB\)](#)¹⁰⁴ were involved in the proposal [N4441](#)¹⁰⁵. Additionally, Intel used its experience with their [Cilk Plus library](#)¹⁰⁶.

The name fork-join is quite easy to explain.

Fork and Join

The most straightforward approach to explain the fork-join paradigm is a graphic.



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 synchronisation is at the end of the task block. The creation of a new task is the fork phase; the synchronisation of the task block is the join phase of the workflow. Admittedly that was simple description. Let's have a look at a piece of code.

¹⁰²<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4742.html>

¹⁰³https://en.wikipedia.org/wiki/Parallel_Patterns_Library

¹⁰⁴https://en.wikipedia.org/wiki/Threading_Building_Blocks

¹⁰⁵<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

¹⁰⁶<https://en.wikipedia.org/wiki/Cilk>

Define a task block

```

1 template <typename Func>
2 int traverse(node& n, Func && f){
3     int left = 0, right = 0;
4     define_task_block(
5         [&](task_block& tb){
6             if (n.left) tb.run([&]{ left = traverse(*n.left, f); });
7             if (n.right) tb.run([&]{ right = traverse(*n.right, f); });
8         }
9     );
10    return f(n) + left + right;
11 }
```

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. Exactly that 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 synchronisation point.



HPX (High Performance ParalleX)

The above example is from the documentation for the [HPX \(High-Performance ParalleX\)¹⁰⁷](#) 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`.

`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.

¹⁰⁷<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 creator thread of the outermost task block (lines 2 - 14) is exactly 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 creator thread of the task block 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). Now it holds that the creator thread executing line 4 is the same thread executing lines 12 and 13.

The Interface

A task block has a very limited 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, therefore, 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 have to be protected from shared access. This is the reason 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`.

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](#)¹⁰⁸ supports both strategies.

¹⁰⁸<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf>

Further Information

Challenges

Programming concurrent applications is inherently complicated. This still holds true 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.

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.

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 full 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.

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

```
1 // fetch_mult.cpp
2
3 #include <atomic>
4 #include <iostream>
5
6 template <typename T>
7 T fetch_mult(std::atomic<T>& shared, T mult){
8     T oldValue = shared.load();
9     while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
10    return oldValue;
11 }
12
13 int main(){
14     std::atomic<int> myInt{5};
15     std::cout << myInt << std::endl;
16     fetch_mult(myInt,5);
17     std::cout << myInt << std::endl;
18 }
```

The key 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 comparison with 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, the 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.

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.

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 behaviour.

There are a few remedies to the ABA problem.

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.

Tagged state reference

You can use the low bits of the address to add a tag to each node indicating how often the node has been successfully modified. The result is that compare and swap (CAS) method eventually fails although the check returns true. This idea does not solve the issue because the tag bits may eventually wrap around. Architectures which support a double word CAS operation can have a bigger counter.

Tagged state reference are typically used in [transactional memory](#).

The next three techniques are based on the idea of deferred reclamation.

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.

Hazard Pointers

From Wikipedia: [Hazard Pointers](#)¹⁰⁹:

In a hazard-pointer system, each thread keeps a list of hazard pointers indicating which nodes the thread is currently accessing. (In many systems this “list” may be 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”.

RCU

RCU stands for Read Copy Update and is a synchronisation 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.

¹⁰⁹https://en.wikipedia.org/wiki/Hazard_pointer

For more details about RCU, read the article [What is RCU, Fundamentally?](#)¹¹⁰ by Paul McKenney.



Two new proposals

As part of a concurrency toolkit there are two proposals for future C++ standards. The proposal [P0233r0](#)¹¹¹ for hazard pointers and the proposal [P0461R0](#)¹¹² for RCU.

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 wakeup.

Here is the program.

Blocking condition variables

```
1 // conditionVariableBlock.cpp
2
3 #include <iostream>
4 #include <condition_variable>
5 #include <mutex>
6 #include <thread>
7
8 std::mutex mutex_;
9 std::condition_variable condVar;
10
11 bool dataReady;
12
13
14 void waitingForWork(){
15
16     std::cout << "Worker: Waiting for work." << std::endl;
17
18     std::unique_lock<std::mutex> lck(mutex_);
19     condVar.wait(lck);
20     // do the work
21     std::cout << "Work done." << std::endl;
```

¹¹⁰<https://lwn.net/Articles/262464/>

¹¹¹<http://www.modernescpp.com/open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0233r0.pdf>

¹¹²<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0461r0.pdf>

```
22
23 }
24
25 void setDataReady(){
26
27     std::cout << "Sender: Data is ready." << std::endl;
28     condVar.notify_one();
29
30 }
31
32 int main(){
33
34     std::cout << std::endl;
35
36     std::thread t1(setDataReady);
37     std::thread t2(waitingForWork);
38
39     t1.join();
40     t2.join();
41
42     std::cout << std::endl;
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).



```
File Edit View Bookmarks Settings Help
rainer@suse:~> conditionVariableBlock
Worker: Waiting for work.
Sender: Data is ready.
Work done.

rainer@suse:~> conditionVariableBlock
Sender: Data is ready.
Worker: Waiting for work.
```

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, with the emphasis on “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](#).

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

```
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) << std::endl;
23 }
24
25 int main(){
26
27     std::cout << std::endl;
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 << std::endl;
45     std::cout << "    acc2.balance: " << acc2.balance << std::endl;
46
47     std::cout << "Final sum: ";
48     printSum(acc1, acc2);
49
50     std::cout << std::endl;
51
52 }
```

At 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

Data Races

A data race is a situation, in which at least two threads access a shared variable at the same time. At least one thread tries to modify the variable.

If your program has a data race, it has undefined behaviour. This means all outcomes are possible and therefore, reasoning about the program makes no sense anymore.

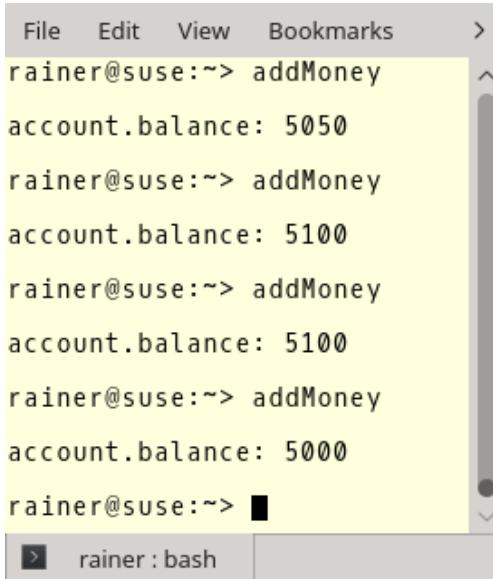
Let me show you a program with a data race.

A data race

```
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 << std::endl;
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 << std::endl;
31
32     std::cout << std::endl;
33
34 }
```

100 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 synchronisation.

Therefore we have a data race, and the result is not valid. This is undefined behaviour, and the final balance (line 30) differs between 5000 and 5100 euro.



A screenshot of a terminal window titled 'rainer : bash'. The window contains the following text:

```
File Edit View Bookmarks >
rainer@suse:~> addMoney
account.balance: 5050
rainer@suse:~> addMoney
account.balance: 5100
rainer@suse:~> addMoney
account.balance: 5100
rainer@suse:~> addMoney
account.balance: 5000
rainer@suse:~> █
```

The terminal window has a yellow background and a vertical scroll bar on the right. The prompt 'rainer@suse:~>' appears four times, each followed by a different balance value: 5050, 5100, 5100, and 5000. The final line ends with a black square character.

A data race causes incorrect balances

Deadlocks

A deadlock is a state in which at least one thread is blocked forever because waits for the release of a resource it does not 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](#)¹¹³ 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](#).

¹¹³<http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/2008/0801/071201hs01/071201hs01.html>



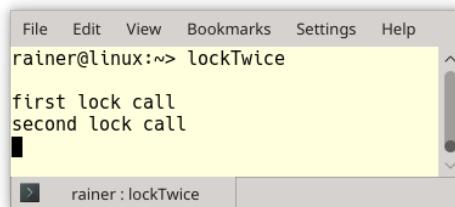
Locking a non-recursive mutex more than once

Locking a non-recursive mutex more than once is undefined behaviour.

Locking more than once

```
1 // lockTwice.cpp
2
3 #include <iostream>
4 #include <mutex>
5
6 int main(){
7
8     std::mutex mut;
9
10    std::cout << std::endl;
11
12    std::cout << "first lock call" << std::endl;
13
14    mut.lock();
15
16    std::cout << "second lock call" << std::endl;
17
18    mut.lock();
19
20    std::cout << "third lock call" << std::endl;
21
22 }
```

Typically, you get a deadlock.



```
File Edit View Bookmarks Settings Help
rainer@linux:~/lockTwice
first lock call
second lock call
```

A deadlock with non-recursive mutexes

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.

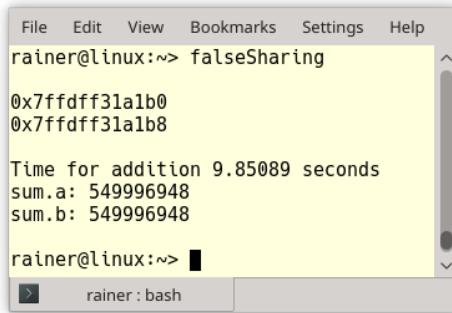
False sharing occurs if two threads read at the same time different variables `a` and `b` that are located on the same cache line. Although `a` and `b` are logically separated, they are physically connected. An expensive hardware synchronisation on the cache line is necessary because `a` and `b` share the same one. 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

```
1 // falseSharing.cpp
2
3 #include <algorithm>
4 #include <chrono>
5 #include <iostream>
6 #include <random>
7 #include <thread>
8 #include <vector>
9
10 constexpr long long size{100000000};
11
12 struct Sum{
13     long long a{0};
14     long long b{0};
15 };
16
17 int main(){
18
19     std::cout << std::endl;
20
21     Sum sum;
22
23     std::cout << &sum.a << std::endl;
24     std::cout << &sum.b << std::endl;
25
26     std::cout << std::endl;
27
28     std::vector<int> randValues, randValues2;
29     randValues.reserve(size);
30     randValues2.reserve(size);
31
32     std::mt19937 engine;
33     std::uniform_int_distribution<> uniformDist(1,10);
34
35     int randValue;
36     for (long long i = 0; i < size; ++i){
```

```
37     randValue = uniformDist(engine);
38     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" << std::endl;
57
58 std::cout << "sum.a: " << sum.a << std::endl;
59 std::cout << "sum.b: " << sum.b << std::endl;
60
61 std::cout << std::endl;
62
63 }
```

The variables `a` and `b` in line 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 output of the program shows the interesting facts. `a` and `b` are aligned at 8-byte boundaries because this is the alignment for `long long` ints on my system.



