

# Lock free programming in C++

## Introduction:

When I was studying advanced topics in C++ last year, I had a chance to learn some basic knowledge about multi-thread programming from the `std::thread` library. The experience by working with C++ thread library. Afterwards, I learned some manual ways to control the synchronizations such as `std::lock`, `std::mutex`, and more. During the winter break in 2022, I invested a significant amount of time reading the book "[C++ Concurrency in Action](#)" written by Anthony Williams, and the inspiring book introduced me to the world of multi-thread programming techniques, which enriched my understanding and proficiency in writing concurrent programming code in C++. Also, the videos from those C++ celebrities such as Herb Sutter and Pitus also provide me with more inspirations while writing this report.

## Purpose:

First of all, lock free programming is probably the most challenging topics I have ever explored in C++ so far. But why C++ coders would like to implement the lock free programming instead of lock implemented concurrency? Clearly, I have not contributed to any projects that require this high level of knowledge. As one of the most popular C++ elders Herb Sutter mentioned, this lock free reduces the overhead in the algorithm and data structures. It further improves the scalability and eliminates bottlenecks. Herb also said `std::atomic` is like a chainsaw, and if you get too comfortable with it and don't respect the tool, bad things will happen. (Sutter, H 2016) My personal understanding, lock free programming is far more than just a chainsaw. It is more like a nuke war head that you could denote it right in front of your face if you don't use it correctly. Moreover, the bugs in parallel programming are often nastier than sequential programming due to the fact - you may or may not replicate it during the debugging and testing. So, my suggestion to all the coders who are reading this report except my professor, don't use it at work unless your team knows the lock is the performance bottle neck of solution performance and understand that the potential consequences.

## What is data race?

First of all, let's look at a code example:

```
#include <iostream>
#include <thread>

int sharedData{ 0 };

void incrementData() {
    for (int i = 0; i < 100000; ++i) {
        sharedData++; // Data race can occur here
    }
}

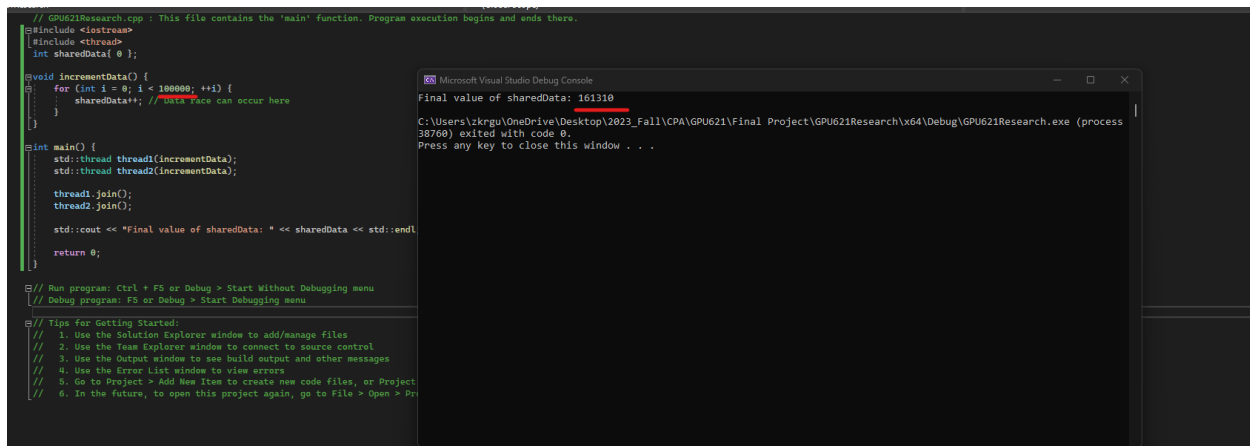
int main() {
    std::thread thread1(incrementData);
    std::thread thread2(incrementData);

    thread1.join();
    thread2.join();

    std::cout << "Final value of sharedData: " << sharedData << std::endl;

    return 0;
}
```

