
18.7. Atomics

In the first example for condition variables (see Section 18.6.1, page 1003), we used a Boolean value `readyFlag` to let one thread signal that something is prepared or provided for another thread. Now, you might wonder why we still need a mutex here. If we have a Boolean value, why can't we concurrently let one thread change the value while another thread checks it? The moment the providing thread sets the Boolean to `true`, the observing thread should be able to see that and perform the consequential processing.

As introduced in Section 18.4, page 982, we have two problems here:

1. In general, reading and writing even for fundamental data types is not atomic. Thus, you might read a half-written Boolean, which according to the standard results in undefined behavior.
2. The generated code might change the order of operations, so the providing thread might set the ready flag before the data is provided, and the consuming thread might process the data before evaluating the ready flag.

With a mutex, both problems are solved, but a mutex might be a relatively expensive operation in both necessary resources and latency of the exclusive access. So, instead of using mutexes and lock, it might be worth using atomics instead.

In this section, I first introduce the *high-level interface* of atomics, which provides atomic operations using the default guarantee regarding the order of memory access. This default guarantee provides *sequential consistency*, which means that in a thread, atomic operations are guaranteed to happen in the order as programmed. Thus, problems of reordered statements as introduced in Section 18.4.3, page 986, do not apply. At the end of this section, I present the *low-level interface* of atomics: operations with relaxed order guarantees.

Note that the C++ standard library does not distinguish between a high-level and a low-level atomics interface. The term *low-level* was introduced by Hans Boehm, one of the authors of the library. Sometimes, it is also called the *weak*, or *relaxed*, atomic interface, and the high-level interface is sometimes also known as the *normal*, or *strong*, atomic interface.

18.7.1. Example of Using Atomics

Let's transfer the example from Section 18.6.1, page 1003, into a program using atomics:

Click here to view code image

```
#include <atomic>      // for atomic types
...
std::atomic<bool> readyFlag(false);

void thread1()
{
    // do something thread2 needs as preparation
    ...
    readyFlag.store(true);
}

void thread2()
{
    // wait until readyFlag is true (thread1 is done)
    while (!readyFlag.load()) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    // do whatever shall happen after thread1 has prepared things
    ...
}
```

First, we include the header file `<atomic>`, where atomics are declared:

```
#include <atomic>
```

Then, we declare an atomic object, using the `std::atomic<>` class template:

```
std::atomic<bool> readyFlag(false);
```

In principle, you can use any trivial, integral, or pointer type as template parameter.

Note that you *always* should initialize atomic objects because the default constructor does not fully initialize it (it's not that the initial value is undefined, it is that the lock is uninitialized).[26] For static-duration atomic objects, you should use a constant to initialize them. If only the default constructor is used, the only operation allowed next is to call a global `atomic_init()` operation as follows:

[26] Thanks to Lawrence Crowl for pointing this out.

```
std::atomic<bool> readyFlag;
```

```
…
std::atomic_init(&readyFlag,false);
```

This way of initialization is provided to be able to write code that also compiles in C (see Section 18.7.3, page 1019).

The two most important statements to deal with atomics are `store()` and `load()` :

- `store()` assigns a new value.
- `load()` yields the current value.

The important point is that these operations are guaranteed to be atomic, so we don't need a mutex to set the ready flag, as we had to without atomics. Thus, in the first thread, instead of

```
{
     std::lock_guard<std::mutex> lg(readyMutex);
     readyFlag = true;
}   // release lock
```

we simply can program:

```
readyFlag.store(true);
```

In the second thread, instead of

Click here to view code image

```
{
    std::unique_lock<std::mutex> l(readyFlagMutex);
    while (!readyFlag) {
        l.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        l.lock();
    }
} // release lock
```

we have to implement only the following:

Click here to view code image

```
while (!readyFlag.load()) {
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
}
```

However, when using condition variables, we still need the mutex for consuming the condition variable:

Click here to view code image

```
// wait until thread1 is ready (readyFlag is true)
{
    std::unique_lock<std::mutex> l(readyMutex);
    readyCondVar.wait(l, []{ return readyFlag.load(); });
} // release lock
```

For atomic types, you can still use the "useful," "ordinary" operations, such as assignments, automatic conversions to integral types, increments, decrements, and so on:

```
std::atomic<bool> ab(false);
ab = true;
if (ab) {
    …
}

std::atomic<int> ai(0);
int x = ai;
ai = 10;
ai++;
ai-=17;
```

Note, however, that to provide atomicity, some usual behavior might be slightly different. For example, the assignment operator yields the assigned value instead of a reference to the atomic the value was assigned to. See Section 18.7.2, page 1016, for details.

Let's look at a complete example using atomics:

Click here to view code image

```
// concurrency/atomics1.cpp

#include <atomic>    // for atomics
#include <future>    // for async() and futures
```

```
#include <thread>     // for this_thread
#include <chrono>     // for durations
#include <iostream>

long data;
std::atomic<bool> readyFlag(false);

void provider ()
{
    // after reading a character
    std::cout << "<return>" << std::endl;
    std::cin.get();

    // provide some data
    data = 42;

    // and signal readiness
    readyFlag.store(true);
}

void consumer ()
{
    // wait for readiness and do something else
    while (!readyFlag.load()) {
        std::cout.put('.').flush();
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
    }

    // and process provided data
    std::cout << "\nvalue : " << data << std::endl;
}

int main()
{
    // start provider and consumer
    auto p = std::async(std::launch::async,provider);
    auto c = std::async(std::launch::async,consumer);
}
```

Here, thread `provider()` first provides some `data` and then uses a `store()` to signal that the data is provided:

```
data = 42;                  // provide some data
readyFlag.store(true);      // and signal readiness
```

The `store()` operation performs a so-called *release* operation on the affected memory location, which by default ensures that all prior memory operations, whether atomic or not, become visible to other threads before the effect of the store operation.

Accordingly, thread `consumer()` performs a loop of `load()` s and processes `data` then:

```
while (!readyFlag.load()) {         // loop until ready
    ...
}
std::cout << data << std::endl;     // and process provided data
```

The `load()` operation performs a so-called *acquire* operation on the affected memory location, which by default ensures that all following memory operations, whether atomic or not, become visible to other threads after the load operation.

As a consequence, because the setting of `data` *happens before* the `provider()` stores `true` in the `readyFlag` and the processing of `data` happens after the `consumer()` has loaded `true` as value of the `readyFlag` , the processing of `data` is guaranteed to happen after the data was provided.

This guarantee is provided because in all atomic operations, we use a default *memory order* named `memory_order_seq_cst` , which stands for *sequential consistent memory order*. With low-level atomics operations, we are able to relax this order guarantee (see Section 18.7.4, page 1019, for details).

18.7.2. Atomics and Their High-Level Interface in Detail

In `<atomic>` , the class template `std::atomic<>` provides the general abilities of atomic data types. It can be used for any trivial type. Specializations are provided for `bool` , all integral types, and pointers:

Click here to view code image

```
template<typename T> struct atomic;      // primary class template
template<> struct atomic<bool>;          // explicit specializations
template<> struct atomic<int>;
```

```
...
template<typename T> struct atomic<T*>;    // partial specialization for pointers
```

Table 18.11 lists the high-level operations provided for atomics. If possible, they map directly to corresponding CPU instructions. Column ***triv*** flags operations provided for `std::atomic<bool>` and atomics of other trivial types; column ***int type*** flags operations provided for `std::atomic<>`, if an integral type is used; and column ***ptr type*** flags operations provided for `std::atomic<>`, if a pointer type is used.