```
File Edit View Bookmarks Settings Help
rainer@linux:~> falseSharing
0x7ffdff31a1b0
0x7ffdff31a1b8

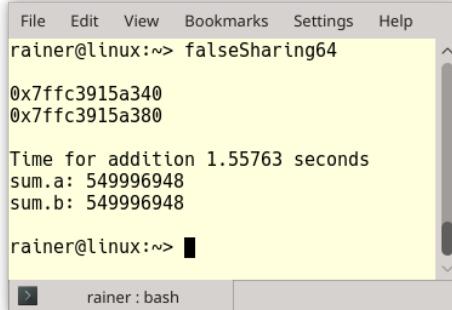
Time for addition 9.85089 seconds
sum.a: 549996948
sum.b: 549996948

rainer@linux:~>
```

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 small 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.

```
struct Sum{
    __alignas(64) long long a{0};
    __alignas(64) long long b{0};
};
```



```
File Edit View Bookmarks Settings Help
rainer@linux:~> falseSharing64
0x7ffc3915a340
0x7ffc3915a380

Time for addition 1.55763 seconds
sum.a: 549996948
sum.b: 549996948

rainer@linux:~>
```

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 optimiser detects the false sharing

If I compile the last programs with maximum optimisation, my optimiser 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 optimised numbers.



```
File Edit View Bookmarks Settings Help
rainer@linux:~> falseSharing
0x7ffd319699e0
0x7ffd319699e8

Time for addition 0.0793453 seconds
sum.a: 549996948
sum.b: 549996948

rainer@linux:~> falseSharing64
0x7ffc4db6aa40
0x7ffc4db6aa80

Time for addition 0.079384 seconds
sum.a: 549996948
sum.b: 549996948

rainer@linux:~> 
```

False sharing resolved by the optimiser



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.

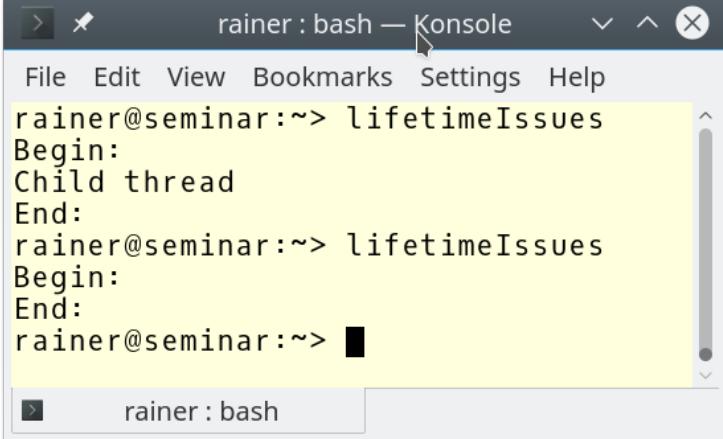
Lifetime Issues of Variables

Creating a C++ example with lifetime related issues is quite 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 don'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

```
1 // lifetimeIssues.cpp
2
3 #include <iostream>
4 #include <string>
5 #include <thread>
6
7 int main(){
8
9     std::cout << "Begin:" << std::endl;
10
11    std::string mess{"Child thread"};
12
13    std::thread t([&mess]{ std::cout << mess << std::endl; });
14    t.detach();
15
16    std::cout << "End:" << std::endl;
17
18 }
```

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.



The screenshot shows a terminal window titled "rainer : bash — Konsole". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The command "lifetimeIssues" is run twice. The first run shows the output "Begin:", "Child thread", and "End:". The second run shows the output "Begin:" and "End:". The terminal window has a scroll bar on the right and a status bar at the bottom showing "rainer : bash".

Lifetime issues of variables

Moving Threads

Moving threads make the [lifetime issues](#) of threads even harder.

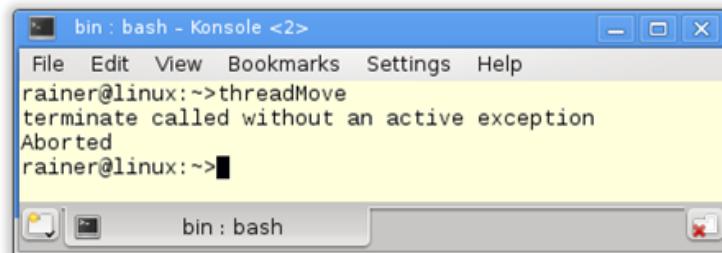
A thread supports the move semantic but not the copy semantic. The reason being the copy constructor of `std::thread` is set to `delete thread(const thread&) = delete;`. Imagine what happens if you copy a thread while the thread is holding a lock.

Let's move a thread.

Erroneous moving a thread

```
1 // threadMoved.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();});
10    std::thread t2([]{std::cout << std::this_thread::get_id();});
11
12    t = std::move(t2);
13    t.join();
14    t2.join();
15
16 }
```

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:



Erroneous moving a thread

What is going wrong? We have two issues:

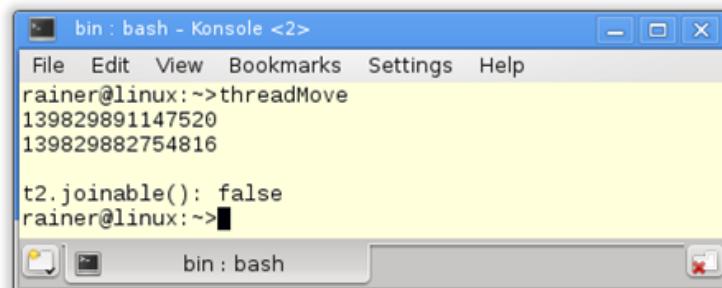
1. By moving the thread `t2`, `t` gets a new callable unit and its destructor is called. As a result `t`'s destructor calls `std::terminate`, because it is still joinable.
2. Thread `t2` has no associated callable unit. The invocation of `join` on a thread without callable unit leads to the exception `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() << std::endl;});
10    std::thread t2([]{std::cout << std::this_thread::get_id() << std::endl;});
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() << std::endl;
18
19 }
```

The result is that thread t2 is not joinable anymore.



```
rainer@linux:~/threadMove
139829891147520
139829882754816

t2.joinable(): false
rainer@linux:~>
```

Erroneous moving resolved

Race Conditions

A race condition is a situation, in which the result of an operation depends on the interleaving of certain individual operations.

Race conditions are quite difficult to spot. It depends on the interleaving of the threads whether they occur. That means the number of cores, the utilisation of your system, or the optimisation level of your executable 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](#).

Best Practices

This chapter provides you with a simple set of rules for writing well-defined and fast, concurrent programs in modern C++. Multithreading, and in particular 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 over time. This holds particularly true for the parallel STL. At the time of updating this book (05/2018), the parallel algorithms of C++17 are not available; therefore, it is too early to formulate best practices for it.

General

Let's start with a few very general best practices that applies to atomics and threads.

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 analyse 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

```
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}};
```

```
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 << std::endl;
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 << std::endl;
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 << std::endl;
54 }
```

```
55     std::cout << "\nThe new telephone book" << std::endl;
56     for (auto teleIt: teleBook){
57         std::cout << teleIt.first << ":" << teleIt.second << std::endl;
58     }
59
60     std::cout << std::endl;
61
62 }
```

The issue is that the call `teleBook[na]` in 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 an exercise. The exercise is to spot the data race. About 10% of the participants find the data race within 5 minutes.

Minimise Sharing of mutable data

You should minimise 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 a exhaustive performance study. How fast can I sum up the values of a `std::vector`?

This was the key part of the single threaded summation.

Single threaded summation

Afterwards, 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];
    }
}
```

...

Optimised 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

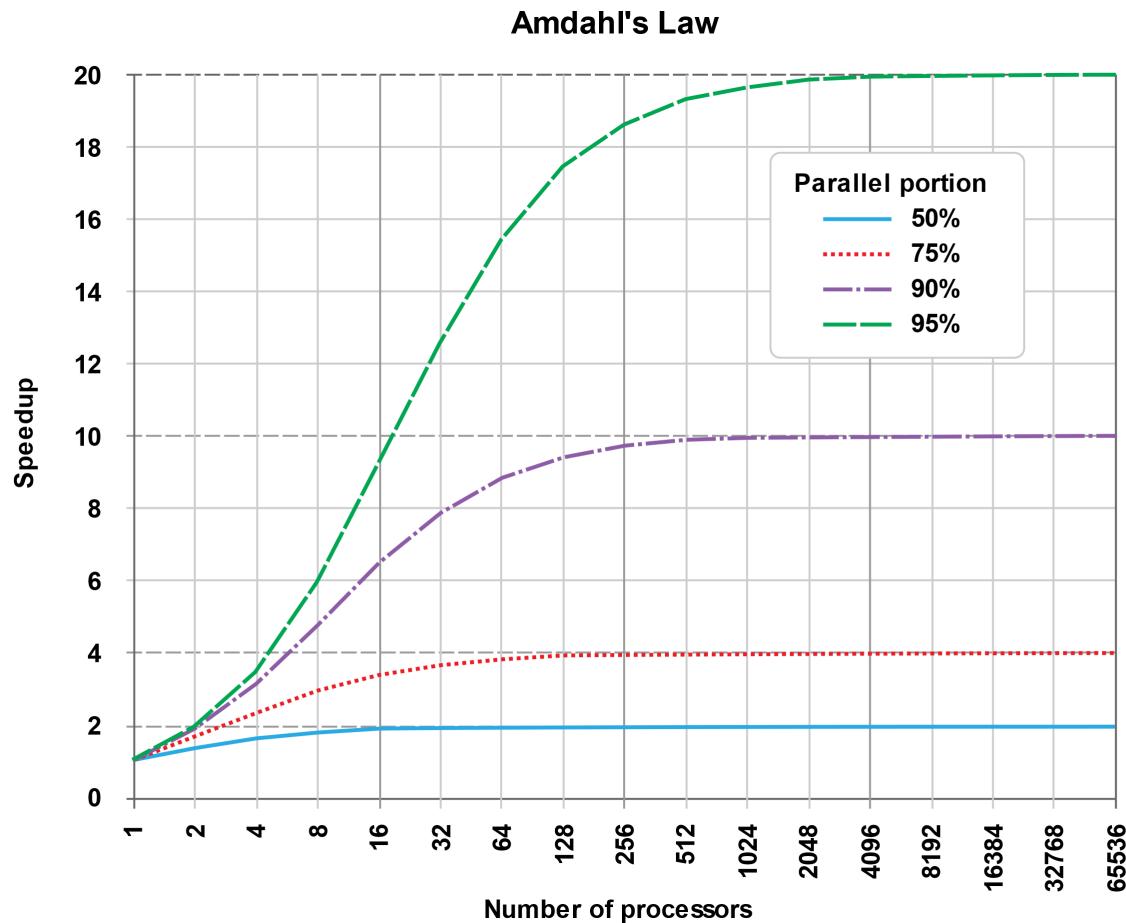
Minimise Waiting

You may have heard of [Amdahl's law](#)¹¹⁴. 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.

¹¹⁴https://en.wikipedia.org/wiki/Amdahl%27s_law



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 your program not faster. If you have 95% parallel code, you reach the peak performance with 2048 cores.

Prefer Immutable Data

A data race is a situation, in which at least two threads access a shared variable at the same time. At least one thread tries to modify the variable. The definition makes it quite obvious. A necessary condition for a data race is mutable, shared state. The graphic makes my point clear.

		Mutable?	
		no	yes
Shared?	no	OK	Ok
	yes	OK	Data Race

Mutable and shared state

If you have immutable data, no data race can happen. You only have to guarantee that the immutable data are initialised in a thread-safe way. I presented in the chapter [Thread-safe initialisation](#) four ways to guarantee this. Here are they:

- early initialisation 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

Functional programming languages such as Haskell, having no mutable data, are very suitable for concurrent programming.

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.

