Username: Pralay Patoria **Book:** The C++ Standard Library: A Tutorial and Reference, Second Edition. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

18.4. Synchronizing Threads, or the Problem of Concurrency

Using multiple threads is almost always combined with concurrent data access. Rarely are multiple threads run independently of one another. Threads might provide data processed by other threads or prepare preconditions necessary to start other processes.

This is where multithreading becomes tricky. Many things can go wrong. Or, put another way, many things can behave differently from what the naive (and even the experienced) programmer might expect.

So, before discussing different ways to synchronize threads and concurrent data access, we have to understand the problem. Then we can discuss the following techniques to synchronize threads:

- Mutexes and locks (see Section 18.5, page 989), including `call_once()` (see Section 18.5.3, page 1000)
- Condition variables (see Section 18.6, page 1003)
- Atomics (see Section 18.7, page 1012)

18.4.1. Beware of Concurrency!

Before discussing the details of the problems of concurrency, let me formulate a first rule just in case you want to start programming without going into the depth of this subsection. If you learn one rule about dealing with multiple threads, it should be the following:

> **The only safe way to concurrently access the same data by multiple threads without synchronization is when *ALL* threads only *READ* the data.**

By "the same data" I mean data that uses the same memory location. If different threads concurrently access *different* variables or objects or different members of them, there is no problem because, since C++11, each variable except a bitfield is guaranteed to have its own memory location.[13] The only exceptions are bitfields, because different bitfields might share a memory location so that accessing different bitfields means shared access of the same data.

[13] The guarantee of separate memory locations for different objects was not given before C++11. C++98/C++03 was a standard for single-threaded applications only. So, strictly speaking, before C++11 concurrent access to different objects resulted in undefined behavior, although in practice it usually caused no problems.

However, when two or more threads concurrently access the *same* variable or object or member of it and at least one of the threads performs modifications, you can easily get into deep trouble if you don't synchronize that access. This is what is called a *data race* in C++. In the C++11 standard, a *data race* is defined as "two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other." A data race always results in undefined behavior.

As always with race conditions, the problem is that your code *might often* do what you intended, but it will not *always* work, which is one of the nastiest problems we can face in programming. Just by using other data, going into production mode, or switching a platform might suddenly break your code. So it's probably a good idea to care about concurrent data access if you use multiple threads.

18.4.2. The Reason for the Problem of Concurrent Data Access

To understand the problems of concurrent data access, we have to understand which guarantees C++ gives regarding the usage of concurrency. Note that a programming language such as C++ always is an abstraction to support different platforms and hardware, which provide different abilities and interfaces according to their architecture and purpose. Thus, a standard such as C++ specifies the *effect* of statements and operations and not the corresponding generated assembler code. The standard describes the *what*, not the *how*.

In general, the behavior is not defined so precisely that there is only one way to implement it. In fact, behavior might even explicitly be undefined. For example, the order of argument evaluation of a function call is unspecified. A program expecting a specific evaluation order would result in undefined behavior.

Thus, the important question is: Which guarantees does a language give? Programmers should not expect more, even though the additional guarantees might be "obvious." In fact, according to the so-called *as-if rule*, each compiler can optimize code as long as the behavior of the program visible from the outside behaves the same. Thus, the generated code is a *black box* and can vary as long as the *observable behavior* remains stable. To quote the C++ standard:

> An implementation is free to disregard any requirement of this International Standard as long as the result is *as if* the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.

Any undefined behavior is provided to give both compiler and hardware vendors the freedom and ability to generate the best code possible, whatever their criteria for "best" are. Yes, it applies to both: Compilers might unroll loops, reorder statements, eliminate dead code, prefetch data, and in modern architectures, for example, a hardware buffer might reorder loads or stores.

Reorderings can be useful to improve the speed of the program, but they might break the behavior. To be able to benefit from fast speed where useful, safety is not the default. Thus, especially for concurrent data access, we have to understand which guarantees are given.

18.4.3. What Exactly Can Go Wrong (the Extent of the Problem)

To give compilers and hardware enough freedom to optimize code, C++ does *not* in general give a couple of guarantees you might expect. The reason is that applying these guarantees in all cases, not only where useful, would cost too much in performance. In fact, in C++, we might have the following problems:

- **Unsynchronized data access**: When two threads running in parallel read and write the same data, it is open which statement comes first.
- **Half-written data**: When one thread reads data, which another thread modifies, the reading thread might even read the data in the *middle* of the write of the other thread, thus reading neither the old nor the new value.
- **Reordered statements:** Statements and operations might be reordered so that the behavior of each single thread is correct, but in combination of *all* threads, expected behavior is broken.