Table 18.11. High-Level Operations of Atomics

Operation	*triv*	*int type*	*ptr type*	Effect
atomic `a=`*val*	Yes	Yes	Yes	Initializes a with *val* (not an atomic operation)
atomic `a; atomic_init(&a,`*val*`)`	Yes	Yes	Yes	Ditto (without `atomic_init()`, a is not initialized)
`a.is_lock_free()`	Yes	Yes	Yes	`true` if type internally does not use locks
`a.store(`*val*`)`	Yes	Yes	Yes	Assigns *val* (returns `void`)
`a.load()`	Yes	Yes	Yes	Returns copy of the value of a
`a.exchange(`*val*`)`	Yes	Yes	Yes	Assigns *val* and returns copy of old value of a
`a.compare_exchange_strong(`*exp*`,` *des*`)`	Yes	Yes	Yes	CAS operation (see below)
`a.compare_exchange_weak(`*exp*`,` *des*`)`	Yes	Yes	Yes	Weak CAS operation
`a =` *val*	Yes	Yes	Yes	Assigns and returns copy of *val*
`a.operator` ***atomic***`()`	Yes	Yes	Yes	Returns copy of the value of a
`a.fetch_add(`*val*`)`		Yes	Yes	Atomic `t+=`*val* (returns copy of new value)
`a.fetch_sub(`*val*`)`		Yes	Yes	Atomic `t-=`*val* (returns copy of new value)
`a +=` *val*		Yes	Yes	Same as `t.fetch_add(`*val*`)`
`a -=` *val*		Yes	Yes	Same as `t.fetch_sub(`*val*`)`
`++a, a++`		Yes	Yes	Calls `t.fetch_add(1)` and returns copy of a or `a+1`
`--a, a--`		Yes	Yes	Calls `t.fetch_sub(1)` and returns copy of a or `a-1`
`a.fetch_and(`*val*`)`		Yes		Atomic `a&=`*val* (returns copy of new value)
`a.fetch_or(`*val*`)`		Yes		Atomic `a|=`*val* (returns copy of new value)
`a.fetch_xor(`*val*`)`		Yes		Atomic `a^=`*val* (returns copy of new value)
`a &=` *val*		Yes		Same as `a.fetch_and(`*val*`)`
`a |=` *val*		Yes		Same as `a.fetch_or(`*val*`)`
`a ^=` *val*		Yes		Same as `a.fetch_xor(`*val*`)`

Note a couple of remarks regarding this table:

- In general, operations yield copies rather than references.
- The default constructor does not initialize a variable/object completely. The only legal operation after default construction is calling `atomic_init()` to initialize the object (see Section 18.7.1, page 1013).
- The constructor for a value of the corresponding type is not atomic.
- All functions except constructors are overloaded for `volatile` and *non-* `volatile`.

For example, for `atomic<int>`, the following assignment operations are declared:

Click here to view code image

```
namespace std {
    // specialization of std::atomic<> for int:
    template<> struct atomic<int> {
      public:
        // ordinary assignment operators are not provided:
        atomic& operator=(const atomic&) = delete;
        atomic& operator=(const atomic&) volatile = delete;
        // but assignment of an int is provided, which yields the passed argument:
        int operator= (int) volatile noexcept;
        int operator= (int) noexcept;
        ...
    };
}
```

With `is_lock_free()`, you can check whether an atomic type internally uses locks to be atomic. If not, you have native hardware support for the atomic operations (which is a prerequisite for using atomics in signal handlers).

Both `compare_exchange_strong()` and `compare_exchange_weak()` are so-called *compare-and-swap* (*CAS*) operations. CPUs often provide this atomic operation to compare the contents of a memory location to a given value and, only if they are the same, modify the contents of that memory location to a given new value. This guarantees that the new value is calculated based on up-to-date information. The effect is something like the following pseudocode:

```
bool compare_exchange_strong (T& expected, T desired)
{
    if (this->load() == expected) {
        this->store(desired);
        return true;
    }
    else {
        expected = this->load();
        return false;
    }
}
```

Thus, if the value had been updated by another thread in the meantime, it returns `false` with the new value in `expected`.

The weak form may spuriously fail so that it returns `false` even when the expected value is present. But the weak form is sometimes more efficient than the strong version.