```
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.

```

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.

```

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.

Look for the Right Abstraction

There are various ways to [initialise 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, to 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 parametrise `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.

Use Static Code Analysis Tools

In the chapter on case studies, I introduced [CppMem](#). CppMem¹¹⁵ is an interactive tool for exploring the behaviour 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 multithreading issues in general.

¹¹⁵<http://svr-pes20-ccppmem.cl.cam.ac.uk/CppMem/>

Use Dynamic Enforcement Tools

ThreadSanitizer¹¹⁶ 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`

```
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 }
```

`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.

¹¹⁶<https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual>



```

File Edit View Bookmarks Settings Help
rainer@suse:~> dataRace
=====
WARNING: ThreadSanitizer: data race (pid=6764)
  Read of size 4 at 0x7fff031ca3bc by thread T2:
#0 operator() /home/rainer/dataRace.cpp:10 (dataRace+0x000000400f01)
#1 __invoke<> /usr/local/include/c++/6.3.0/functional:1391 (dataRace+0x000000401b3d)
#2 operator() /usr/local/include/c++/6.3.0/functional:1380 (dataRace+0x000000401a51)
#3 __run /usr/local/include/c++/6.3.0/thread:196 (dataRace+0x0000004019bc)
#4 execute_native_thread_routine ../../../../libstdc++-v3/src/c++11/thread.cc:83 (libstdc++.so.6+0x0000000
0c1be)

  Previous write of size 4 at 0x7fff031ca3bc by thread T1:
#0 operator() /home/rainer/dataRace.cpp:9 (dataRace+0x000000400eb9)
#1 __invoke<> /usr/local/include/c++/6.3.0/functional:1391 (dataRace+0x000000401be7)
#2 operator() /usr/local/include/c++/6.3.0/functional:1380 (dataRace+0x000000401a8b)
#3 __run /usr/local/include/c++/6.3.0/thread:196 (dataRace+0x000000401a06)
#4 execute_native_thread_routine ../../../../libstdc++-v3/src/c++11/thread.cc:83 (libstdc++.so.6+0x0000000
0c1be)

  Location is stack of main thread.

  Thread T2 (tid=6767, running) created by main thread at:
#0 pthread_create ../../../../libsanitizer/tsan/tsan_interceptors.cc:876 (libtsan.so.0+0x0000002aaed)
#1 __gthread_create /home/rainer/languages/C++/gcc-6.3.0/x86_64-pc-linux-gnu/libstdc++-v3/include/x86_64-pc
-linux-gnu/bits/gthr-default.h:662 (libstdc++.so.6+0x0000000c14b4)
#2 std::thread::_M_start_thread(std::unique_ptr<std::thread::State, std::default_delete<std::thread::Stat
e> >, void (*)()) ../../../../libstdc++-v3/src/c++11/thread.cc:163 (libstdc++.so.6+0x0000000c14b4)
#3 main /home/rainer/dataRace.cpp:10 (dataRace+0x000000400f97)

  Thread T1 (tid=6766, finished) created by main thread at:
#0 pthread_create ../../../../libsanitizer/tsan/tsan_interceptors.cc:876 (libtsan.so.0+0x0000002aaed)
#1 __gthread_create /home/rainer/languages/C++/gcc-6.3.0/x86_64-pc-linux-gnu/libstdc++-v3/include/x86_64-pc
-linux-gnu/bits/gthr-default.h:662 (libstdc++.so.6+0x0000000c14b4)
#2 std::thread::_M_start_thread(std::unique_ptr<std::thread::State, std::default_delete<std::thread::Stat
e> >, void (*)()) ../../../../libstdc++-v3/src/c++11/thread.cc:163 (libstdc++.so.6+0x0000000c14b4)
#3 main /home/rainer/dataRace.cpp:9 (dataRace+0x000000400f70)

SUMMARY: ThreadSanitizer: data race /home/rainer/dataRace.cpp:10 in operator()

ThreadSanitizer: reported 1 warnings
rainer@suse:~>
```

A data race detected with ThreadSanitizer

I highlighted in red the critical line of the screenshot. There is a data race on line 10.

Multithreading

Threads

Threads are the basic building blocks for writing concurrent programs.

Minimise 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 than about the costs of its creation.

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](#)¹¹⁷ because most of the times they are internally used.

¹¹⁷https://en.wikipedia.org/wiki/POSIX_Threads

- Windows systems: the post [Thread Stack Size¹¹⁸](#) gave me the answer: 1 MB.
- POSIX systems: the [pthread_create¹¹⁹](#) man-page provides me with the answer: 2MB. This is the sizes for the i386 and x86_64 architectures. If you want to know the sizes for further 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`

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 optimisation. This means on Linux the flag `O3` and on Windows `Ox`.

Here is my small test program.

¹¹⁸[https://msdn.microsoft.com/en-us/library/windows/desktop/ms686774\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms686774(v=vs.85).aspx)

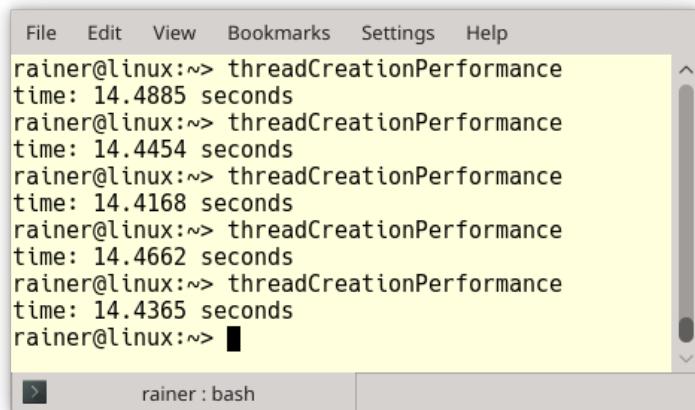
¹¹⁹http://man7.org/linux/man-pages/man3/pthread_create.3.html

A small performance test for thread creation

```
1 // threadCreationPerformance.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <thread>
6
7 static const long long numThreads= 1'000'000;
8
9 int main(){
10
11     auto start = std::chrono::system_clock::now();
12
13     for (volatile int i = 0; i < numThreads; ++i) std::thread([]{}).detach();
14
15     std::chrono::duration<double> dur= std::chrono::system_clock::now() - start;
16     std::cout << "time: " << dur.count() << " seconds" << std::endl;
17
18 }
```

The program creates 1 million threads which execute the empty lambda function in line 13. These are the numbers for Linux and Windows:

Linux

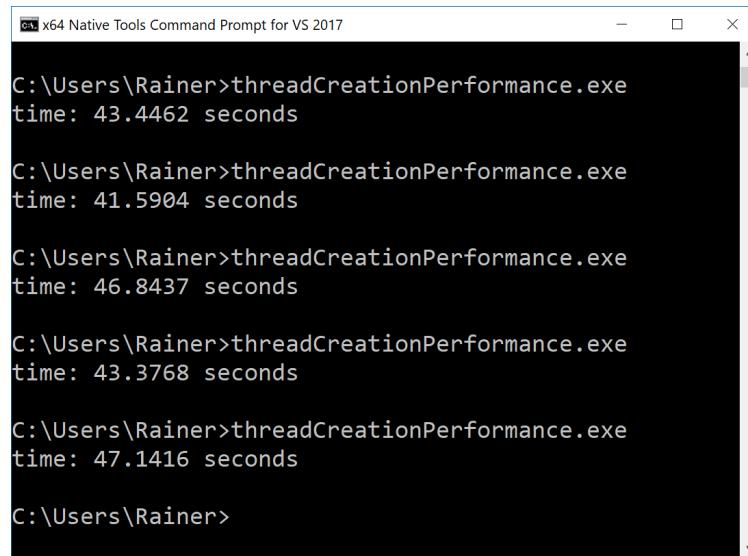


```
File Edit View Bookmarks Settings Help
rainer@linux:~> threadCreationPerformance
time: 14.4885 seconds
rainer@linux:~> threadCreationPerformance
time: 14.4454 seconds
rainer@linux:~> threadCreationPerformance
time: 14.4168 seconds
rainer@linux:~> threadCreationPerformance
time: 14.4662 seconds
rainer@linux:~> threadCreationPerformance
time: 14.4365 seconds
rainer@linux:~> █
```

Thread creation on Linux

This means that the creation of a thread took about $14.5 \text{ sec} / 1000000 = 14.5 \text{ microseconds}$ on Linux.

Windows



```
C:\Users\Rainer>threadCreationPerformance.exe
time: 43.4462 seconds

C:\Users\Rainer>threadCreationPerformance.exe
time: 41.5904 seconds

C:\Users\Rainer>threadCreationPerformance.exe
time: 46.8437 seconds

C:\Users\Rainer>threadCreationPerformance.exe
time: 43.3768 seconds

C:\Users\Rainer>threadCreationPerformance.exe
time: 47.1416 seconds

C:\Users\Rainer>
```

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**.

Use tasks instead of threads

std::async versus threads

```
1 // asyncVersusThread.cpp
2
3 #include <future>
4 #include <thread>
5 #include <iostream>
6
7 int main(){
8
9     std::cout << std::endl;
10
11    int res;
12    std::thread t([&]{ res = 2000 + 11; });
13    t.join();
14    std::cout << "res: " << res << std::endl;
15
16    auto fut= std::async([]{ return 2000 + 11; });
17    std::cout << "fut.get(): " << fut.get() << std::endl;
```

```
18
19     std::cout << std::endl;
20
21 }
```

Based on the program there are many reasons for preferring tasks over threads. The main reasons are:

- you can use a safe communication channel for returning the result of the communication. If you use a shared variable, you have to synchronise 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`.

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 << std::endl; });
t.detach();
```

Because thread `t` is detached from the lifetime of its creator, two [race condition](#) can cause undefined behaviour.

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 lifetime of the main thread.

Consider using an automatic joining thread

A thread `t` with a [callable unit](#) is called joinable if neither `t.join()` nor a `t.detach()` call happened. The destructor of a joinable thread throws the `std::terminate` exception. In order not to 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 given thread in the destructor.

You don't have to build this wrapper by your own. Use the [scoped_thread](#) from Anthony Williams or the [gs1::joining_thread](#) from the [guideline support library](#)¹²⁰.

¹²⁰<https://github.com/Microsoft/GSL>

Data Sharing

With data sharing of mutable data the challenges in multithreading programming start.

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 synchronise the access to `s` in the creator thread and the created thread `t2` preventively. This is error-prone and expensive.

Use `std::shared_ptr` to share ownership between unrelated threads

Imagine, you have a object which you want to share between unrelated threads. The key question is, who is the owner of the object and, therefore, responsible for releasing the memory? Now you can choose between a memory leak if you don't deallocate the memory or undefined behaviour because you invoked delete more than once. Most of the times, the undefined behaviour 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" << std::endl;
```

```
12     }
13 };
14
15 void showNumber(MyInt* myInt){
16     std::cout << myInt->val << std::endl;
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 << std::endl;
32
33     threadCreator();
34     std::this_thread::sleep_for(1s);
35
36     std::cout << std::endl;
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 synchronisation, but it helps me to make my point. The vital issue of the program is: Who is responsible for the deletion of tmpInt in 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:



```
File Edit View Bookmarks Settings Help
rainer@linux:~> threadSharesOwnership
2017
2017
rainer@linux:~>
> rainer : bash
```

Unrelated threads share ownership

The lifetime issues are quite easy to handle if I use a `std::shared_ptr`.



