**5.2 Atomic operations and types in C++**

An atomic operation is an indivisible operation. It’s either done or not done.

If the non-atomic operation is composed of atomic operations (for example, assignment to a struct with atomic members), you might observe or end up with a value that is a mixed-up combination of the various values stored.

# 5.2.1 The standard atomic types

All operations on atomic types are atomic, and only operations on these types are atomic in the sense of the language definition.

For an atomic type is\_lock\_free() member function

returns true, if operations on a given type are done directly with atomic instructions

returns false, if operations on a given type are done using a lock internal to the compiler and library

The library provides a set of macros to identify at compile time whether the atomic types for the various integral types are lock-free.

Since C++17, all atomic types have a static constexpr member variable, X::is\_always\_lock\_free

ATOMIC\_BOOL\_LOCK\_FREE,

ATOMIC\_CHAR\_LOCK\_FREE,ATOMIC\_CHAR16\_T\_LOCK\_FREE, ATOMIC\_CHAR32\_T\_LOCK\_FREE, ATOMIC\_WCHAR\_T\_LOCK\_FREE,

ATOMIC\_SHORT\_LOCK\_FREE,ATOMIC\_INT\_LOCK\_FREE, ATOMIC\_LONG\_LOCK\_FREE, ATOMIC\_LLONG\_LOCK\_FREE,

and ATOMIC\_POINTER\_LOCK\_FREE

evaluate to the value

* 0 if the atomic type is never lock-free
* 1 if the lock-free status of the corresponding atomic type is a runtime property
* 2 if the atomic type is always lock-free

std::atomic\_flag is a simple Boolean flag, and operations on this type are required to be lock-free

std::atomic<> class templates are a bit more full-featured but may not be lockfree

You can use the set of names, to refer to the implementation-supplied atomic types

C++17, these are always aliases for the corresponding std::atomic<> specializations

The **standard atomic types are not copyable or assignable in the conventional sense**

std::atomic<> can be used to create an atomic variant of a user defined type.

Because it’s a generic class template, the operations are limited to load(), store() (and assignment from and conversion to the user-defined type), exchange(), compare\_exchange\_weak(), and compare\_exchange\_strong().

Each of the operations on the atomic types has an optional memory-ordering argument which is one of the values of the std::memory\_order enumeration.

The operations are divided into three categories:

1. **Store operations**, which can have memory\_order\_relaxed, memory\_order\_release, or memory\_order\_seq\_cst ordering
2. **Load operations**, which can have memory\_order\_relaxed, memory\_order\_consume, memory\_order\_acquire, or memory\_order\_seq\_cst ordering
3. **Read-modify-write operation**s, which can have memory\_order\_relaxed, memory\_order\_consume, memory\_order\_acquire, memory\_order\_release, memory\_order\_acq\_rel, or memory\_order\_seq\_cst ordering

# 5.2.2 Operations on std::atomic\_flag

**std::atomic\_flag**

* represents a Boolean flag
* two states: set or clear
* Objects of the std::atomic\_flag type must be initialized to a clear state with ATOMIC\_FLAG\_INIT
* the flag always starts clear

std::atomic\_flag f = ATOMIC\_FLAG\_INIT;

* only atomic type to require such special treatment for initialization
* guaranteed to be lock-free

after initialization, only three things you can do with it:

1. destroy it,
2. clear it, or
3. set it and query the previous value

clear() is a store operation and so can’t have memory\_order\_acquire or memory\_order\_acq\_rel semantics

test\_and\_set() is a read-modify-write operation and so can have any of the memory-ordering

default for both is memory\_order\_seq\_cst

std::atomic\_flag ideally suited to use as a spinlock mutex

By its nature it does a busy-wait in lock()

it can’t even be used as a general Boolean flag, because it doesn’t have a simple non modifying query operation

class spinlock\_mutex {

std::atomic\_flag flag;

public:

spinlock\_mutex() : flag(ATOMIC\_FLAG\_INIT) {}

void lock() {

while (flag.test\_and\_set(std::memory\_order\_acquire));

}

void unlock() {

flag.clear(std::memory\_order\_release);

}

};

For C++20 changes visit <https://en.cppreference.com/w/cpp/atomic/atomic_flag>

# 5.2.3 Operations on std::atomic<bool>

* more fullfeatured Boolean flag
* can construct it from a non-atomic bool, so it can be initially true or false
* can also assign to instances of std::atomic<bool> from a non-atomic bool

std::atomic<bool> b(true);

b=false;

another common pattern with the atomic types: the assignment operators they support return values (of the corresponding non-atomic type) rather than references

**store(),** to write

**exchange(),**to replace the stored value with a new one of your choosing and atomically retrieve the original value

exchange() is a read-modify-write operation

**load(),** to query of the value, also supports a plain nonmodifying implicit conversion to plain bool

std::atomic<bool> b;

bool x=b.load(std::memory\_order\_acquire);

b.store(true);

x=b.exchange(false,std::memory\_order\_acq\_rel);

## Storing A New Value (Or Not) Depending On The Current Value

**compare\_exchange\_weak() and compare\_exchange\_strong()**

compares the value of the atomic variable with a supplied expected value and if they’re

equal stores the supplied desired value

not equal the expected value is updated with the value of the atomic variable

return

true if the store was performed and operation succeed

false otherwise

For compare\_exchange\_weak(), the store might fail even if the original value was equal to the expected value

This is called a spurious failure, because the reason for the failure is a function of timing rather than the values of the variables

compare\_exchange\_strong() is guaranteed to return false only if the value wasn’t equal to the expected value

compare-exchange functions are also unusual in that they can take two memoryordering parameters

it might be desirable for a successful call to have memory\_order\_acq\_rel semantics, whereas a failed call has memory\_order\_relaxed semantics

You also can’t supply stricter memory ordering for failure than for success

If you don’t specify an ordering for failure, it’s assumed to be the same as that for success, except that the release part of the ordering is stripped:

Following two calls to compare\_exchange\_weak() are equivalent:

std::atomic<bool> b;

bool expected;

b.compare\_exchange\_weak(expected,true, memory\_order\_acq\_rel,memory\_order\_acquire);

b.compare\_exchange\_weak(expected,true,memory\_order\_acq\_rel);

# References

Anthony Williams - C++ Concurrency in Action

https://en.cppreference.com/w/cpp/atomic/atomic\_flag