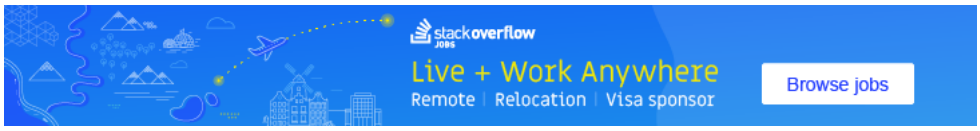
how to use std::atomic<>



I have a class that I wan to use it in different threads and I think I may be able to use std::atomic such as this:

```
classs A
{
    int x;

public:
    A()
    {
        x=0;
    }

    void Add()
    {
        x++;
    }

    void Sub()
    {
        x--;
    }
};
```

and in my code:

```
  std::atomic<A> a;
```

and in different thread:

```
  a.Add();
```

and

```
  a.Sub();
```

but when I am getting error that a.Add() is not known. How can i achieve this?

Is there any better way to do this?

Edit 1

Please note that it is a sample example, and what I want is to make sure that access to class A is threadsafe, so I can not use

```
std::atomic<int> x;
```

How can I make a class thread safe using std::atomic ?

c++ multithreading c++11 atomic

edited Jun 10 '15 at 20:46 — p.i.g. 1,769 1 9 26

asked Jun 10 '15 at 15:47 — mans 4,243 17 72 157

2 Take a look on tutorial baptiste-wicht.com/posts/2012/07/…. – user1929959 Jun 10 '15 at 21:08

4 Answers

You need to make the `x` attribute atomic, and not your whole class, as followed:

```
class A
{
    std::atomic<int> x;

    public:
      A() {
        x=0;
      }
      void Add() {
        x++;
      }
      void Sub() {
        x--;
      }
};
```

The error you get in you original code is completely normal: there is no `std::atomic<A>::Add` method (see here) unless you provide a specialization for `std::atomic<A>` .

Referring your edit: you cannot magically make your `class A` thread safe by using it as template argument of `std::atomic` . To make it thread safe, you can make its attributes atomic (as suggested above and provided the standard library gives a specialization for it), or use mutexes to lock your ressources yourself. See the mutex header. For example:

```
class   A
{
  std::atomic<int>      x;
  std::vector<int>      v;
  std::mutex            mtx;

  void  Add() {
    x++;
  }
  void  Sub() {
    x--;
  }

  /* Example method to protect a vector */
  void  complexMethod() {
    mtx.lock();

    // Do whatever complex operation you need here
    //  - access element
    //  - erase element
    //  - etc ...

    mtx.unlock();
  }

  /*
  ** Another example using std::lock_guard, as suggested in comments
  ** if you don't need to manually manipulate the mutex
  */
  void  complexMethod2() {
    std::lock_guard<std::mutex> guard(mtx);

    // access, erase, add elements ...
  }

};
```

edited Jun 11 '15 at 20:23

answered Jun 10 '15 at 15:58

Unda
908 2 11 23

5 I would sugest `std::lock_guard` instead of manually locking and releasing. – Luke B. Jun 10 '15 at 20:55

@LukeB. Edited, thanks for the suggestion. – Unda Jun 11 '15 at 20:24

@LukeB.Is the idea behind lock_guard to rely on the automatic variables's destructiors to be called when exiting the scope in order to release the lock, just like with RAII ? – Virus721 Sep 22 '16 at 22:31

@Virus721 Yes, that's the idea behind `std::lock_guard` . – Unda Sep 26 '16 at 13:31

Declare the class member x as atomic, then you don't have to declare the object as atomic

```
class A
{
   std::atomic<int> x;
}
```

answered Jun 10 '15 at 15:51

ivanw
310 4 11

what if I want to use a std:;vector<A> and access different member of vector in different threads? – mans Jun 10 '15 at 15:55

1 @mans You can safely access different elements of an `std::vector` from multiple threads as long as you don't access the same element from several threads. That's a guarantee of the standard library. – Morwenn Jun 10 '15 at 15:58

1 @Morwenn But it seems that I can not write. If I read in one thread and write in another thread, the system crashes, Am I wrong? – mans Jun 10 '15 at 16:01

1 @mans If you read the same memory location that you are writing into, this is indeed a data race. – Morwenn Jun 10 '15 at 16:02

2 That's like asking "how can I unscrew this philips head screw with a socket wrench?" a screwdriver and a wrench are both tools to turn things, but one of them is not the right tool for the job. You need a proper mutex or some other more heavy duty synchronization construct in order to protect an entire vector. – Cogwheel Jun 10 '15 at 16:16

The `.` operator can be used on an object to call its class's member function, not some other class's member function (unless you explicitly write the code that way).

```
std::atomic<A> a ;
a.Add(); // Here, a does not know what Add() is (a member function of the type
parameter)
         // It tries to call Add() method of its own class i.e. std::atomic
         // But std::atomic has no method names Add or Sub
```

As the answer by @ivanw mentions, make `std::atomic<int>` a member of your class instead and then use it.

Here is another example:

```
template <typename T> class A
{};

class B { public: void hello() { std::cout << "HELLO!!!"; } };

A<B> a ;
a.hello(); // This statement means that call a's hello member function
           // But the typeof(a) which is A does not have such a function
           // Hence it will be an error.
```

answered Jun 10 '15 at 16:02

a_pradhan
2,450 1 7 17

I think the problem with the answers above is that they don't explain what I think is, at a minimum, an ambiguity in the question, and most likely, a common threaded development fallacy.

You can't make an object "atomic" because the interval between two functions (first "read x" and then later "write x") will cause a race with other uses. If you think you need an "atomic" object, then you need to carefully design the API and member functions to expose now to begin and commit updates to the object.

If all you mean by "atomic" is "the object doesn't corrupt its internal state," then you can achieve this through `std::atomic<>` for single plain-old-data types that have no invariant between them (a doesn't depend on b) but you need a lock of some sort for any dependent rules you need to enforce.

answered Aug 8 '16 at 0:37

Jon Watte
3,231 2 23 40