```
File Edit View Bookmarks Settings Help
rainer@linux:~> threadSharesOwnershipSharedPtr
2017
2017
Good Bye
rainer@linux:~>
> rainer : bash
```

Unrelated threads share ownership via `std::shared_ptr`

Unrelated threads share ownership via `std::shared_ptr`

```
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" << std::endl;
13     }
14 };
15
16 void showNumber(std::shared_ptr<MyInt> myInt){
17     std::cout << myInt->val << std::endl;
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 << std::endl;
33
34     threadCreator();
35     std::this_thread::sleep_for(1s);
36
37     std::cout << std::endl;
38
39 }
```

Two small 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.

Minimise 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" << std::endl;
    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" << std::endl;
    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 synchronised 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`.

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 lifetime of the lock automatically ends; therefore, the underlying mutex is unlocked.

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.

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 synchronisation. 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 behaviour:

```
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" << std::endl;  
13     }  
14     ~MyGuard(){  
15         myMutex.unlock();  
16         std::cout << "unlock" << std::endl;  
17     }  
18 };  
19  
20 int main(){  
21  
22     std::cout << std::endl;  
23  
24     std::mutex m;  
25     MyGuard<std::mutex> {m};  
26     std::cout << "CRITICAL SECTION" << std::endl;
```

```

27
28     std::cout << std::endl;
29
30 }
```

The `MyGuard` calls `lock` and `unlock` in its constructor and in its destructor. Because of the temporary, the call to the constructor and destructor happens in line 25. In particular, this means that the call of the destructor happens at line 25 and not, as usual, in line 31. As consequence, the critical section in line 26 is executed without synchronisation.

This screenshot of the program shows that the output of `unlock` happens before the output of `CRITICAL SECTION`.

Locks having no name

Use `std::lock` or `std::scoped_lock` for locking more mutexes atomically

If a thread needs more than one mutex, you have to be extremely careful that you lock the mutex always in the same sequence. If not, a bad interleaving of threads may cause a [deadlock](#).

```

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 synchronise 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);});

...
```

Never call unknown code while holding a lock

Calling an `unknownFunction` while holding a mutex is a recipe for undefined behaviour.

```
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`, that is undefined behaviour. 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 synchronisation mechanism. Additionally, the time for holding a lock is reduced to its bare minimum: assigning the value of `tempVar` to `sharedVariable`.

Condition Variables

Synchronising 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](#)).

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

```
1 // conditionVariableLostWakeup.cpp
2
3 #include <condition_variable>
4 #include <mutex>
5 #include <thread>
6
7 std::mutex mutex_;
8 std::condition_variable condVar;
9
10 void waitingForWork(){
11     std::unique_lock<std::mutex> lck(mutex_);
12     condVar.wait(lck);
13     // do the work
14 }
15
16 void setDataReady(){
17     condVar.notify_one();
18 }
19
20 int main(){
21
22     std::thread t1(setDataReady);
23     std::thread t2(waitingForWork);
24
25     t1.join();
26     t2.join();
27
28 }
```

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 protect against a **spurious wakeup** because of the waiting thread checks at first if the notification was from the right thread.

Condition variables with a predicate

```
1 // conditionVariableLostWakeupSolved.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 }
```

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.

Promises and Futures

Promises and futures can often be used as an easy-to-use replacement for threads or condition variables.

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() << std::endl;
```

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](#)¹²¹. You are only interested in picking up the result in the future: `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.

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.

Don’t use volatile for synchronisation

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 optimised storing in caches takes place.

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](#).

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 synchronisation. You shift the application challenges to the infrastructure.

¹²¹https://en.wikipedia.org/wiki/Graphics_processing_unit

Don't build your abstraction, use guarantees of the language

Thread-safe initialisation 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.

Don't reinvent the wheel

Writing thread-safe data structures is a quite challenging endeavour. Writing lock-free data-structures is way harder; therefore, use existing libraries such as [Boost.Lockfree](#)¹²² or [CDS](#)¹²³.

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)

CDS

CDS stands for Concurrent Data Structures and contains a lot of 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)

¹²²http://www.boost.org/doc/libs/1_66_0/doc/html/lockfree.html

¹²³<http://libcds.sourceforge.net/>

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.

The Interplay of Time Point, Time Duration, and Clock

Time point

The time point is given by its starting point - the so-called epoch¹²⁴ - 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.

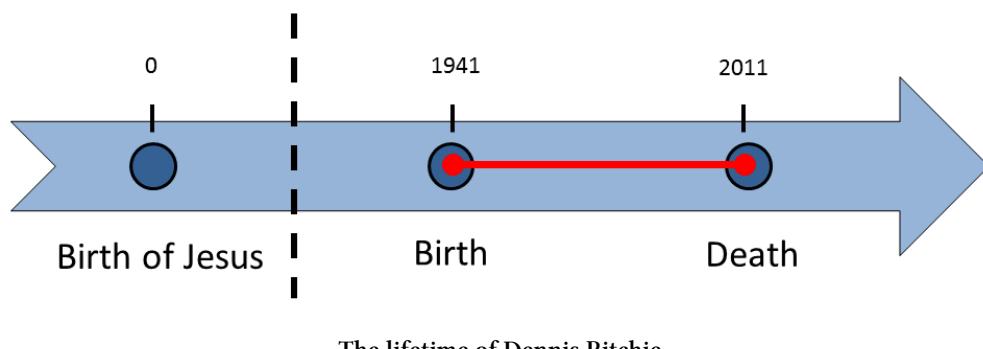
Clock

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 Ritchie¹²⁵. The creator of C who died in 2011. For the sake of simplicity, I'm only interested in the years.

Here is the lifetime.



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 time 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.

¹²⁴[https://en.wikipedia.org/wiki/Epoch_\(reference_date\)](https://en.wikipedia.org/wiki/Epoch_(reference_date))

¹²⁵https://en.wikipedia.org/wiki/Dennis_Ritchie

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 the 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**¹²⁶. It holds further that `std::chrono::high_resolution_clock` has the highest accuracy.

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`¹²⁷ gives you the calendar time, expressed in **Coordinated Universal Time**¹²⁸ (UTC). In the end this calendar time can be used as the input for the function `std::asctime`¹²⁹ to get a textual representation of the calendar time.

¹²⁶https://en.wikipedia.org/wiki/Unix_time

¹²⁷<http://en.cppreference.com/w/cpp/chrono/c/gmtime>

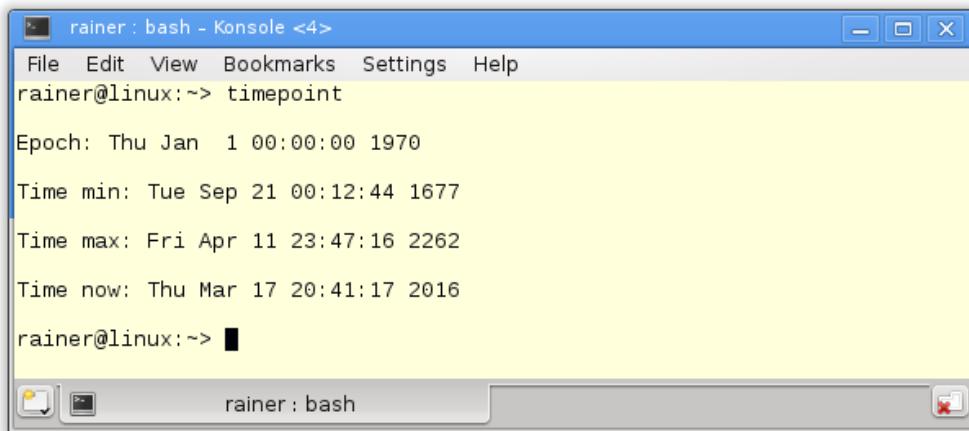
¹²⁸https://en.wikipedia.org/wiki/Coordinated_Universal_Time

¹²⁹<http://en.cppreference.com/w/cpp/chrono/c/asctime>

Display the calendar time

```
1 // timepoint.cpp
2
3 #include <chrono>
4 #include <ctime>
5 #include <iostream>
6 #include <string>
7
8 int main(){
9
10    std::cout << std::endl;
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 << std::endl;
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 << std::endl;
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 << std::endl;
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 << std::endl;
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.



```
rainer@linux:~> timepoint
Epoch: Thu Jan 1 00:00:00 1970
Time min: Tue Sep 21 00:12:44 1677
Time max: Fri Apr 11 23:47:16 2262
Time now: Thu Mar 17 20:41:17 2016
rainer@linux:~> █
```

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 behaviour.

Cross the valid Time Range

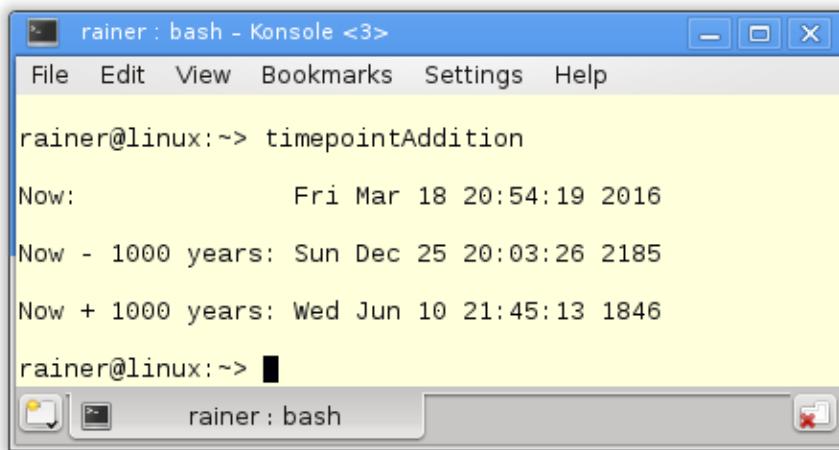
The following example uses the current time and adds or subtracts 1000 years. For the sake of simplicity, I ignore leap years and assume that a year has 365 days.

