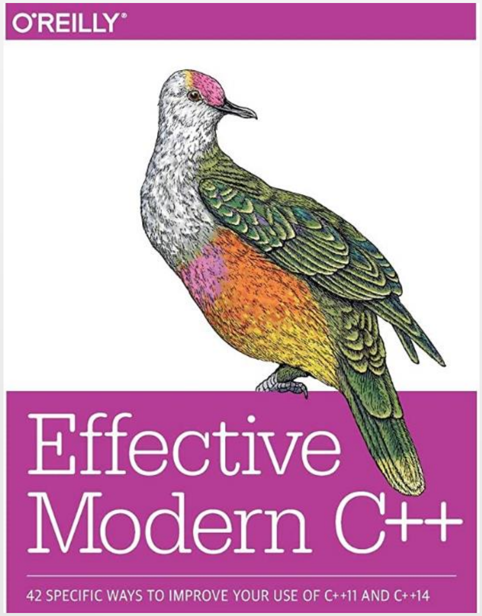
ITEM 16:
Make **const** member functions thread safe

Scott Meyers - Effective Modern C++



Presented By
@7eRoM

- Consider a polynomials class contains a member function computes the root(s) of a polynomial:

```
class Polynomial {
public:
  using RootsType =           // data structure holding values
    std::vector<double>;      // where polynomial evals to zero
  …                           // (see Item 9 for info on "using")

  RootsType roots() const;


  …

};
```

- Computing the roots of a polynomial can be expensive, so:
  - we don’t want to do it if we don’t have to.
  - if we do have to do it, we certainly don’t want to do it more than once.

```cpp
class Polynomial {
public:
  using RootsType = std::vector<double>;

  RootsType roots() const
  {
    if (!rootsAreValid) {            // if cache not valid

      …                              // compute roots,
                                     // store them in rootVals

      rootsAreValid = true;
    }

    return rootVals;
  }

private:
  mutable bool rootsAreValid{ false };    // see Item 7 for info
  mutable RootsType rootVals{};           // on initializers
};
```

- Imagine now that two threads simultaneously call roots on a Polynomial object:

```
Polynomial p;

…

/*-----  Thread 1  ----- */      /*-------  Thread 2  ------- */

auto rootsOfP = p.roots();       auto valsGivingZero = p.roots();
```

- roots is a const member function, and that means it represents a read operation. Having multiple threads perform a read operation without synchronization is safe.
- In case of write operation, due to the lack of synchronization, the code has undefined behavior.
- The problem is that roots is declared const, but it's not thread safe.

```
class Polynomial {
public:
  using RootsType = std::vector<double>;

  RootsType roots() const
  {
    std::lock_guard<std::mutex> g(m);     // lock mutex

    if (!rootsAreValid) {                 // if cache not valid

      …                                   // compute/store roots

      rootsAreValid = true;
    }

    return rootVals;
  }                                       // unlock mutex

private:
  mutable std::mutex m;
  mutable bool rootsAreValid{ false };
  mutable RootsType rootVals{};
};
```

- It’s worth noting that because std::mutex is a move-only type (i.e., a type that can be moved, but not copied), a side effect of adding m to Polynomial is that Polynomial loses the ability to be copied. It can still be moved, however.
- In some situations, a mutex is overkill.
- For example, if all you’re doing is counting how many times a member function is called, a std::atomic counter will often be a less expensive way to go.

```
class Point {                                  // 2D point
public:
  …

  double distanceFromOrigin() const noexcept     // see Item 14
  {                                              // for noexcept

    ++callCount;                               // atomic increment

    return std::sqrt((x * x) + (y * y));
  }

private:
  mutable std::atomic<unsigned> callCount{ 0 };
  double x, y;
};
```

- Like std::mutexes, std::atomics are move-only types, so the existence of call Count in Point means that Point is also move-only.

- Because operations on std::atomic variables are often less expensive than mutex acquisition and release, you may be tempted to lean on std::atomics more heavily than you should.
    - For example, in a class caching an expensive-to-compute int, you might try to use a pair of std::atomic variables instead of a mutex

```cpp
class Widget {
public:
  …

  int magicValue() const
  {
    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cachedValue = val1 + val2;             // uh oh, part 1
      cacheValid = true;                     // uh oh, part 2
      return cachedValue;
    }
  }

private:

   mutable std::atomic<bool> cacheValid{ false };
   mutable std::atomic<int> cachedValue;
 };
```

- A thread calls Widget::magicValue, sees cacheValid as false, performs the two expensive computations, and assigns their sum to cachedValue.
- At that point, a second thread calls Widget::magicValue, also sees cacheValid as false, and thus carries out the same expensive computations that the first thread has just finished. (This "second thread" may in fact be several other threads.)

