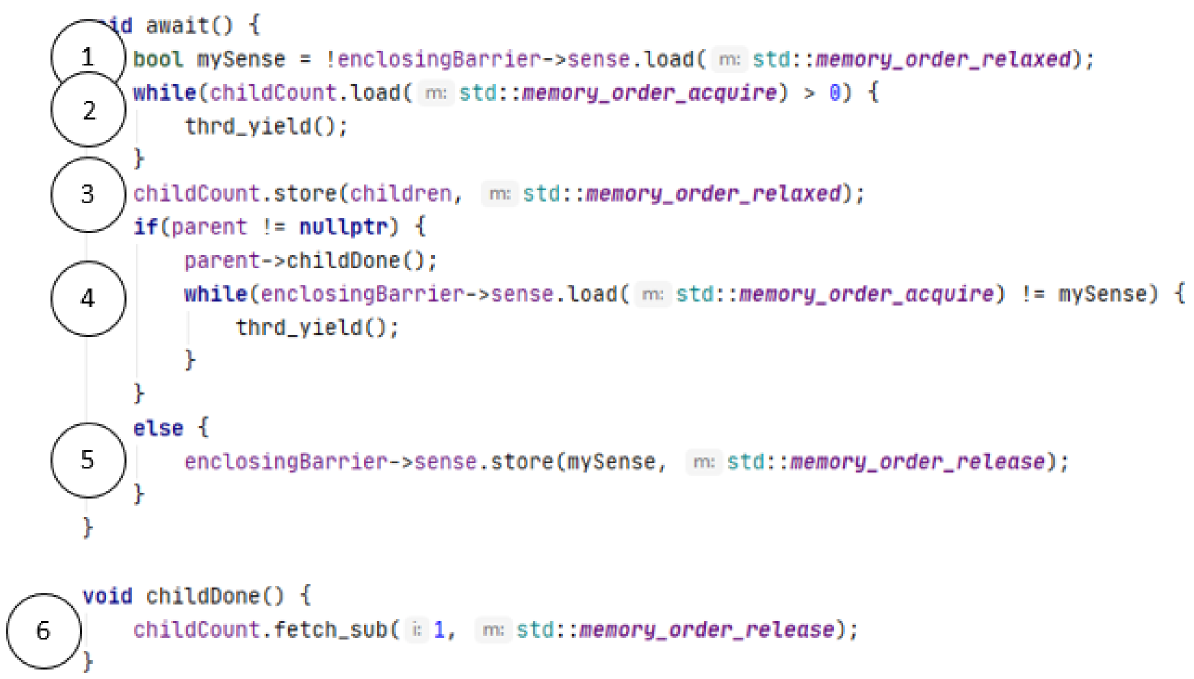
Two methods of `StaticTreeBarrier::Node` class:



```
    id await() {
1     bool mySense = !enclosingBarrier->sense.load( m: std::memory_order_relaxed);
2     while(childCount.load( m: std::memory_order_acquire) > 0) {
          thrd_yield();
      }
3     childCount.store(children,  m: std::memory_order_relaxed);
      if(parent != nullptr) {
          parent->childDone();
4         while(enclosingBarrier->sense.load( m: std::memory_order_acquire) != mySense) {
              thrd_yield();
          }
      }
      else {
5         enclosingBarrier->sense.store(mySense,  m: std::memory_order_release);
      }
  }

  void childDone() {
6     childCount.fetch_sub( i: 1,  m: std::memory_order_release);
  }
```

Why the barrier implementation is correct:

My barrier implementation ported to C++ is almost an exact copy of the Java version from the book's section 17.5, so I will just describe the differences in my C++ implementation when referring to its correctness. The differences are the additionally specified memory orderings and the lack of the C++ equivalent of `ThreadLocal<Boolean> threadSense` object in `StaticTreeBarrier` class. `thread_local` class members must be declared `static` in C++ and it would corrupt the barrier data structure if multiple barrier objects were used; in addition, I realized that this object is not actually needed in the static tree barrier data structure at all, regardless of the language, because we could simply read a negated value from `enclosingBarrier->sense` in line **(1)** of the above snippet. (An `enclosingBarrier` object is needed in each node due to the fact that in C++, unlike in Java, we can't refer to the members of the outer class if they're not static).

Let's consider the usage of the barrier object (and corresponding `Node` objects) for the first time after it's been constructed. The whole computation depicted above has essentially two phases. Phase one consists of multiple transitive happens-before relations, where each of the children nodes establishes **(6) happens-before (2)** with its parent. The phase two happens upon completion of phase one, and that phase consists simply of the root signaling the end of a computation phase to all the other threads, thus establishing **(5) happens-before (4)** relations with all of them. Because **(3)** is between those two happens-before phases, it can be relaxed. The same goes for **(1)** – having constructed the barrier object, it will read the value of the newly constructed `sense` object, and in subsequent calls it will read the value stored by the previous barrier phase, since it will appear sequenced after **(4)**, which is part of a synchronizes-with relation. Since phase two happens after the first one, the barrier entrance of all the threads (which is the equivalent of my phase one) happens before the barrier exits of all the threads (which is the equivalent of my phase two), therefore the barrier works as expected.

The first test case uses the barrier for coordinating 3 threads in a tree where nodes have 2 children, the second test case coordinates 4 threads in a tree where nodes have 3 children. Each test case simply checks if threads don't enter the next phase of computation prematurely, thus causing data races on shared variables.