Unsynchronized Data Access

The following code ensures that `f()` is called for the absolute value of `val`, negating the argument `val` if it is negative:

```
if (val >= 0) {
    f(val);     // pass positive val
}
else {
    f(-val);    // pass negated negative val
}
```

In a single-threaded environment, this code works fine. However, in a multithreaded context, this code does not necessarily work. If multiple threads have access to `val`, the value of `val` might change between the `if` clause and the call of `f()` so that a negative value is passed to `f()`.

For the same reason, simple code such as:

```
std::vector<int> v;
...
if (!v.empty()) {
    std::cout << v.front() << std::endl;
}
```

can be a problem if `v` is shared between multiple threads, because between the call of `empty()` and the call of `front()`, `v` might become empty resulting in undefined behavior (see Section 7.3.2, page 275).

Note that this problem also applies to code implementing a function provided by the C++ standard library. For example, the guarantee that

```
v.at(5) // yield value of element with index 5
```

throws an exception if `v` does not have enough elements no longer applies if another thread might modify `v` while `at()` is called. Thus, keep in mind the following:

> **Unless otherwise stated, C++ standard library functions usually don't support writes or reads concurrently performed with writes to the same data structure.**[14]

[14] As Hans Boehm points out, an approach to support concurrent access to library objects would not be useful in general because if I need synchronization around data structure accesses, it's usually not just the individual accesses I need to protect but larger sections of code. This means that programmers need to do their own locking anyway, and the library-provided locking would be at best redundant.

That is, unless otherwise stated, multiple calls on the same object from multiple threads will result in undefined behavior.

However, the C++ standard library provides some guarantees regarding thread safety (see Section 4.5, page 56). For example:

- Concurrent access to *different elements* of the same container is possible (except for class `vector<bool>`). Thus, different threads might concurrently read and/or write different elements of the same container. For example, each thread might process something and store the result in "its" element of a shared vector.
- Concurrent access to a string stream, file stream, or stream buffer results in undefined behavior. However, as we have seen in this chapter before, formatted input and output to a standard stream that is synchronized with C I/O (see Section 15.14.1, page 845) is possible, although it might result in interleaved characters.

Half-Written Data

Consider that we have a variable[15]

[15] This example is taken with permission from [*N2480:MemMod*].

```
long long x = 0;
```

and one thread writing the data:

```
x = -1;
```

and one thread reading the data:

```
std::cout << x;
```

What is the output of the program; that is, which value does the second thread read when it outputs `x`? Well, the following answers are

possible:

- `0` (the old value of `x`), if the first thread has not assigned `-1` yet
- `-1` (the new value of `x`), if the first thread assigned `-1` already
- *Any other value*, if the second thread reads `x` during the assignment of `-1` by the first thread

The last option — *any other value* — can, for example, easily happen if, on a 32-bit machine, the assignment results in two stores and the read by the second thread happens when the first store was done but the second store was not yet done.

And beware, this does not apply to `long long` only. Even for a fundamental data type, such as `int` or `bool`, the standard does *not* guarantee that a read or a write is *atomic*; that is, that a read or write is an exclusive noninterruptable data access. A data race might be less likely, but to eliminate the possibility, you have to take the steps.

The same applies to more complicated data structures, even if they are provided by the C++ standard library. For example, for a `std::list<>` (see Section 7.5, page 290), it's up to the programmer to ensure that it doesn't get modified by another thread while a thread inserts or deletes an element. Otherwise, the other thread might use an inconsistent state of the list, where, for example, the forward pointer is modified already but the backward pointer is not.

Reordered Statements

Let's discuss another simple example.[16] Suppose we have two shared objects, an `int` to pass data from one thread to another and a Boolean `readyFlag`, which signals when the first thread has provided the data:

[16] This example is taken from multiple articles in *Bartosz Milewski's Programming Cafe* (see [*Milewski:Multicore*] and [*Milewski:Atomics*] for details).

```
long data;
bool readyFlag = false;
```

A naive approach is to synchronize the setting of the `data` in one thread with the consumption of the `data` in another thread. Thus, the providing thread calls:

```
data = 42;
readyFlag = true;
```

and the consuming thread calls:

```
while (!readyFlag) {    // loop until data is ready
    ;
}
foo(data);
```

Without knowing any details, almost every programmer at first would suppose that the second thread calls `foo()` when `data` has the value `42`, assuming that the call of `foo()` can be reached only if the `readyFlag` is `true`, which itself can be the case only after the first thread assigned `42` to `data`, because this happens before the `readyFlag` becomes `true`.

But this is not necessarily the case. In fact, the output of the second thread might be the value `data` had *before* the first thread assigned `42` (or even any other value, because the assignment of `42` might be half-done).