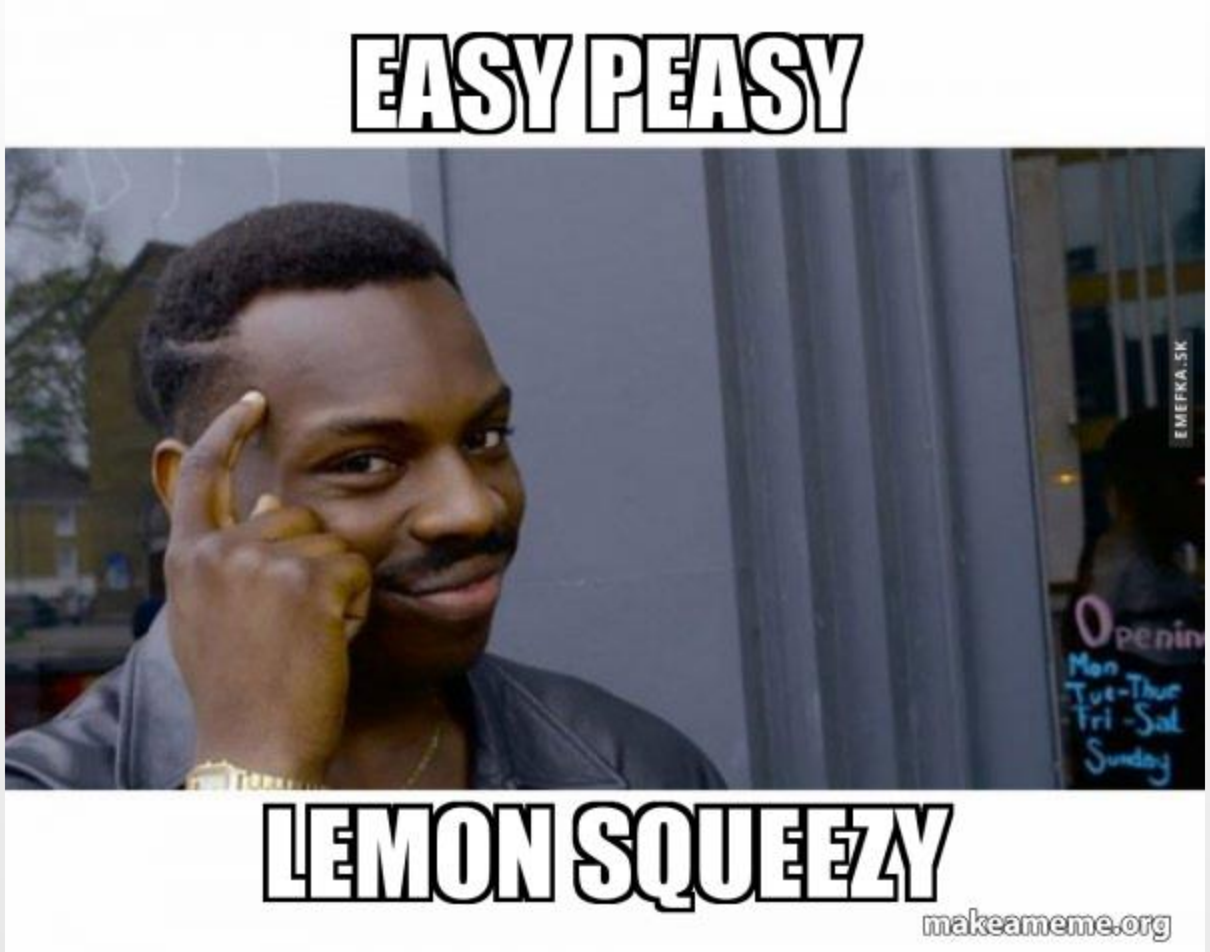
```
class Widget {
public:
  …

  int magicValue() const
  {
    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cacheValid = true;                    // uh oh, part 1
      return cachedValue = val1 + val2;     // uh oh, part 2
    }
  }

  …

};
```

- Imagine that cacheValid is false, and then:
  - One thread calls Widget::magicValue and executes through the point where cacheValid is set to true.
  - At that moment, a second thread calls Widget::magicValue and checks cacheValid. Seeing it true, the thread returns cachedValue, even though the first thread has not yet made an assignment to it. The returned value is therefore incorrect.

```cpp
class Widget {
public:
  …

  int magicValue() const
  {
    std::lock_guard<std::mutex> guard(m);   // lock m

    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cachedValue = val1 + val2;
      cacheValid = true;
      return cachedValue;
    }
  }                                         // unlock m
  …

private:
  mutable std::mutex m;
  mutable int cachedValue;                  // no longer atomic
  mutable bool cacheValid{ false };         // no longer atomic
};
```

- For a single variable or memory location requiring synchronization, use of a std::atomic is adequate.

- but once you get to two or more variables or memory locations that require manipulation as a unit, you should reach for a mutex.


EASY PEASY
LEMON SQUEEZY
makeameme.org