Crossing the valid time range

```
1 // timepointAddition.cpp
2
3 #include <chrono>
4 #include <ctime>
5 #include <iostream>
6 #include <string>
7
8 using namespace std::chrono;
9 using namespace std;
10
11 string timePointAsString(const time_point<system_clock>& timePoint){
12     time_t tp= system_clock::to_time_t(timePoint);
13     return asctime(gmtime(&tp));
14 }
15
16 int main(){
17
18     cout << endl;
19
20     time_point<system_clock> nowTimePoint= system_clock::now();
```

```
21 cout << "Now: " << timePointAsString(nowTimePoint) << endl;
22
23 const auto thousandYears= hours(24*365*1000);
24 time_point<system_clock> historyTimePoint= nowTimePoint - thousandYears;
25 cout << "Now - 1000 years: " << timePointAsString(historyTimePoint) << endl;
26
27 time_point<system_clock> futureTimePoint= nowTimePoint + thousandYears;
28 cout << "Now + 1000 years: " << timePointAsString(futureTimePoint) << endl;
29
30 }
```

For readability, I introduced the namespace `std::chrono`. The output of the program 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.



```
rainer@linux:~> timepointAddition
Now: Fri Mar 18 20:54:19 2016
Now - 1000 years: Sun Dec 25 20:03:26 2185
Now + 1000 years: Wed Jun 10 21:45:13 1846
rainer@linux:~>
```

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.

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`

```
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

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

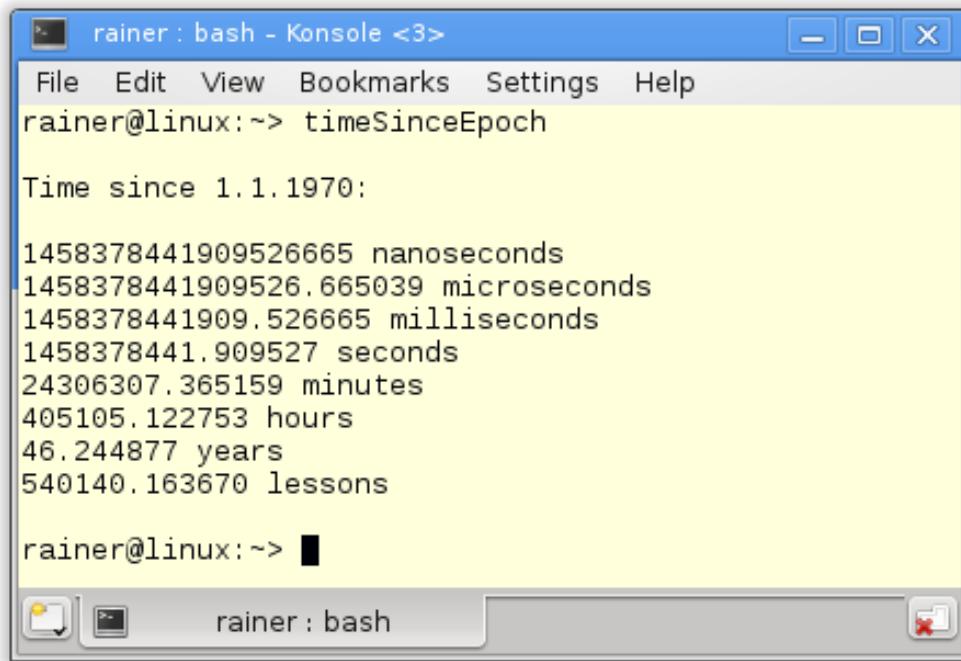
How much time has passed since the UNIX epoch (1.1.1970)? Thanks to type aliases for the different time durations, I can answer the question quite easily. In the following example, I ignore leap years and assume that a year has 365 days.

Crossing the valid time range

```
1 // timeSinceEpoch.cpp
2
3 #include <chrono>
4 #include <iostream>
5
6 using namespace std;
7
8 int main(){
9
10    cout << fixed << endl;
11}
```

```
12     cout << "Time since 1.1.1970:\n" << endl;
13
14     const auto timeNow= chrono::system_clock::now();
15     const auto duration= timeNow.time_since_epoch();
16     cout << duration.count() << " nanoseconds " << endl;
17
18     typedef chrono::duration<long double, ratio<1, 1000000>> MyMicroSecondTick;
19     MyMicroSecondTick micro(duration);
20     cout << micro.count() << " microseconds" << endl;
21
22     typedef chrono::duration<long double, ratio<1, 1000>> MyMilliSecondTick;
23     MyMilliSecondTick milli(duration);
24     cout << milli.count() << " milliseconds" << endl;
25
26     typedef chrono::duration<long double> MySecondTick;
27     MySecondTick sec(duration);
28     cout << sec.count() << " seconds " << endl;
29
30     typedef chrono::duration<double, ratio<60>> MyMinuteTick;
31     MyMinuteTick myMinute(duration);
32     cout << myMinute.count() << " minutes" << endl;
33
34     typedef chrono::duration<double, ratio<60*60>> MyHourTick;
35     MyHourTick myHour(duration);
36     cout << myHour.count() << " hours" << endl;
37
38     typedef chrono::duration<double, ratio<60*60*24*365>> MyYearTick;
39     MyYearTick myYear(duration);
40     cout << myYear.count() << " years" << endl;
41
42     typedef chrono::duration<double, ratio<60*45>> MyLessonTick;
43     MyLessonTick myLesson(duration);
44     cout << myLesson.count() << " lessons" << endl;
45
46     cout << endl;
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.



```
rainer : bash - Konsole <3>
File Edit View Bookmarks Settings Help
rainer@linux:~> timeSinceEpoch

Time since 1.1.1970:

1458378441909526665 nanoseconds
1458378441909526.665039 microseconds
1458378441909.526665 milliseconds
1458378441.909527 seconds
24306307.365159 minutes
405105.122753 hours
46.244877 years
540140.163670 lessons

rainer@linux:~> █
```

Time since epoch

It's quite convenient to calculate with time durations, as the next section illustrates.

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 emphasise 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

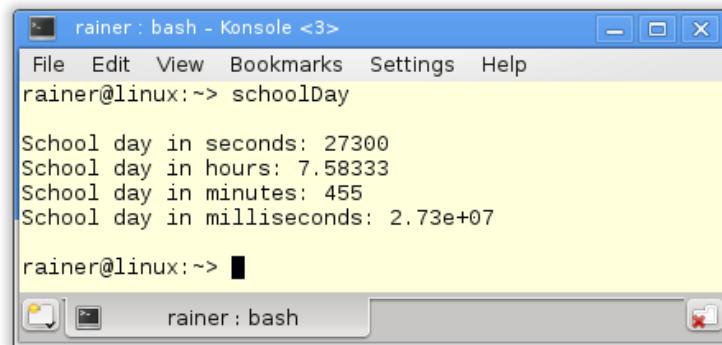
Type	Suffix	Example
std::chrono::hours	h	5h
std::chrono::minutes	min	5min
std::chrono::seconds	s	5s
std::chrono::milliseconds	ms	5ms
std::chrono::microseconds	us	5us
std::chrono::nanoseconds	ns	5ns

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

```
1 // schoolDay.cpp
2
3 #include <iostream>
4 #include <chrono>
5
6 using namespace std::literals::chrono_literals;
7 using namespace std::chrono;
8 using namespace std;
9
10 int main(){
11
12     cout << endl;
13
14     constexpr auto schoolHour= 45min;
15
16     constexpr auto shortBreak= 300s;
17     constexpr auto longBreak= 0.25h;
18
19     constexpr auto schoolWay= 15min;
20     constexpr auto homework= 2h;
21
22     constexpr auto schoolDaySec= 2*schoolWay + 6 * schoolHour + 4 * shortBreak +
23                         longBreak + homework;
24
25     cout << "School day in seconds: " << schoolDaySec.count() << endl;
26
27     constexpr duration<double, ratio<3600>> schoolDayHour = schoolDaySec;
28     constexpr duration<double, ratio<60>> schoolDayMin = schoolDaySec;
29     constexpr duration<double, ratio<1,1000>> schoolDayMilli= schoolDaySec;
30
31     cout << "School day in hours: " << schoolDayHour.count() << endl;
32     cout << "School day in minutes: " << schoolDayMin.count() << endl;
33     cout << "School day in milliseconds: " << schoolDayMilli.count() << endl;
34
35     cout << endl;
36
37 }
```

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 way to school (line 19), and his homework (line 20). The result of the calculation `schoolDaysInSeconds` (line 22) is available at compile time.



```
rainer : bash - Konsole <3>
File Edit View Bookmarks Settings Help
rainer@linux:~> schoolDay
School day in seconds: 27300
School day in hours: 7.58333
School day in minutes: 455
School day in milliseconds: 2.73e+07
rainer@linux:~> █
```

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`.

Clocks

The fact that there are three different types of clocks begs the question: What are the differences?

- `std::chrono::system_clock`: is the system-wide real time clock ([wall-clock¹³⁰](#)). 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 typically the boot time of your PC.

Accuracy and Steadiness

It is quite 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 " << endl;
13     typedef typename ratio_multiply<T,kilo>::type MillSec;
14     typedef typename ratio_multiply<T,mega>::type MicroSec;

```

¹³⁰https://en.wikipedia.org/wiki/Wall-clock_time

```

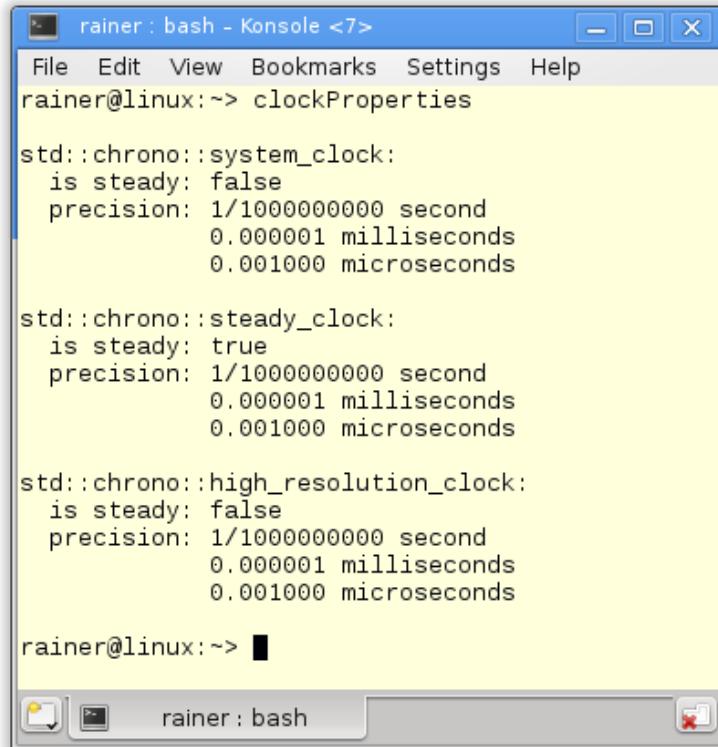
15     cout << fixed;
16     cout << "           " << static_cast<double>(MillSec::num)/MillSec::den
17             << " milliseconds " << endl;
18     cout << "           " << static_cast<double>(MicroSec::num)/MicroSec::den
19             << " microseconds " << endl;
20 }
21
22 int main(){
23
24     cout << boolalpha << endl;
25
26     cout << "std::chrono::system_clock: " << endl;
27     cout << "  is steady: " << system_clock::is_steady << endl;
28     printRatio<chrono::system_clock::period>();
29
30     cout << endl;
31
32     cout << "std::chrono::steady_clock: " << endl;
33     cout << "  is steady: " << chrono::steady_clock::is_steady << endl;
34     printRatio<chrono::steady_clock::period>();
35
36     cout << endl;
37
38     cout << "std::chrono::high_resolution_clock: " << endl;
39     cout << "  is steady: " << chrono::high_resolution_clock::is_steady
40             << endl;
41     printRatio<chrono::high_resolution_clock::period>();
42
43     cout << endl;
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](http://en.cppreference.com)¹³¹.

