std::memory\_order\_consume is the most legendary of the[six memory models](http://modernescpp.com/index.php/synchronization-and-ordering-constraints). That's for two reasons. At one hand, std::memory\_order\_consume is extremely hard to get. At the other hand - that may change in the future - no compiler supports it.

How can it happen, that a compiler supports the C++11 standard, but doesn't support the memory model std::memory\_order\_consume? The answer is, that compiler maps std::memory\_order\_consume to std::memory\_order\_acquire. That is fine because both are load or acquire operations. std::memory\_order\_consume requires weaker [synchronisation and ordering constraints](http://modernescpp.com/index.php/synchronization-and-ordering-constraints). So the release-acquire ordering is potentially slower than the release-consume ordering but - that is the key point - well defined.

To get an understanding of the release-consume ordering, it's a good idea to compare it with the release-acquire ordering. I speak in the post explicitly from the release-acquire ordering and not from the [acquire-release semantic](http://modernescpp.com/index.php/acquire-release-semantic) to emphasise the strong relationship of std::memory\_order\_consume and std::memory\_order\_acquire.

Release-acquire ordering

As starting point I use a program with two threads t1 and t2. t1 plays the role of the producer, t2 the role of the consumer. The atomic variable ptr helps to synchronize the producer and consumer.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | // acquireRelease.cpp  #include <atomic>  #include <thread>  #include <iostream>  #include <string>    std::atomic<std::string\*> ptr;  int data;  std::atomic<int> atoData;    void producer(){  std::string\* p = new std::string("C++11");  data = 2011;  atoData.store(2014,std::memory\_order\_relaxed);  ptr.store(p, std::memory\_order\_release);  }    void consumer(){  std::string\* p2;  while (!(p2 = ptr.load(std::memory\_order\_acquire)));  std::cout << "\*p2: " << \*p2 << std::endl;  std::cout << "data: " << data << std::endl;  std::cout << "atoData: " << atoData.load(std::memory\_order\_relaxed) << std::endl;  }    int main(){    std::cout << std::endl;    std::thread t1(producer);  std::thread t2(consumer);    t1.join();  t2.join();    std::cout << std::endl;    } |

Before I analyse the program, I want to introduce a small variation. I replace in line 21 the memory model std::memory\_order\_acquire by std::memory\_order\_consume.

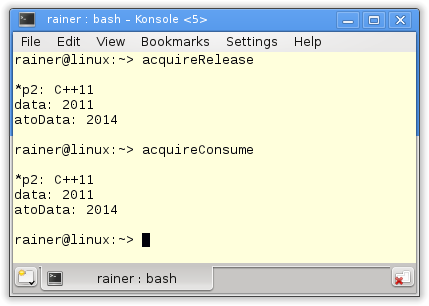
Release-consume ordering

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | // acquireConsume.cpp  #include <atomic>  #include <thread>  #include <iostream>  #include <string>    std::atomic<std::string\*> ptr;  int data;  std::atomic<int> atoData;    void producer(){  std::string\* p = new std::string("C++11");  data = 2011;  atoData.store(2014,std::memory\_order\_relaxed);  ptr.store(p, std::memory\_order\_release);  }    void consumer(){  std::string\* p2;  while (!(p2 = ptr.load(std::memory\_order\_consume)));  std::cout << "\*p2: " << \*p2 << std::endl;  std::cout << "data: " << data << std::endl;  std::cout << "atoData: " << atoData.load(std::memory\_order\_relaxed) << std::endl;  }    int main(){    std::cout << std::endl;    std::thread t1(producer);  std::thread t2(consumer);    t1.join();  t2.join();    std::cout << std::endl;    } |

That was easy. But now the program has undefined behaviour. That statement is very hypothetical, because my compiler implements std::memory\_order\_consume by std::memory\_order\_acquire. So under the hood both program actually do the same.

Release-acquire versus Release-consume ordering

The output of the programs is identical.



Although I repeat myself, I want to sketch in a few words, why the first program acquireRelease.cpp is well defined.

The store operation in line 16 synchronizes-with the load operation in line 21. The reason is, that the store operation uses std::memory\_order\_release, that the load operation uses std::memory\_order\_acquire. That was the synchronization. What's about the ordering constraints of the release-acquire ordering? The release-acquire ordering guarantees, that all operations before the store operation (line 16) are available after the load operation (line 21). So the release-acquire operation orders in addition the access on the non-atomic variable data (line 14) and the atomic variable atoData (line 15). That holds, although atoData uses the std::memory\_order\_relaxed memory model.

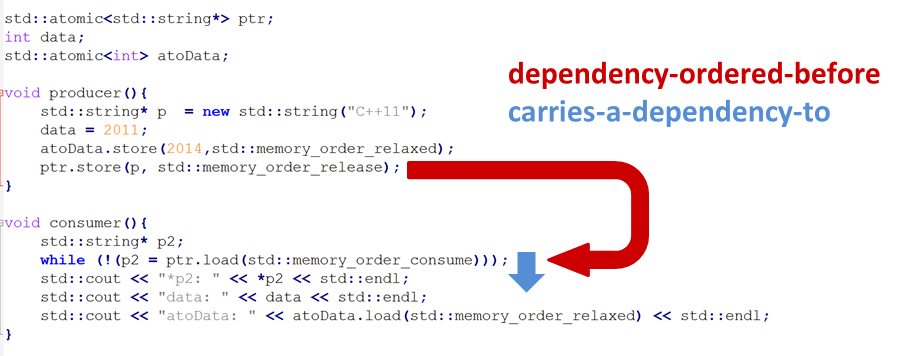
The key question is. What happens, if I replace in the program std::memory\_order\_acquire by std::memory\_order\_consume?

Data dependencies with std::memory\_order\_consume

The std::memory\_order\_consume is about data dependencies on atomics. Data dependencies exist in two ways. At first *carries-a-dependency-to* in a thread and *dependency-ordered\_before* between two threads. Both dependencies introduce a *happens-before* relation. That is this kind of relation a well defined program needs.  But what means *carries-a-dependency-to* and *dependency-order-before*?

* ***carries-a-dependency-to:***If the result of an operation A is used as an operand of an operation B, then: A*carries-a-dependency-to* B.
* ***dependecy-ordered-before:***A store operation (with std::memory\_order\_release, std::memory\_order\_acq\_rel or std::memory\_order\_seq\_cst), is *dependency-ordered-before* a load operation B (with std::memory\_order\_consume), if the result of the load operation B is used in a further operation C in the same thread. The operations B and C have to be in the same thread.

Of course I know from personal experience, that both definitions are not easy to digest. So I will use a graphic to visually explain them.



The expression ptr.store(p, std::memory\_order\_release) is *dependency-ordered-before* while (!(p2 = ptr.load(std::memory\_order\_consume))), because in the following line std::cout << "\*p2: " << \*p2 << std::endl the result of the load operation will be read. Further, holds: while (!(p2 = ptr.load(std::memory\_order\_consume)) *carries-a-dependency-to* std::cout << "\*p2: " << \*p2 << std::endl, because the output of \*p2 uses the result of the ptr.load operation.

But we have no guarantee for the following outputs of data and atoData. That's because both have no *carries-a-dependency* relation to the ptr.load operation. But it gets even worse. Because data is a non-atomic variable, there is a [race condition](http://modernescpp.com/index.php/threads-sharing-data)on data. The reason is, that both threads can access data at the same time and thread t1 wants to modify data. Therefore, the program is undefined.