18.7.3. The C-Style Interface of Atomics

For the atomic proposal for C++, there was a corresponding proposal for C, which should provide the same semantics but could, of course, not use such specific C++ features as templates, references, and member functions. Therefore, the whole atomic interface has a C-style equivalent, which also was proposed as an extension to the C standard.

For example, you can also declare an `atomic<bool>` as `atomic_bool`, and instead of `store()` and `load()`, you can use global functions, which use a pointer to the object:

Click here to view code image

```
std::atomic_bool ab;            // equivalent to: std::atomic<bool> ab
std::atomic_init(&ab,false);    // see Section 18.7.1, page 1013
...
std::atomic_store(&ab,true);    // equivalent to: ab.store(true)
...
if (std::atomic_load(&ab)) {    // equivalent to: if (ab.load())
    ...
}
```

However, C added another interface, using `_Atomic` and `_Atomic()`, so the C-style interface in general is useful only for code that needs to be both C and C++ compilable in the nearer term.

However, using the C-style atomic types is pretty common in C++. Table 18.12 lists the most important atomic type names. There are more provided for less common types, such as `atomic_int_fast32_t` for `atomic<int_fast32_t>`.

Table 18.12. Some Named Types of *`std::atomic<>`*

Named Type	Corresponding Type
atomic_bool	atomic<bool>
atomic_char	atomic<char>
atomic_schar	atomic<signed char>
atomic_uchar	atomic<unsigned char>
atomic_short	atomic<short>
atomic_ushort	atomic<unsigned short>
atomic_int	atomic<int>
atomic_uint	atomic<unsigned int>
atomic_long	atomic<long>
atomic_ulong	atomic<unsigned long>
atomic_llong	atomic<long long>
atomic_ullong	atomic<unsigned long long>
atomic_char16_t	atomic<char16_t>
atomic_char32_t	atomic<char32_t>
atomic_wchar_t	atomic<wchar_t>
atomic_intptr_t	atomic<intptr_t>
atomic_uintptr_t	atomic<uintptr_t>
atomic_size_t	atomic<size_t>
atomic_ptrdiff_t	atomic<ptrdiff_t>
atomic_intmax_t	atomic<intmax_t>
atomic_uintmax_t	atomic<uintmax_t>

Note that for shared pointers (see Section 5.2.1, page 76) special atomic operations are provided. The reason is that a declaration, such as `atomic<shared_ptr< T >>` , is not possible, because a shared pointer is not trivially copyable. The atomic operations follow the naming conventions of the C-style interface. See Section 5.2.4, page 96, for details.

18.7.4. The Low-Level Interface of Atomics

The *low-level interface* of atomics means using the atomic operations in a way that we have no guaranteed sequential consistency. Thus, compilers and hardware might (partially) reorder access on atomics (see Section 18.4.3, page 986).

Beware again: Although I give an example, this area is a minefield. You need a lot of expertise to know when memory reorderings are worth the effort, and even experts often make mistakes in this area.[27]

[27] Special thanks to Hans Boehm and Bartosz Milewski for their support in letting me understand this and their help in providing the right wording. Any flaws are my fault.

An expert using this feature should be familiar with the material mentioned in *[N2480:MemMod]* and *[BoehmAdve:MemMod]* or, in general, all material listed at *[BoehmC++MM]*.

An Example for the Low-Level Interface of Atomics

Consider the second example for using atomics, introduced in Section 18.7.1, page 1014, where we declared an atomic flag to control access to some data:

```
long data;
std::atomic<bool> readyFlag(false);
```

and a thread providing the data:

```
data = 42;                    // provide some data
readyFlag.store(true);        // and signal readiness
```

and a thread consuming the data:

Click here to view code image

```
while (!readyFlag.load()) {       // loop until ready
    ...
}
std::cout << data << std::endl;   // and process provided data
```

Because we use the default memory order, which guarantees sequential consistency, this works as described in Section 18.7.1, page 1015. In fact, what we really call is:

```
data = 42;
readyFlag.store(true,std::memory_order_seq_cst);
```

and

```
while (!readyFlag.load(std::memory_order_seq_cst)) {
    …
}
std::cout << data << std::endl;
```

Thus, each operation has an optional argument to pass the memory order, which by default is `std::memory_order_seq_cst` (*sequential consistent memory order*).

