Memory order in C++

memory_order_relaxed	memory_order_consume	memory_order_acquire	memory_order_release	memory_order_acq_rel	memory_order_seq_cst
Relaxed operation: there are no synchronization or ordering constraints imposed on other reads or writes, only this operation's atomicity is guaranteed	A load operation with this memory order performs a *consume operation* on the affected memory location: no reads or writes in the current thread dependent on the value currently loaded can be reordered **before** this load. Writes to **data-dependent** variables in other threads that release the same atomic variable are visible in the current thread. On most platforms, this affects compiler optimizations only.	A load operation with this memory order performs the *acquire operation* on the affected memory location: no reads or writes in the current thread can be reordered **before** this load. All writes in other threads that **release** the same atomic variable are visible in the current thread	A store operation with this memory order performs the *release operation*: no reads or writes in the current thread can be reordered **after** this store. All writes in the current thread are visible in other threads that acquire the same atomic variable and writes that carry a **dependency** into the atomic variable become visible in other threads that consume the same atomic.	A **read-modify-write** operation with this memory order is both an *acquire operation* and a *release operation*. No memory reads or writes in the current thread can be reordered before or after this store. All writes in other threads that release the same atomic variable are visible before the modification and the modification is visible in other threads that acquire the same atomic variable.	**Any** operation with this memory order is both an *acquire operation* and a *release operation*, plus a single total order exists in which all threads observe all modifications in the same order
	Special operation that is like memory_order_acquire, but for operations that are **dependent** on the current variable. It seems compilers don't implement this well.	I have eg. atomic flag that says whether some data are valid. I should use this memory order to read the flag, because reading from the other variables will not be reordered before the read of the flag.	Eg. I have following statement: p.store(new X(5, 2), …) I don't want the stores inside X to be applied after store to the pointer. Question: It is based on dependency, so can this be used for boolean flag? Like: x = 5; flag = true? There is no dependency! :-(	Like memory_order_seq_cst, but for read-modify-write only.	

Nice article about memory-order consume: http://preshing.com/20141124/fixing-gccs-implementation-of-memory_order_consume/