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 2017-05-26

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

Contents

Introduction

Concurrency with Modern C++ is a journey through current and upcoming concurrency in C++.

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

This book explains you the details to concurrency in modern C++ and gives you, in addition, many running code examples. Therefore you can combine the theory with the practices and get the most of it.

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

Conventions

I promise only a few conventions.

Special Fonts

Italic
: I use *Bold Font* if something is very important.

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

Special Boxes

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

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 the code examples. The name of the source file is in the header of the source code example.

How to read the book?

I have a strong advice how you should read the book.

If you are not very familiar with concurrency in C++, start with the part A Quick Overview.

Now, you have the great picture in mind and can proceed with the The Details. Skip the memory model in your first and maybe the second iteration of the book, unless you are totally sure, what you are looking for. The chapter about the Future: C++20 is optional. I'm very curious about the future. Maybe you not.

All, that belongs to the last part Further Information[1] should give you additional guidance to better understand my book and get the most out of it.

Personal Notes

Acknowledgements

About Me

I worked as a software architect, team lead and instructor for about 20 years. In my spare time, I like to write articles on the topics C++, Python and Haskell, but I also like to speak at conferences. Since 2016 I'm independent giving seminars about modern C++ and Python.

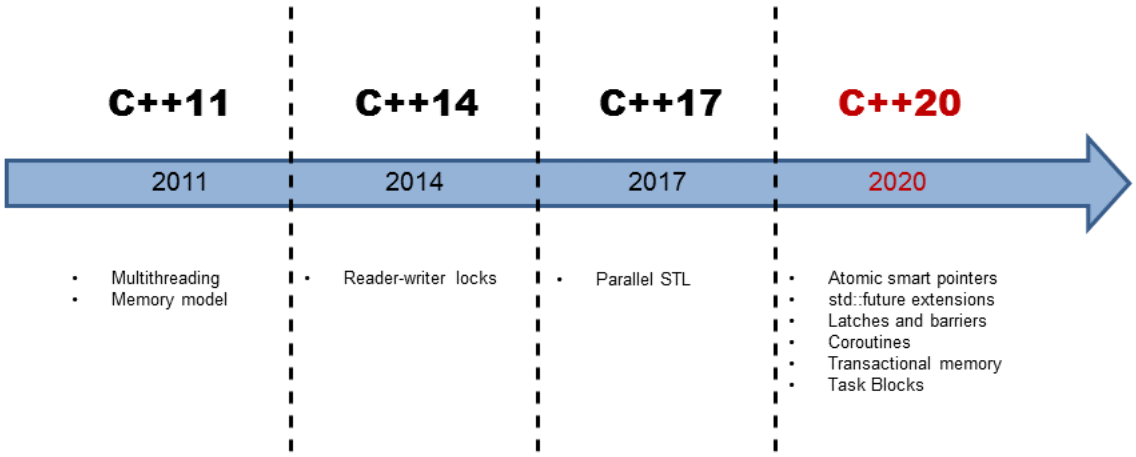
[1]

My Special Circumstances

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

A Quick Overview

Concurrency with Modern C++



C++ gets with the 2011 published C++ standard a multithreading library. This library has the basic building blocks like atomic variables, threads, locks and condition variables. That's the base on which upcoming C++ standards such as C++17 and C++20 can build higher abstractions. But C++11 already knows tasks, which provide a higher abstraction than the cited basic building blocks.

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

C++11 and C++14: The Base

For the first time with C++11, C++ supports native multithreading. This support consists of two parts: A *well-defined* memory model and a standardised threading interface. 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 points:

- Atomic operations: Operations that can be performed without interruption.
- Partial ordering of operations: Sequence of operations that must not be reordered.

- Visible effects of operations: Guarantees when operations on shared variables are visible in other threads.

The C++ memory model has a lot in common with its predecessor: the Java memory model. On the contrary, C++ permits the breaking of the sequential consistency. The sequential consistency is the default behaviour of atomic operations.

The sequential consistency provides two guarantees.

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

The memory model is based on atomic operations on atomic data types.

Atomic Data Types

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

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

Multithreading

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

Threads

A thread `std::thread` represents an executable unit. This executable unit, which the thread immediately starts, gets its work package as a callable unit. A callable unit can be a function, a function object or a lambda function.