This is just a typical data race scenario. The data race happens when the two instructions from different threads access the same memory location. At least one of these accesses is a write. And there is no synchronization tools to control their accesses. In this case, `sharedData++`. Two threads are trying to read and write at the same time, which potentially caused unpredictable output out the result.



```
// GPU621Research.cpp : This file contains the 'main' function. Program execution begins and ends there.
#include <iostream>
#include <thread>
int sharedData{ 0 };

void incrementData() {
    for (int i = 0; i < 10000; ++i) {
        sharedData++; // Data race can occur here
    }
}

int main() {
    std::thread thread1(incrementData);
    std::thread thread2(incrementData);

    thread1.join();
    thread2.join();

    std::cout << "Final value of sharedData: " << sharedData << std::endl;
    return 0;
}
```

Microsoft Visual Studio Debug Console

Final value of sharedData: 161310

C:\Users\skrgu\OneDrive\Desktop\2023\_Fall\CPA\GPU621\Final Project\GPU621Research\x64\Debug\GPU621Research.exe (process 38768) exited with code 0.  
Press any key to close this window . . .

Run program: Ctrl + F5 or Debug > Start Without Debugging menu  
Debug program: F5 or Debug > Start Debugging menu

Tips for Getting Started:

1. Use the Solution Explorer window to add/manage files
2. Use the Team Explorer window to connect to source control
3. Use the Output window to see build output and other messages
4. Use the Error List window to view errors
5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files
6. In the future, to open this project again, go to File > Open > Recent Projects

If you find it is hard to understand, I have a perfect example here. You are living in a rental house shared with 4 other housemates. And all of you are sharing one washroom, of course, the bad thing could happen if you forget to lock the bathroom door while you are inside. In this example, you and other renters are threads, and the variable “sharedData” memory address can be considered as the “washroom”. The example sounds pretty funny, but data race causes undefined behavior. According to the language standard, once an application contains any undefined behavior, the complete solution became undefined, which means it may do anything you could imagine. Anthony Williams mentioned a data race could lead to undefined behaviors which caused someone’s monitor on fire. (Williams, 2012)

## Solutions to prevent the data Race:

Ultimately, we should minimize the communication between threads if it is possible. Because the following solutions do cause overheads to different extents, which will essentially affect on the algorithm’s performance. But if the multi-thread program does need communication, there are generally 3 ways to synchronize the threads.

### Solution A Lock:

This topic is probably the most common approach for the CPP coders to prevent the data race. However, we will not discuss in this report. Keep in mind that working around with locks just like a game that always keep the lock on to prevent other people comes in while you are inside the bathroom. And while you are about to get out from the washroom, please make sure you

unlock it. Unlocking and locking are equally important in lock involved multithread programming.

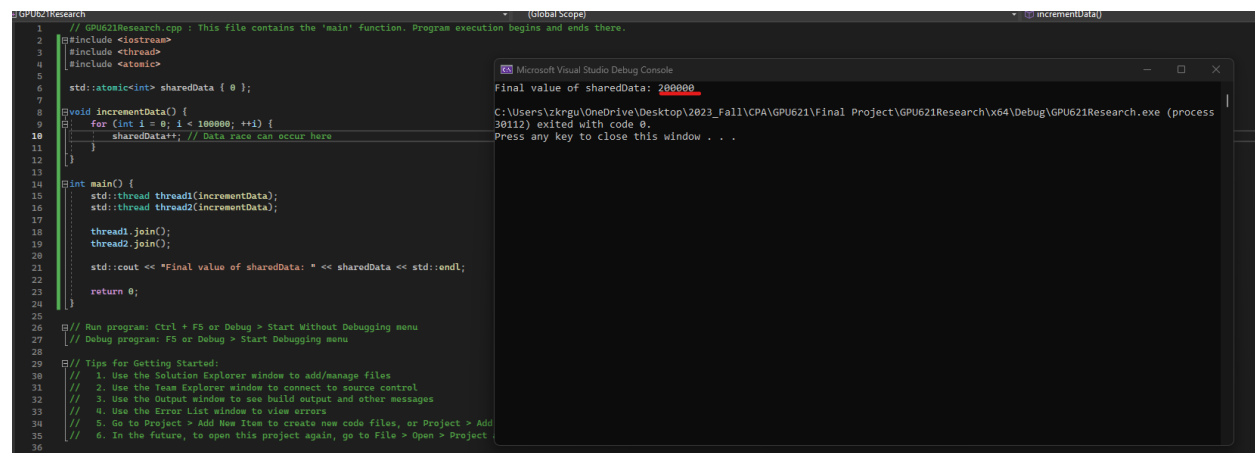
### Solution B Using parallel programming abstractions:

Yes, the perfect example of this term can be TBB, OpenMP which we have learned from this semester. By using the pre-compiler directives or C++ class and API libraries, the synchronization was handled internally, which can be considered as a very convenient way to implement the parallel solutions with less risk. The last way to prevent the data race would be lock free approach.

