Chapter 45. Boost.Atomic

Boost.Atomic provides the class `boost::atomic`, which can be used to create atomic variables. They are called atomic variables because all access is atomic. Boost.Atomic is used in multithreaded programs when access to a variable in one thread shouldn’t be interrupted by another thread accessing the same variable. Without `boost::atomic`, attempts to access shared variables from multiple threads would need to be synchronized with locks.

`boost::atomic` depends on the target platform supporting atomic variable access. Otherwise, `boost::atomic` uses locks. The library allows you to detect whether a target platform supports atomic variable access.

If your development environment supports C++11, you don’t need Boost.Atomic. The C++11 standard library provides a header file `atomic` that defines the same functionality as Boost.Atomic. For example, you will find a class named `std::atomic`.

Boost.Atomic supports more or less the same functionality as the standard library. While a few functions are overloaded in Boost.Atomic, they may have different names in the standard library. The standard library also provides some functions, such as `std::atomic_init()` and `std::kill_dependency()`, which are missing in Boost.Atomic.

Example 45.1. Using `boost::atomic`

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  ++a;
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Example 45.1 uses two threads to increment the `int` variable **a**. Instead of a lock, the example uses `boost::atomic` for atomic access to **a**. The example writes `2` to standard output.

`boost::atomic` works because some processors support atomic access on variables. If incrementing an `int` variable is an atomic operation, a lock isn’t required. If this example is run on a platform that cannot increment a variable as an atomic operation, `boost::atomic` uses a lock.

Example 45.2. `boost::atomic` with or without lock

```cpp
#include <boost/atomic.hpp>
#include <iostream>

int main()
{
  std::cout.setf(std::ios::boolalpha);

  boost::atomic<short> s;
  std::cout << s.is_lock_free() << '\n';

  boost::atomic<int> i;
  std::cout << i.is_lock_free() << '\n';

  boost::atomic<long> l;
  std::cout << l.is_lock_free() << '\n';
}
```

You can call `is_lock_free()` on an atomic variable to check whether accessing the variable is done without a lock. If you run the example on an Intel x86 processor, it displays `true` three times. If you run it on a processor without lock-free access on `short`, `int` and `long` variables, `false` is displayed.

Boost.Atomic provides the `BOOST_ATOMIC_INT_LOCK_FREE` and `BOOST_ATOMIC_LONG_LOCK_FREE` macros to check, at compile time, which data types support lock-free access.

Example 45.2 uses integral data types only. You should not use `boost::atomic` with classes like `std::string` or `std::vector`. Boost.Atomic supports integers, pointers, booleans (`bool`), and trivial classes. Examples of integral types include `short`, `int` and `long`. Trivial classes define objects that can be copied with `std::memcpy()`.

Example 45.3. `boost::atomic` with **`boost::memory_order_seq_cst`**

```cpp
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  a.fetch_add(1, boost::memory_order_seq_cst);
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Example 45.3 increments **a** twice – this time not with `operator++` but with a call to `fetch_add()`. The member function `fetch_add()` can take two parameters: the number by which **a** should be incremented and the *memory order*.

The memory order specifies the order in which access operations on memory must occur. By default, this order is undetermined and does not depend on the order of the lines of code. Compilers and processors are allowed to change the order as long as a program behaves as if memory access operations were executed in source code order. This rule only applies to a thread. If more than one thread is used, variations in the order of memory accesses can lead to a program acting erroneously. Boost.Atomic supports specifying a memory order when accessing variables to make sure memory accesses occur in the desired order in a multithreaded program.

Note

Specifying memory order optimizes performance, but it increases complexity and makes it more difficult to write correct code. Therefore, in practice, you should have a really good reason for using memory orders.

Example 45.3 uses the memory order **boost::memory_order_seq_cst** to increment **a** by 1. The memory order stands for *sequential consistency*. This is the most restrictive memory order. All memory accesses that appear before the `fetch_add()` call must occur before this member function is executed. All memory accesses that appear after the `fetch_add()` call must occur after this member function is executed. Compilers and processors may reorder memory accesses before and after the call to `fetch_add()`, but they must not move a memory access from before to after the call to `fetch_add()` or vice versa. **boost::memory_order_seq_cst** is a strict boundary for memory accesses in both directions.

boost::memory_order_seq_cst is the most restrictive memory order. It is used by default when memory order is not set. Therefore, in Example 45.1, when **a** is incremented with `operator++`, **boost::memory_order_seq_cst** will be used.

boost::memory_order_seq_cst isn't always necessary. For example, in Example 45.3 there is no need to synchronize memory accesses for other variables because **a** is the only variable the threads use. **a** is written to standard output in `main()`, but only after both threads have terminated. The call to `join()` guarantees that **a** is only read after both threads have finished.

Example 45.4. `boost::atomic` with **boost::memory_order_relaxed**

```cpp
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  a.fetch_add(1, boost::memory_order_relaxed);
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Example 45.4 sets the memory order to **boost::memory_order_relaxed**. This is the least restrictive memory order: it allows arbitrary reordering of memory accesses. This example works with this memory order because the threads access no variables except **a**. Therefore, no specific order is required.

Example 45.5. boost::atomic with **memory_order_release** and **memory_order_acquire**

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};
int b = 0;

void thread1()
{
  b = 1;
  a.store(1, boost::memory_order_release);
}

void thread2()
{
  while (a.load(boost::memory_order_acquire) != 1)
    ;
  std::cout << b << '\n';
}

int main()
{
  std::thread t1{thread1};
  std::thread t2{thread2};
  t1.join();
  t2.join();
}
```

There are choices between the most restrictive memory order, **boost::memory_order_seq_cst**, and the least restrictive one, **boost::memory_order_relaxed**. Example 45.5 introduces the memory orders **boost::memory_order_release** and **boost::memory_order_acquire**.

Memory accesses that appear in the code before the **boost::memory_order_release** statement are executed before the **boost::memory_order_release** statement is executed. Compilers and processors must not move memory accesses from before to after **boost::memory_order_release**. However, they may move memory accesses from after to before **boost::memory_order_release**.

boost::memory_order_acquire works like **boost::memory_order_release**, but refers to memory accesses after **boost::memory_order_acquire**. Compilers and processors must not move memory accesses from after the **boost::memory_order_acquire** statement to before it. However, they may move memory accesses from before to after **boost::memory_order_acquire**.

Example 45.5 uses **boost::memory_order_release** in the first thread to make sure that **b** is set to 1 before **a** is set to 1. **boost::memory_order_release** guarantees that the memory access on **b** occurs before the memory access on **a**.

To specify a memory order when accessing `a`, `store()` is called. This member function corresponds to an assignment with `operator=`.

The second thread reads `a` in a loop. This is done with the member function `load()`. Again, no assignment operator is used.

In the second thread, `boost::memory_order_acquire` makes sure that the memory access on `b` doesn't occur before the memory access on `a`. The second thread waits in the loop for `a` to be set to 1 by the first thread. Once this happens, `b` is read.

The example writes `1` to standard output. The memory orders ensure that all memory accesses occur in the right order. The first thread always writes 1 to `b` first before the second thread accesses and reads `b`.

Tip

For more details and examples on how memory orders work, you can find an explanation in the GCC Wiki article on memory model synchronization modes.