The output on Linux differs from that on Windows. `std::chrono::system_clock` is far more accurate on Linux; `std::chrono::high_resolution_clock` is steady on Windows.

¹³¹<http://en.cppreference.com/w/cpp/numeric/ratio>



```
rainer : bash - Konsole <7>
File Edit View Bookmarks Settings Help
rainer@linux:~/> clockProperties

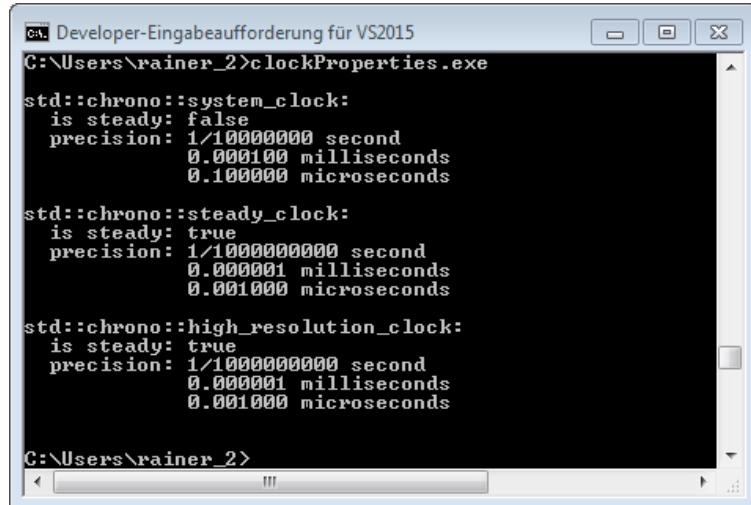
std::chrono::system_clock:
  is_steady: false
  precision: 1/10000000000 second
              0.000001 milliseconds
              0.001000 microseconds

std::chrono::steady_clock:
  is_steady: true
  precision: 1/10000000000 second
              0.000001 milliseconds
              0.001000 microseconds

std::chrono::high_resolution_clock:
  is_steady: false
  precision: 1/10000000000 second
              0.000001 milliseconds
              0.001000 microseconds

rainer@linux:~/> █
```

Accuracy and steadiness of the three clocks on Linux



```
Developer-Eingabeaufforderung für VS2015
C:\Users\rainer_2>clockProperties.exe

std::chrono::system_clock:
  is_steady: false
  precision: 1/10000000 second
              0.000100 milliseconds
              0.100000 microseconds

std::chrono::steady_clock:
  is_steady: true
  precision: 1/10000000000 second
              0.000001 milliseconds
              0.001000 microseconds

std::chrono::high_resolution_clock:
  is_steady: true
  precision: 1/10000000000 second
              0.000001 milliseconds
              0.001000 microseconds

C:\Users\rainer_2>
```

Accuracy and steadiness of different clocks on Windows

Although the C++ standard doesn't specify the epoch of the clock, you can calculate it.

Epoch

Thanks to the auxiliary function `time_since_epoch`¹³², each clock returns how much time has passed since the epoch.

Calculating the epoch for each clock

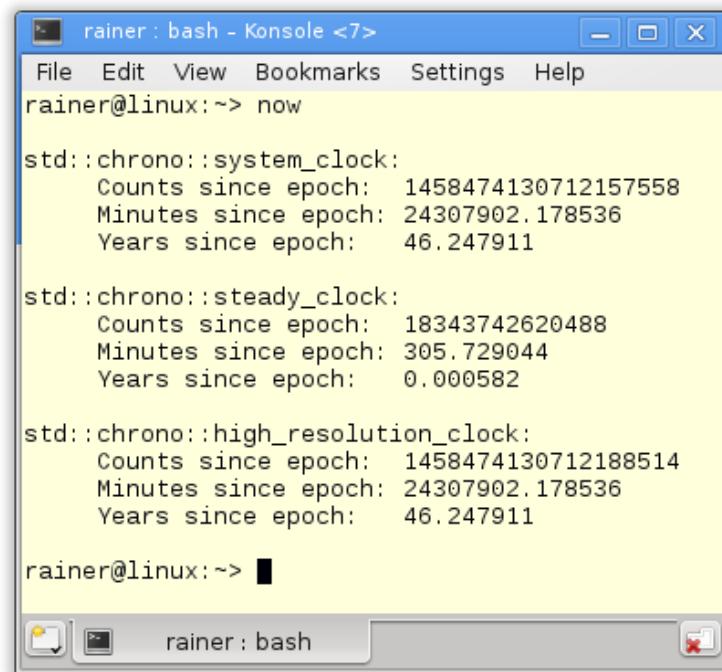
```
1 // now.cpp
2
3 #include <chrono>
4 #include <iomanip>
5 #include <iostream>
6
7 using namespace std::chrono;
8
9 template <typename T>
10 void durationSinceEpoch(const T dur){
11     std::cout << "    Counts since epoch: " << dur.count() << std::endl;
12     typedef duration<double, std::ratio<60>> MyMinuteTick;
13     const MyMinuteTick myMinute(dur);
14     std::cout << std::fixed;
15     std::cout << "    Minutes since epoch: " << myMinute.count() << std::endl;
16     typedef duration<double, std::ratio<60*60*24*365>> MyYearTick;
17     const MyYearTick myYear(dur);
18     std::cout << "    Years since epoch: " << myYear.count() << std::endl;
19 }
20
21 int main(){
22
23     std::cout << std::endl;
24
25     system_clock::time_point timeNowSysClock = system_clock::now();
26     system_clock::duration timeDurSysClock= timeNowSysClock.time_since_epoch();
27     std::cout << "system_clock: " << std::endl;
28     durationSinceEpoch(timeDurSysClock);
29
30     std::cout << std::endl;
31
32     const auto timeNowStClock = steady_clock::now();
33     const auto timeDurStClock= timeNowStClock.time_since_epoch();
34     std::cout << "steady_clock: " << std::endl;
35     durationSinceEpoch(timeDurStClock);
36 }
```

¹³²http://en.cppreference.com/w/cpp/chrono/time_point/time_since_epoch

```
37     std::cout << std::endl;
38
39     const auto timeNowHiRes = high_resolution_clock::now();
40     const auto timeDurHiResClock = timeNowHiRes.time_since_epoch();
41     std::cout << "high_resolution_clock: " << std::endl;
42     durationSinceEpoch(timeDurHiResClock);
43
44     std::cout << std::endl;
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 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.



```
rainer : bash - Konsole <7>
File Edit View Bookmarks Settings Help
rainer@linux:~> now

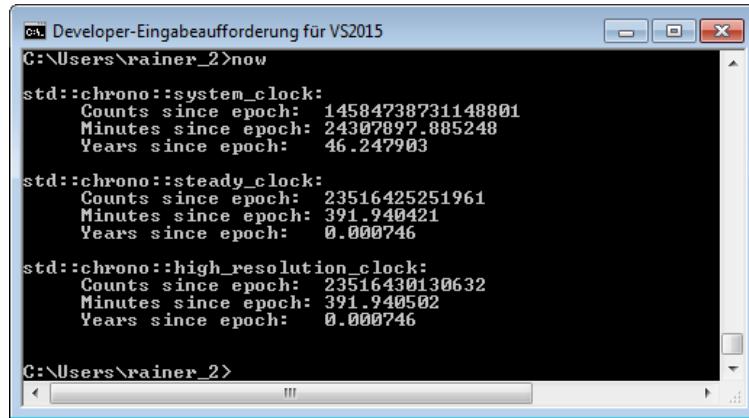
std::chrono::system_clock:
Counts since epoch: 1458474130712157558
Minutes since epoch: 24307902.178536
Years since epoch: 46.247911

std::chrono::steady_clock:
Counts since epoch: 18343742620488
Minutes since epoch: 305.729044
Years since epoch: 0.000582

std::chrono::high_resolution_clock:
Counts since epoch: 1458474130712188514
Minutes since epoch: 24307902.178536
Years since epoch: 46.247911

rainer@linux:~> █
```

The epoch for each clock on Linux



```
C:\ Developer-Eingabeaufforderung für VS2015
C:\Users\rainer_2>now
std::chrono::system_clock:
  Counts since epoch: 14584738731148801
  Minutes since epoch: 24307897.885248
  Years since epoch: 46.247903

std::chrono::steady_clock:
  Counts since epoch: 23516425251961
  Minutes since epoch: 391.940421
  Years since epoch: 0.000746

std::chrono::high_resolution_clock:
  Counts since epoch: 23516430130632
  Minutes since epoch: 391.940502
  Years since epoch: 0.000746

C:\Users\rainer_2>
```

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.

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.

Conventions

The methods for handling time in multithreading programs follow a simple convention. Methods ending with `_for` have to be parametrised by a [time duration](#); methods ending with `_until` by a [time point](#). Here is a concise overview of the methods dealing with sleeping, blocking, and waiting.

Methods for sleeping, blocking, and waiting

Multithreading Component	<code>_until</code>	<code>_for</code>
<code>std::thread th</code>	<code>th.sleep_until(in2min)</code>	<code>th.sleep_for(2s)</code>
<code>std::unique_lock lk</code>	<code>lk.try_lock_until(in2min)</code>	<code>lk.try_lock(2s)</code>
<code>std::condition_variable cv</code>	<code>cv.wait_until(in2min)</code>	<code>cv.wait_for(2s)</code>
<code>std::future fu</code>	<code>fu.wait_until(in2min)</code>	<code>fu.wait_for(2s)</code>
<code>std::shared_future shFu</code>	<code>shFu.wait(in2min)</code>	<code>shFu.wait_for(2s)</code>

`in2min` stands for a time 2 minutes in the future. `2s` is a time duration of 2 seconds. Although I use `auto` in the initialisation 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.

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 for the notification of 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

```
1 // sleepAndWait.cpp
2
3 #include <utility>
4 #include <iostream>
5 #include <future>
6 #include <thread>
7 #include <utility>
8
9 using namespace std;
10 using namespace std::chrono;
11
12 mutex coutMutex;
13
14 long double getDifference(const steady_clock::time_point& tp1,
15                           const steady_clock::time_point& tp2){
16     const auto diff= tp2 - tp1;
17     const auto res= duration<long double, milli> (diff).count();
18     return res;
19 }
20
21 void producer(promise<int>&& prom){
22     cout << "PRODUCING THE VALUE 2011\n\n";
23     this_thread::sleep_for(seconds(5));
24     prom.set_value(2011);
25 }
26
27 void consumer(shared_future<int> fut,
28                steady_clock::duration dur){
29     const auto start = steady_clock::now();
30     future_status status= fut.wait_until(steady_clock::now() + dur);
31     if ( status == future_status::ready ){
32         lock_guard<mutex> lockCout(coutMutex);
33         cout << this_thread::get_id() << " ready => Result: " << fut.get()
34             << endl;
35     }
36     else{
```

```
37     lock_guard<mutex> lockCout(coutMutex);
38     cout << this_thread::get_id() << " stopped waiting." << endl;
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" << endl;
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." << endl;
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() << endl;
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" << endl;
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." << endl;
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() << endl;
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" << endl;
92 }
93
94 int main(){
95
96     cout << endl;
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 << endl;
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 work package of the thread, 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 synchronisation of the program. Both the clock determining the current time and `std::cout` are shared variables, but no synchronisation is necessary. Firstly, the method 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.