The creator of a thread has to take care of the lifetime of its created thread. The executable unit of the created thread ends with the end of the callable. Either the creator is waiting until the created thread `t` is done (`t.join()`) or the creator detaches itself from the created thread: `t.detach()`. A thread `t` is *joinable* if no operation `t.join()` or `t.detach()` was performed on it. A *joinable* thread calls in its destructor the exception `std::terminate` 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 in particular that it can get an arbitrary number of arguments by copy or by reference. Either the callable or the thread can get the arguments.

Shared Data

If more than one thread is using a variable at the same time and the variable is mutable, you have to coordinate the access. Reading and writing at the same time a shared variable is a data race and therefore undefined behaviour. To coordinate the access to a shared variable is the job for mutexes and locks in C++.

Mutex

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

Locks

You should encapsulate a mutex in a lock to release the mutex automatically. A lock is an implementation of the RAII idiom because the lock binds the lifetime of the mutex to its lifetime. C++ has a `std::lock_guard` for the simple and a `std::unique_lock` and a `std::shared_lock` for the advanced use case, respectively.

Thread-safe Initialisation of Data

If you don't modify the data, it's sufficient to initialize them in a *thread-safe* way. C++ offers various ways to achieve this: using constant expression, using static variables 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 get its own copy. Therefore, there is no conflict. The lifetime of the thread local data is bound to the lifetime of its thread.

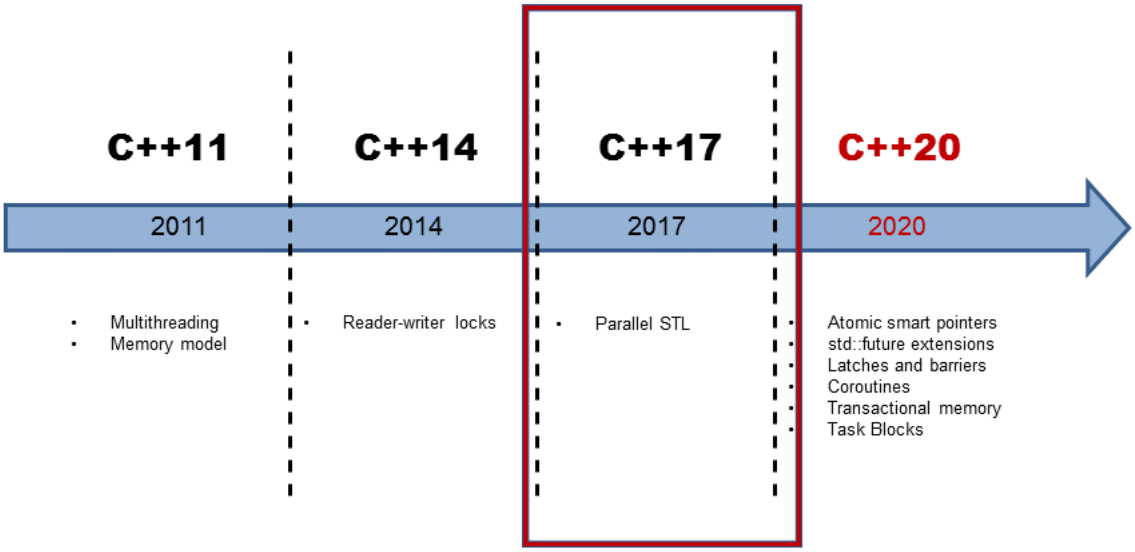
Condition Variables

Condition variables enables threads to be synchronized via messages. One thread acts as the sender, and the other as the receiver of the message. The receiver is waiting for the notification of the sender. Typical use cases for condition variables are producer-consumer workflows. A condition variable can be the sender but also the receiver of the message. Using condition variables right is quite challenging. Therefore, tasks are often the easier solution.

Tasks

Tasks have a lot in common with threads. But while you explicitly create a thread, a task is simply a job you start. The C++ runtime will automatically handle in the simple case of `std::async` the lifetime of the tasks. Tasks are like data channels. The promise puts data into the data channel, the future picks the value up. The data can be a value, an exception or simply a notification. In addition to `std::async`, C++ has the function `std::promise` and `std::future` that gives you more control about the task.