By passing other values as memory order, we can weaken the order guarantees. In our case, it is, for example, enough to require that the provider not delay operations past the atomic store and that the consumer not bring forward operations following the atomic load:

```
data = 42;
readyFlag.store(true,std::memory_order_release);
```

and

```
while (!readyFlag.load(std::memory_order_acquire)) {
    …
}
std::cout << data << std::endl;
```

However, relaxing all constraints on the order of atomic operations would result in undefined behavior:

```
// ERROR: undefined behavior:
data = 42;
readyFlag.store(true,std::memory_order_relaxed);
```

The reason is that `std::memory_order_relaxed` doesn't guarantee that all prior memory operations become visible to other threads before the effect of the store operation. Thus, the provider might write `data` after setting the ready flag, so the consumer might read `data` while it gets written, which is a *data race*.

Note that you could also make `data` atomic and use `std::memory_order_relaxed` as memory order:

Click here to view code image

```
std::atomic<long> data(0);
std::atomic<bool> readyFlag(false);

// providing thread:
data.store(42,std::memory_order_relaxed);
readyFlag.store(true,std::memory_order_relaxed);

// consuming thread:
while (!readyFlag.load(std::memory_order_relaxed)) {
    …
}
std::cout << data.load(std::memory_order_relaxed) << std::endl;
```

Strictly speaking, this is not *undefined behavior*, because we don't have a *data race*. However, this also would not work as expected, because the resulting value of `data` might not be `42` yet (the memory order is still not guaranteed). It's behavior that results in `data` having an *unspecified* value.

Using `memory_order_relaxed` would be useful only if we have atomic variables where reads and/or writes are independent of one another. An example would be a global counter, which different threads might increment or decrement and where we need only the final value after all threads ended.

Overview of Low-Level Operations

Table 18.13 lists the supplementary low-level operations provided for atomics. As you can see, the load, store, exchange, CAS, and fetch operations provide the supplementary ability to pass a memory order as an additional argument.

Table 18.13. Supplementary Low-Level Operations of Atomics

Operation	*triv*	*int type*	*ptr type*
a.store(*val*,*mo*)	Yes	Yes	Yes
a.load(*mo*)	Yes	Yes	Yes
a.exchange(*val*,*mo*)	Yes	Yes	Yes
a.compare_exchange_strong(*exp*,*des*,*mo*)	Yes	Yes	Yes
a.compare_exchange_strong(*exp*,*des*,*mo1*,*mo2*)	Yes	Yes	Yes
a.compare_exchange_weak(*exp*,*des*,*mo*)	Yes	Yes	Yes
a.compare_exchange_weak(*exp*,*des*,*mo1*,*mo2*)	Yes	Yes	Yes
a.fetch_add(*val*,*mo*)		Yes	Yes
a.fetch_sub(*val*,*mo*)		Yes	Yes
a.fetch_and(*val*,*mo*)		Yes	
a.fetch_or(*val*,*mo*)		Yes	
a.fetch_xor(*val*,*mo*)		Yes	

Some additional functions are provided to manually control memory access. For example, `atomic_thread_fence()` and `atomic_signal_fence()` are provided to manually program fences, which are barriers for memory-access reordering.

No More Details

I *don't* explain these low-level interfaces in more detail because this feature is for real concurrency experts or those who want to become experts. So, you should definitely use specific resources for that.

One good starting point is Anthony Williams book *C++ Concurrency in Action* (see [*Williams:C++Conc*]), especially Chapters 5 and 7. Another is Hans Boehm's list of URLs for material about memory models (see [*Boehm:C++MM*]).