CppMem - An Overview

CppMem¹³³ is an interactive tool for exploring the behaviour 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:

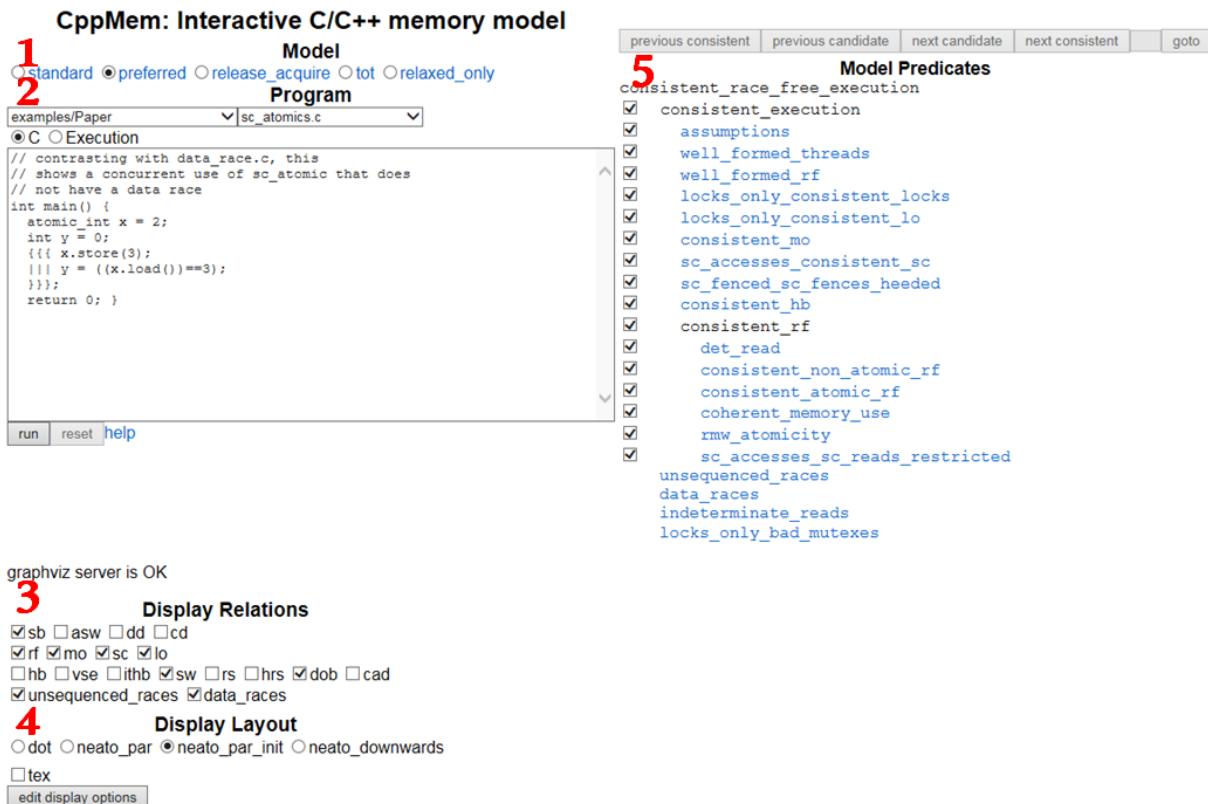
1. CppMem verifies the behaviour of small code snippets. Based on the chosen variant of the C++ memory model, the tool considers all possible interleavings of threads, visualises each of them in a graph and annotates these graphs with additional details.
2. The very accurate analysis of CppMem gives 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 the very detailed analysis related to this incredibly challenging topic and is highly configurable. Therefore, I plan to present the components of the tool.

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 to understand my process of ongoing optimisation.

¹³³<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.

1. Model

- Specifies the C++ memory model. *preferred* is a simplified but equivalent variant of the C++11 memory model.

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 the details of these programs read the very well written article [Mathematizing C++ Concurrency](#)¹³⁴. Of course, you can also run your code.
- CppClass 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.
 - 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`.

¹³⁴<http://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf>

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

4. Display Layout

- With this switch you can choose, which Doxygraph¹³⁵ graph is used.

5. Model Predicates

- With this button, you can set the predicates for the chosen model, that can cause a non-consistent (not data-race-free) execution; therefore, if you get a non-consistent execution, you see exactly the reason for the non-consistent execution. I do not use this buttons in this book.

See the [documentation¹³⁶](#) 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.

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.

¹³⁵<https://sourceforge.net/projects/doxygen/>

¹³⁶<http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/help.html>

Paper

The examples/Paper category gives you a few examples which are intensely discussed in the paper [Mathematizing C++ Concurrency](#)¹³⁷.

- `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.

The Test Run

Choose the program `data_race.c` from the CppMem samples. The run button shows immediately there is a [data race](#).

¹³⁷<https://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf>

CppMem: Interactive C/C++ memory model

Model
 standard preferred release_acquire tot relaxed_only

Program
 C Execution
 examples/Paper data_race.c

1 // a data race (dr)
 int main() {
 int x = 2;
 int y;
 {{{ x = 3;
 }}} y = (x==3);
 }};
 return 0; }

2 reset help 2 executions; 1 consistent, not race free

3 previous consistent previous candidate next candidate next consistent 2 goto

Execution candidate no. 2 of 2

Model Predicates

consistent_race_free_execution = false	
<input checked="" type="checkbox"/> consistent_execution = true	= true
<input checked="" type="checkbox"/> assumptions	= true
<input checked="" type="checkbox"/> well_formed_threads	= true
<input checked="" type="checkbox"/> well_formed_rf	= true
<input checked="" type="checkbox"/> locks_only_consistent_locks	= true
<input checked="" type="checkbox"/> locks_only_consistent_lo	= true
<input checked="" type="checkbox"/> consistent_mo	= true
<input checked="" type="checkbox"/> sc_accesses_consistent_sc	= true
<input checked="" type="checkbox"/> sc_fenced_sc_fences_headed	= true
<input checked="" type="checkbox"/> consistent_hb	= true
<input checked="" type="checkbox"/> consistent_rf	= true
<input checked="" type="checkbox"/> det_read	= true
<input checked="" type="checkbox"/> consistent_non_atomic_rf	= true
<input checked="" type="checkbox"/> consistent_atomic_rf	= true
<input checked="" type="checkbox"/> coherent_memory_use	= true
<input checked="" type="checkbox"/> rmw_atomicity	= true
<input checked="" type="checkbox"/> sc_accesses_sc_reads_restricted	= true
unsequenced_races are absent	
data_races are present	
indeterminate_reads are absent	
locks_only_bad_mutexes are absent	

4

a:Wna x=2
 b:Wna x=3
 c:Rna x=2
 d:Wna y=0

sw
 rf,sw
 dr
 sb

Computed executions

Display Relations

sb asw dd cd
 rf mo sc lo
 hb vse ithb sw rs hrs dob cad
 unsequenced_races data_races

Display Layout

dot neato_par neato_par_init neato_downwards
 tex

edit display options

Files: out_exc, out_dot, out_dsp, out_tex

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 synchronisation.
 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 the Display Relations.
 - a:`Wna` $x=2$ is in the graphic the a-th statement, which is a non-atomic write. `Wna` stands for “Write non-atomic”.
 - The key edge in the graph is the edge between the writing of x (b:`Wna`) and the reading of x (C:`Rna`). That’s the data race on x .

Further Categories

The further 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](#)¹³⁸. If possible, I present the program with [sequential consistency](#).

Store Buffering (`examples/SB_store_buffering`)

Two threads write to separate locations and then read from the other location.

`SB+sc_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;
}
```

Message Passing (`examples/MP_message_passing`)

One thread writes data (non-atomic) and the sets an atomic flag while the second thread waits for the flag and read 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 synchronisation
// 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).readsvvalue(1);
        r2=x; } } }
    return 0;
}
```

¹³⁸<https://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf>

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;
}
```

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 the 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;
}
```

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;
}
```

Glossary

The idea of this glossary is by no means to be exhaustive but to provide a reference for the essential terms.

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.

CAS

CAS stands for *compare-and-swap* and is an atomic operation. It compares an 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`.

Callable Unit

A callable unit (short callable) is something that behaves like a function. Not only are these named functions but also function objects and lambda functions. If a callable accepts one argument, it's called unary callable; if taking two arguments, binary callable.

[Predicates](#) are special callables that return a boolean as a result.

Concurrency

Concurrency means that the execution of several tasks overlaps. Concurrency is a superset of [parallelism](#).

Critical Section

A critical section is a section of code which at most one thread can use at a time.

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.

Function Objects

First of all, don't call them **functors**¹³⁹. That's a *well-defined* term from a branch of mathematics called **category theory**¹⁴⁰.

Function objects are objects that behave like functions. They achieve this by implementing the function call operator. As function objects are objects, they can have attributes and, therefore, state.

```
struct Square{
    void operator()(int& i){i= i*i;}
};

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

std::for_each(myVec.begin(), myVec.end(), Square());

for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
```



Instantiate function objects to use them

It's a common error that the name of the function object (**Square**) is used in an algorithm instead of 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())`

¹³⁹<https://en.wikipedia.org/wiki/Functor>

¹⁴⁰https://en.wikipedia.org/wiki/Category_theory

Lambda Functions

Lambda functions provide their functionality in-place. The compiler gets its information right on the spot and has therefore excellent optimisation 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.

```
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.

Lock-free

A non-blocking algorithm is lock-free if there is guaranteed system-wide progress.

Lost Wakeup

A lost wakeup is a situation in which a thread misses its wakeup notification due to a [race condition](#). That may happen if you use a [condition variable without a predicate](#).

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: $x * y = y * x$

Memory Location

A memory location is according to [cppreference.com¹⁴¹](http://en.cppreference.com/w/cpp/language/memory_model)

- 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.

Memory Model

The memory model defines the relation between objects and [memory location](#) and deals in particular with the question: What happens if two threads access the same memory locations.

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.

Monad

Haskell as a pure functional language has only pure functions. A key feature of these pure functions is that they always return the same result when given the same arguments. Thanks to this property called [referential transparency¹⁴²](#) 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 that are dependent 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.

¹⁴¹http://en.cppreference.com/w/cpp/language/memory_model

¹⁴²https://en.wikipedia.org/wiki/Referential_transparency

The concept of the monad is from [category theory](#)¹⁴³, 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:

- **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 identity function and the bind operator. 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](#)¹⁴⁴ which is a kind of a Maybe monad. With C++20/23 we will probably get [extended futures](#) and the [ranges library](#)¹⁴⁵ from Eric Niebler. Both are monads.

