```
// my_app.cpp:
#include <my_shareddata.h>
#include <thread>  // std::thread

int main()
{
    std::thread t2(accessSharedData);
    std::thread t1(  initSharedData);

    t1.join();
    t2.join();
}
```

When this *release-acquire synchronization paradigm* is used, the compiler must maintain the statements' ordering to avoid breaking the *release-acquire* guarantee; the compiler must also insert memory-fence instructions to prevent the hardware from breaking this guarantee.

If we wanted to modify the example above to use *release-consume* semantics, we would somehow need to make the `assert` statements a part of the dependency chain on the `load` from the `guard` object. We can accomplish this goal because reading data through a pointer establishes a dependency chain between the reading of that pointer value and the reading of the referenced data. Since *release-consume* allows the developer to specify that data of concern, using that policy instead of *release-acquire* policy (in the code example above) allows the compiler to be more selective in its use of memory fences:

```
// my_shareddata.cpp (use *consume, not *_acquire):
#include <my_shareddata.h>

#include <atomic>  // std::atomic, std::memory_order_release, and
                   // std::memory_order_consume (not *_acquire)
#include <cassert> // standard C assert macro

struct S
{
    /* definition not changed */
};

static S               data;           // static for insulation (as before)
static std::atomic<S*> guard(nullptr); // guards just one struct S.

void initSharedData()
{
    data.i = 42;   // as before
    data.c = 'c';  // as before
    data.d = 5.0;  // as before

    guard.store(&data, std::memory_order_release);  // Set &data, not 1.
}
```