C++17: Parallel Algorithms of the Standard Template Library



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

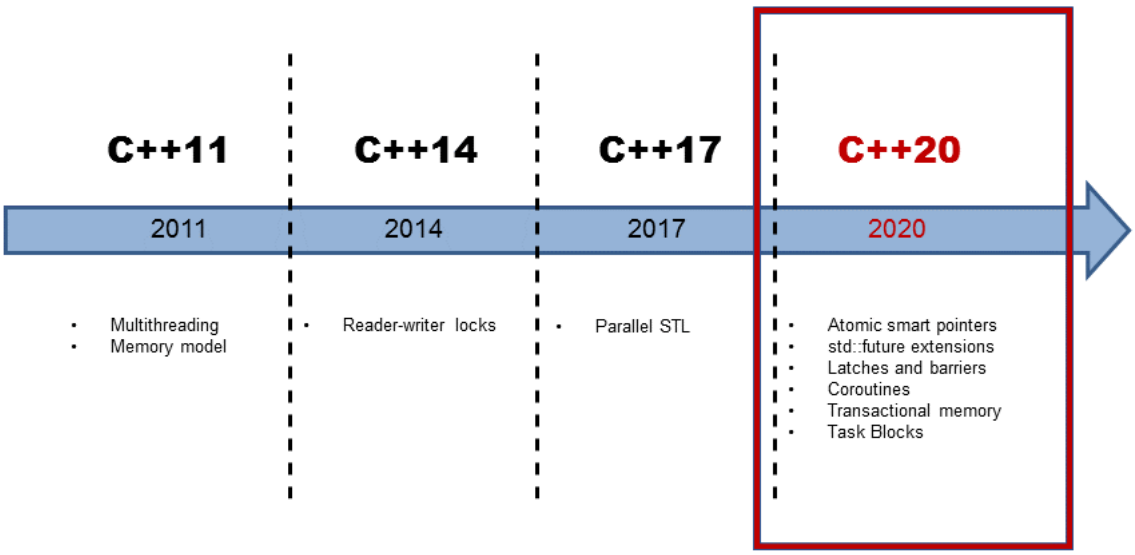
Execution Policy

With C++17, the most of the algorithms of the STL will be available in a parallel version. Therefore, you can invoke an algorithm with a so-called execution policy. This execution policy specifies if the algorithm runs sequential (`std::seq`), parallel (`std::par`), or parallel and vectorized (`std::par_unseq`).

New Algorithms

Beside of the 69 algorithms that are available in overloaded versions for parallel or parallel and vectorized execution, we get eight additional algorithms. The new ones are well suited for parallel reduce, scan, or transform operations.

C++20: The concurrent future



Itâ€™s Difficult to Make Predictions, Especially About the Future (Niels Bohr[2]). In contrary, I will make statements about the concurrent future of C++.

Atomic Smart Pointers

The smart pointer `std::shared_ptr`[3] and `std::weak_ptr`[4] have a conceptional issue in concurrent programs. They share per se mutable state. Therefore, they are prone to data races and therefore undefined behaviour. `std::shared_ptr` and `std::weak_ ptr` guarantee that the in- or decrementing of the reference counter is an atomic operation and the resource will be deleted exactly once, but both does not guarantee that the access to its resource is atomic. The new atomic smart pointers `std::atomic_shared_ptr` and `std::atomic_weak_ptr` solve this issue.

`std::future` Extensions

With tasks called promises and futures, we got a new multithreading concept in C++11. Although tasks have a lot to offer, they have a big drawback. Futures in C++11 can not be composed. That will not hold for the extended futures in C++20. Therefore, an extended future becomes ready, when its predecessor (`then`) becomes ready, `when_any` one of its predecessors become ready, or `when_all` of its predecessors become ready.

[2] https://en.wikipedia.org/wiki/Niels_Bohr
[3] http://en.cppreference.com/w/cpp/memory/shared_ptr
[4] http://en.cppreference.com/w/cpp/memory/weak_ptr

Latches and Barriers

C++14 has no semaphores. Semaphores enable it that threads can control access to a common resource. No problem, with C++20 we get latches and barriers. You can use latches and barriers for waiting at a synchronisation point until the counter becomes zero. The difference is, `std::latch` can only be used once; `std::barrier` and `std::flex_barrier` more the once. In contrary to a `std::barrier`, a `std::flex_barrier` can adjust its counter after each iteration.

