```
void accessSharedData()
{
    S* sharedDataPtr = nullptr;

    // Load using *_consume, not *_acquire.
    while (nullptr == (sharedDataPtr = guard.load(std::memory_order_consume)))
        /* empty */ ;

    assert(&data == sharedDataPtr);

    assert(42  == sharedDataPtr->i);
    assert('c' == sharedDataPtr->c);
    assert(5.0 == sharedDataPtr->d);
}
```

Finally, if we want to start to refactor the work of the `my_sharedata` component into multiple functions across different translation units, we would want to carefully apply the `[[carries_dependency]]` attribute to the newly refactored functions, so calling into these functions might conceivably be better optimized:

```
// my_shareddataimpl.h:

struct S
{
    int    i;
    char   c;
    double d;
};

[[carries_dependency]] S* getSharedDataPtr();
    // Return the address of the shared data in this translation unit.

void releaseSharedData(S* sharedDataPtr [[carries_dependency]]);
    // Release the shared data in this translation unit.  The behavior is
    // undefined unless getSharedDataPtr() == sharedDataPtr.

[[carries_dependency]] S* accessInitializedSharedData();
    // Return the address of the initialized shared data in this translation
    // unit.

void checkSharedDataValue(S* s [[carries_dependency]],
                          int    i,
                          char   c,
                          double d);
    // Confirm that data at the specified s has the specified i, c, and
    // d as constituent values.
```

```
// my_shareddataimpl.cpp:

#include <my_shareddataimpl.h>

#include <cassert>  // standard C offsetof macro
#include <atomic>   // std::atomic, std::memory_order_*

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

[[carries_dependency]] S* getSharedDataPtr()
{
    return &data;
}

void releaseSharedData(S* sharedDataPtr [[carries_dependency]])
{
    assert(&data == sharedDataPtr);

    guard.store(sharedDataPtr, std::memory_order_release);
}

[[carries_dependency]] S* accessInitializedSharedData()
{
    S* sharedDataPtr = nullptr;

    while (nullptr == (sharedDataPtr = guard.load(std::memory_order_consume)))
        /* empty */ ;

    assert(&data == sharedDataPtr);

    return sharedDataPtr;
}

void checkSharedDataValue(S*     s [[carries_dependency]],
                          int    i,
                          char   c,
                          double d)
{
    assert(i == s->i);
    assert(c == s->c);
    assert(d == s->d);
}
```