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](#)¹⁴⁶.

¹⁴³https://en.wikipedia.org/wiki/Category_theory

¹⁴⁴<http://en.cppreference.com/w/cpp/utility/optional>

¹⁴⁵<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4128.html>

¹⁴⁶<http://jcip.net/>

Parallelism

Parallelism means that several tasks are performed at the same time. Parallelism is a subset of [Concurrency](#).

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.

RAII

Resource Acquisition Is Initialization, in short RAII, stands for a popular technique in C++, in which the resource acquisition and release are bound to the lifetime of an object. This means for a lock that the mutex will be locked in the constructor and unlocked in the destructor.

Typical use cases in C++ are [locks](#) that handle the lifetime of its underlying [mutex](#chapterXXXMutexes, or a smart pointer that handles the lifetime of its resource (memory).

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.

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.

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.

Spurious Wakeup

A spurious wakeup is a phenomenon of condition variables. It may happen that the waiting component of the condition variable erroneously gets a notification, although the notification component didn't notify it.

Thread

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, 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 component of a process. 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](#)¹⁴⁷.

Total order

A total order is a binary relation (\leq) on some set X which is antisymmetric, transitive, and total.

- **Antisymmetric:** if $a \leq b$ and $b \leq a$ then $a = b$
- **Transitivity:** if $a \leq b$ and $b \leq c$ then $a \leq c$
- **Totality:** $a \leq b$ or $b \leq a$

volatile

`volatile` is typically used to denote objects which can change independently of the regular program flow. These are, for example, objects in embedded programming which 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 optimised storing in caches takes place.

wait-free

A non-blocking algorithm is wait-free if there is guaranteed per-thread progress.

¹⁴⁷[https://en.wikipedia.org/wiki/Thread_\(computing\)](https://en.wikipedia.org/wiki/Thread_(computing))

Index

Entries in capital letters stand for sections and subsections.

A

A Cooperatively Interruptible Joining Thread
A Generator Function
A Quick Overview
A thread-safe singly linked list
ABA
accumulate
Accuracy and Steadiness
ACI(D)
ACID (Glossary)
acquire fence
Acquire-Release Fences
Acquire-Release Semantic
adopt_lock
All Atomic Operations
All Multithreading Numbers with a Shared Variable
All Single Threaded Numbers
arrive_and_drop
arrive_and_wait
asctime
associative (Glossary)
async
Atomic Operations on shared_ptr
Atomic Smart Pointers
atomic<bool>
atomic<integral type>
atomic<shared_ptr<T>>
atomic<T*>
atomic<user-defined type>
atomic<weak_ptr<T>>
atomic_bool
atomic_cancel
atomic_char16_t
atomic_char32_t

atomic_char
atomic_clear
atomic_clear_explicit
atomic_commit
atomic_int16_t
atomic_int32_t
atomic_int64_t
atomic_int8_t
atomic_int
atomic_int_fast16_t
atomic_int_fast32_t
atomic_int_fast64_t
atomic_int_fast8_t
atomic_int_least16_t
atomic_int_least32_t
atomic_int_least64_t
atomic_int_least8_t
atomic_intmax_t
atomic_intptr_t
atomic_llong
atomic_long
atomic_noexcept
atomic_ptrdiff_t
atomic_schar
atomic_shared_ptr
atomic_short
atomic_signal_fence
atomic_size_t
atomic_test_and_set
atomic_test_and_set_explicit
atomic_thread_fence
atomic_uchar
atomic_uint16_t
atomic_uint32_t

atomic_uint64_t
atomic_uint8_t
atomic_uint
atomic_uint_fast16_t
atomic_uint_fast32_t
atomic_uint_fast64_t
atomic_uint_fast8_t
atomic_uint_least16_t
atomic_uint_least32_t
atomic_uint_least64_t
atomic_uint_least8_t
atomic_uintmax_t
atomic_uintptr_t
atomic_ullong
atomic_ulong
atomic_ushort
atomic_wchar_t
atomic_weak_ptr
atomicity
Atomics
await_ready
await_resume
await_suspend
B
barrier
Basics of the Memory Model
Best Practices
Blocking Issues
Breaking of Program Invariants
busy waiting
C
Calculating the Sum of a Vector
Calculations
call_once
callable (Glossary)
Callable Unit
carries-a-dependency
CAS
CAS
Case Studies
catch fire semantic
Challenges
clear
Clocks
co_await
co_return
co_yield
commutative (Glossary)
compare_exchange_strong
compare_exchange_weak
completion phase
Concurrency (Glossary)
Concurrent Calculation
Condition Variables
condition_variable
condition_variable_any
consistency
Constant Expressions
ContinuableFuture
Continuation with then
Conventions
Coroutines
count_down
count_down_and_wait
CppMem: Atomics with Acquire-Release Semantic
CppMem: Atomics with Non-atomics
CppMem: Atomics with Relaxed Semantic
CppMem: Atomics with Sequential Consistency
CppMem: Locks
CppMem: Non-Atomic Variables
CppMem
CppMem
Creating new Futures
CriticalSection (Glossary)
Cross the valid Time Range
current_exception
D
Data Races
Deadlocks
defer_lock
deferred (future_status)
define_task_block
define_task_block_restore_thread
dependency-order-before

Design Goals
detach
Details
Different Synchronisation and Ordering Constraints
Double-Checked Locking Pattern
durability
E
epoch (time_point)
Epoch
Exceptions
exchange
exclusive_scan
execution agent
execution context
execution function
Execution Policies
execution resource
execution::require
execution::par
execution::par_unseq
execution::parallel_policy
execution::parallel_policy
execution::parallel_unsequenced_policy
execution::parallel_unsequenced_policy
execution::seq
execution::sequenced_policy
execution::sequenced_policy
Executor (Glossary)
executor propagation
Executors
Extended Futures
F
False Sharing
Fences
fetch_add
fetch_and
fetch_or
fetch_sub
fetch_xor
Fire and Forget
flex_barrier
foldl1
foldl
for_each
for_each_n
Fork and Join
Free Atomic Function
From Time Point to Calendar Time
full fence
Function Objects (Glossary)
Further Information
future
future_errc::broken_promise
future_error
future_status
FutureContinuation
G
get
get_future (parameter_pack)
get_future (promise)
get_id
Glossary
gmtime
H
h (time literal)
hardware_constructive_interference_size
hardware_destructive_interference_size
hazard pointers
high_resolution_clock
hours
I
inclusive_scan
interrupt (interrupt_token)
Interrupt Tokens
interrupt_token
is_always_lock_free
is_execution_policy
is_interrupted (interrupt_token)
is_lock_free
is_ready
isolation
Issues of Mutexes
J
join and detach

join
joinable
Joining Threads
jthread
K
Kind of Atomic Operations
L
Lambda Functions (Glossary)
latch
Latches and Barriers
Lifetime Issues of Variables
load
LoadLoad
LoadStore
lock (unique_lock)
lock-free (Glossary)
lock
lock_guard
Locks
Lost Wakeup (Glossary)
Lost Wakeup
M
make_exception_ptr
make_exceptional_future
make_ready_future
make_ready_future_at_thread_exit
map
Math Laws (Glossary)
max (time_point)
Memory Barriers
Memory Location (Glossary)
Memory Model (Glossary)
Memory Model
memory_order_acq_rel

memory_order_acquire
memory_order_consume
memory_order_relaxed
memory_order_release
memory_order_seq_cst
Meyers Singleton
microseconds
milliseconds
min (time literal)
min (time_point)
minutes
Modification Order (Glossary)
modification order consistency
Monad (Glossary)
Moving Threads
ms (time literal)
Multithreaded Summation with a Shared Variable
Multithreading
mutex
Mutexes
My Performance Measurement
N
nanoseconds
native_handle
native_handle
Non-blocking (Glossary)
Notifications
notify_all
notify_one
now (time_point)
ns (time literal)
O
once_flag
Ongoing Optimisation
Operations
owns_lock
P
packaged_task
Parallel Algorithms of the STL
Parallelism (Glossary)
partial_sum
Performance Numbers of the various Thread-Safe Singleton
Implementations
Predicate(Glossary)
promise and future
promise
R
Race Condition
RAII(Glossary)
ratio
RCU

ready (future_status)
recursive_mutex
recursive_timed_mutex
reduce
Reference PCs
relaxed block
Relaxed Semantic
release fence
Release Sequence(Glossary)
release
reset
resumable function
S
s (time literal)
scanl1
scanl
scoped_lock
scoped_thread
seconds
SemiFuture
Sequence Point(Glossary)
Sequential Consistency(Glossary)
Sequential Consistency
set_exception
set_exception_at_thread_exit
set_value(parameter_pack)
set_value(promise)
set_value_at_thread_exit
share
Shared Data
shared_future
shared_lock
shared_mutex
shared_ptr
shared_timed_mutex
SharedFuture
signal
SIGTERM
Single Threaded Addition of a Vector
singleton
Sleep and Wait
Spinlock versus Mutex
spinlock
Spurious wakeup(Glossary)
Spurious Wakeup
Static Variables
std::call_once with std::once_flag
steady_clock
store
StoreLoad
StoreStore
Strong Memory Model
Strong versus Weak Memory Model
Summation of a Vector: The Conclusion
Synchronisation with Atomic Variables or Fences
Synchronized and Atomic Blocks
system_clock
T
tagged state reference
Task Blocks
task_cancelled_exception
Tasks
test_and_set
The Atomic Flag
The Challenges
The Class Template std::atomic
The Contract
The Details
The Foundation
The functional Heritage
The Future: C++20_23
The Interface
The Interplay of Time Point, Time Duration, and Clock
The New Algorithms
The Scheduler
The Six Variants
The Start Policy
The Synchronisation and Ordering Constraints
The Test Run
The three Fences
The Typical Minunderstanding
The Wait Workflow
this_thread::get_id
this_thread::sleep_for

this_thread::sleep_until
this_thread::yield
Thread Arguments
Thread Creation
Thread Lifetime
Thread Safe Initialization
Thread(Glossary)
Thread-Local Data
Thread-Local Summation
Thread-Safe Initialisation of a Singelton
Thread-Safe Meyers Singleton
thread::hardware_concurrency
Threads versus Tasks
Threads
Time Duration
Time Library
Time Point
time_since_epoch
time_t
timed_mutex
timeout (future_status)
to_time_t
total order(Glossary)
transaction-unsafe
transaction_safe versus transaction_unsafe Code
transaction_safe
transaction_unsafe
Transactional Memory
transform_exclusive_scan
transform_inclusive_scan
transform_reduce
Transitivity
try_lock
try_lock_for
try_lock_until
Typical Use-Cases
U
Underlying Concepts
Unified Futures
unique_lock
unlock
us (time literal)

V
valid(future)
valid(interrupt_token)
valid(parameter_pack)
Various waiting strategies
volatile(Glossary)

W
wait (condition_variable)
wait (future)
wait (latch)
wait-free(Glossary)
wait_for (condition_variable)
wait_for (future)
wait_until (condition_variable)
wait_until (future)
Weak Memory Model
when_all
when_any