That is, the compiler and/or the hardware might reorder the statements so that effectively the following gets called:

```
readyFlag = true;
data = 42;
```

In general, such a reordering is allowed due to the rules of C++, which requires only that the *observable behavior inside a thread* of the generated code be correct. For the behavior of the first thread, it doesn't matter whether we first modify `readyFlag` or `data`; from the viewpoint of this thread, they are independent of each other. Thus, reorderings of statements are allowed as long as the visible effect to the outside of a single thread is the same.

For the same reason, even the second thread might reorder the statements, provided that the behavior of this thread is not affected:

```
foo(data);
while (!readyFlag) {    // loop until data is ready
    ;
}
```

Note that the observable behavior might be affected by such a reordering if `foo()` throws. Thus, it depends on details whether such reorderings are allowed, but in principle, the problem applies.

Again, the reason to allow such modifications is that by default, C++ compilers shall be able to generate code that is highly optimized, and some optimizations might be to reorder statements. By default, these optimizations are not required to care about possible other threads, which makes these optimizations easier because local analyses are enough.

18.4.4. The Features to Solve the Problems

To solve the three major problems of concurrent data access, we need the following concepts:

- **Atomicity:** This means that read or write access to a variable or to a sequence of statements happens exclusively and without any interruption, so that one thread can't read intermediate states caused by another thread.
- **Order:** We need some ways to guarantee the order of specific statements or of a group of specific statements.

The C++ standard library provides very different ways to deal with these concepts, so that programs benefit from additional guarantees regarding concurrent access:

- You can use *futures* (see Section 18.1, page 946) and *promises* (see Section 18.2.2, page 969), which guarantee both atomicity and order: Setting the *outcome* (return value or exception) of a *shared state* is guaranteed to happen before the processing of this outcome, which implies that read and write access does not happen concurrently.
- You can use *mutexes and locks* (see Section 18.5, page 989) to deal with *critical sections*, or *protected zones*, whereby you can grant exclusive access so that, for example, nothing can happen between a check and an operation based on that check. Locks provide atomicity by blocking all access using a second lock until a first lock on the same resource gets released. More precisely: The release of a lock object acquired by one thread is guaranteed to happen before the acquisition of the same lock object by another thread is successful. However, if two threads use locked access to data, the order in which they access it may change from run to run.
- You can use *condition variables* (see Section 18.6, page 1003) to efficiently allow one thread to wait for some predicate controlled by another thread to become true. This helps to deal with the order of multiple threads by allowing one or more threads to process data or a status provided by one or more other threads.[17]

[17] Concurrency experts won't consider condition variables to be a tool to deal with the problem of concurrent data access, because they're more a tool to improve performance than to provide correctness.

- You can use *atomic data types* (see Section 18.7, page 1012) to ensure that each access to a variable or object is atomic while the order of operations on the atomic types remains stable.
- You can use the *low-level interface of atomic data types* (see Section 18.7.4, page 1019), which allow experts to relax the order of atomic statements or to use manual barriers for memory access (so-called *fences*).

In principle, this list is sorted from high-level to low-level features. High-level features, such as futures and promises or mutexes and locks, are easy to use and provide little risk. Low-level features, such as atomics and especially their low-level interface, might provide better performance because they have lower latency and therefore higher scalability, but the risk of misuse grows significantly. Nevertheless, low-level features sometimes provide simple solutions for specific high-level problems.

With atomics, we go in the direction of *lock-free programming*, which even experts sometimes do wrong. To quote Herb Sutter from [*Sutter:LockFree*]: "[Lock-free code is] hard even for experts. It's easy to write lock-free code that appears to work, but it's very difficult to write lock-free code that is correct and performs well. Even good magazines and refereed journals have published a substantial amount of lock-free code that was actually broken in subtle ways and needed correction."

`volatile` and Concurrency

Note that I didn't mention `volatile` here as a feature for concurrent data access, although you might have expected that for the following reasons:

- `volatile` is known as a C++ keyword to prevent too much optimization.
- In Java, `volatile` provides some guarantees about atomicity and order.

In C++, `volatile` "only" specifies that access to external resources, such as shared memory, should not be optimized away. For example, without `volatile`, a compiler might eliminate redundant loads of the same shared memory segment because it can't see any modification of the segment throughout the whole program. But in C++, `volatile` provides neither atomicity nor a specific order.[18] Thus, the semantics of `volatile` between C++ and Java now differs.

[18] Thanks to Scott Meyers for pointing this out to me.

See also Section 18.5.1, page 998, for a discussion, why `volatile` usually is not required when mutexes are used to read data in a loop.