Coroutines

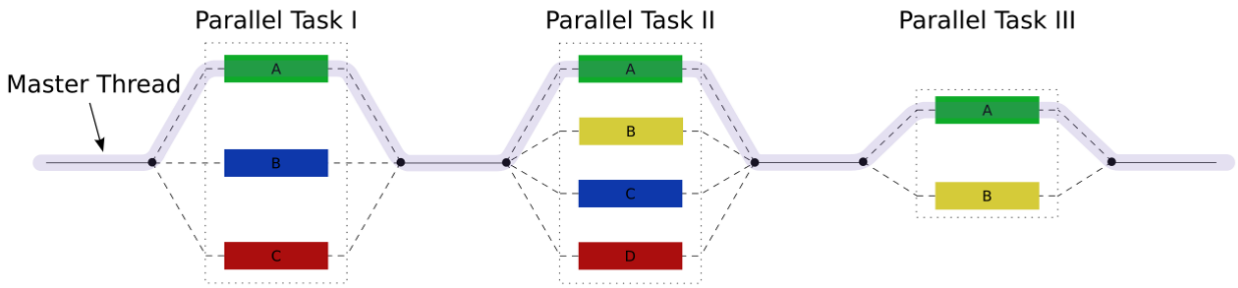
Coroutines are functions that can suspend and resume their execution while keeping their state. Coroutines are often the matter of choice to implement cooperative multitasking in operating systems, event loop, infinite lists, or pipelines.

Transactional Memory

The transactional memory idea is based on transactions from the database theory. A transaction is an action which provides the properties Atomicity, Consistency, Isolation and Durability (ACID). Accept for durability, all properties will hold for transactional memory in C++. C++ will have transactional memory in two flavours. One is called synchronised blocks and the other atomic blocks. Both have in common, that they will be executed in total order and behave as they were protected by a global lock. In contrary to synchronised blocks, atomic blocks can not execute transaction-unsafe code.

Task Blocks

Task Blocks implement the fork-join paradigm. The graphic shows the key idea.



Challenges

Programming concurrently is inherently complicated. That holds, in particular, true if you use C++11 and C++14 features. Therefore I will describe in detail the most challenging issues. My hope is that

if I dedicate a whole chapter to the challenges of concurrent programming, you may become more aware of the pitfalls. I will write about challenges such as critical sections, race contions, data races , and deadlocks.

Best Practices

Concurrent programming is inherently complicated. Therefore, best practices for data sharing, the right abstraction, or static code analysis make a lot of sense.

Time Library

The time library is a key component of the concurrent facilities of C++. Often you put a thread for a specific time duration or until a specific time point to sleep. The time library consists of the parts time point, time duration, and clock.

The Details

Memory Model

The Contract

Atomics

The Atomic Flag

The Class Template `std::atomic`

User Defined Atomics

Fences

The Synchronisation and Ordering Constraints

Sequential Consistency

Acquire-Release Semantic

Relaxed Semantic

The Glory Details

Multithreading

Threads

Creation

Lifetime

Arguments

Operations

Shared Data

Mutexes

Locks

Thread-safe Initialization

Constant expressions

Static variables with block scope

`std::call_once` and `std::once_flag`

Thread Local Data

Condition Variables

Tasks

Threads versus Tasks

std::async

std::packaged_task

std::promise and std::future

Tasks versus Condition Variables

Parallel Algorithms of the Standard Template Library

Execution policies

New algorithms

The Future: C++20

Atomic Smart Pointers

`std::atomic_shared_ptr` and `std::atomic_weak_ptr`

`std::future` Extensions

Latches and Barriers

Coroutines

Transaction Memory

Task Blocks

Further Improvements

Further Information

Challenges

Programming concurrently is inharently complicated. That holds in particular true if you use C++11 and C++14 features. I will not mention the memory model.My hope is that if I dedicate a whole chapter to the challanges of concurrent programm, you may become more aware of the pitfalls.

Critical section

Deadlocks

Data Races

Lifetime of data

Program Invariants

Race conditions

Best Practices

Data Sharing

The Right Abstractions

Static Code Analysis

The Time Library

Time Point

Time Duration

Clocks

Glossary

The idea of this glossary is by no means to be exhaustive. The ides is to provide a reference to the most important terms.

Callable unit

: ...

Concurrency

: ...

Modification order

: ...

Parallelism

: ...

RAII

: ...

Thread

: ...

Index