### Lock-free:

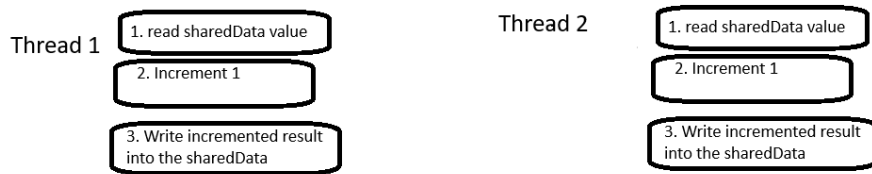
We cannot introduce lock-free programming in C++ without covering `std::atomic`. First of all, atomic operation is an operation that is guaranteed to execute as a single transaction: Other threads will see the state of the system before the operation started or after it finished but cannot see any intermediate state. At the low level, atomic operations are special hardware instructions.

Here is the lock free version to fix the data race solution from the above section.

The image shows a screenshot of a Visual Studio IDE with a C++ project named 'GPU621Research'. The code in 'main.cpp' defines a shared integer variable 'sharedData' of type 'std::atomic<int>' initialized to 0. A function 'incrementData()' is defined, which increments 'sharedData' in a loop from 0 to 100,000. The 'main()' function creates two threads, 'thread1' and 'thread2', both of which call 'incrementData()'. After joining the threads, 'main()' prints the final value of 'sharedData'. The output window shows the final value as 200000, indicating that the data race has been resolved by using atomic operations. The code includes comments about data races and tips for getting started with the IDE.

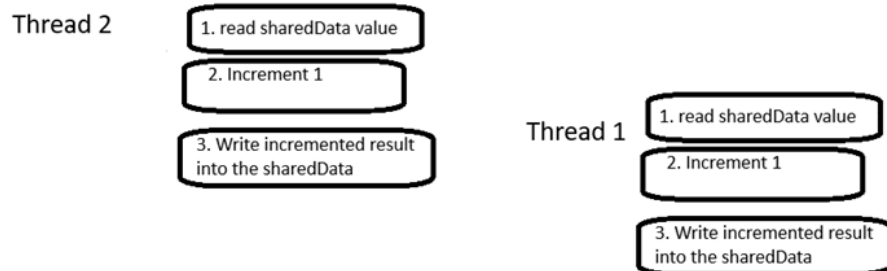
```
1 // GPU621Research.cpp : This file contains the 'main' function. Program execution begins and ends there.
2 #include <iostream>
3 #include <thread>
4 #include <atomic>
5
6 std::atomic<int> sharedData { 0 };
7
8 void incrementData() {
9     for (int i = 0; i < 100000; ++i) {
10         sharedData++; // Data race can occur here
11     }
12 }
13
14 int main() {
15     std::thread thread1(incrementData);
16     std::thread thread2(incrementData);
17
18     thread1.join();
19     thread2.join();
20
21     std::cout << "Final value of sharedData: " << sharedData << std::endl;
22
23     return 0;
24 }
25
26 // Run program: Ctrl + F5 or Debug > Start Without Debugging menu
27 // Debug program: F5 or Debug > Start Debugging menu
28
29 // Tips for Getting Started:
30 // 1. Use the Solution Explorer window to add/manage files
31 // 2. Use the Team Explorer window to connect to source control
32 // 3. Use the Output window to see build output and other messages
33 // 4. Use the Error List window to view errors
34 // 5. Go to Project > Add New Item to create new code files, or Project > Add
35 // 6. In the future, to open this project again, go to File > Open > Project
```

What did I change here? Notice I created atomic object by using the `std::atomic` class template. And yes, data race is fixed. To further explain this, I draw a few charts here.

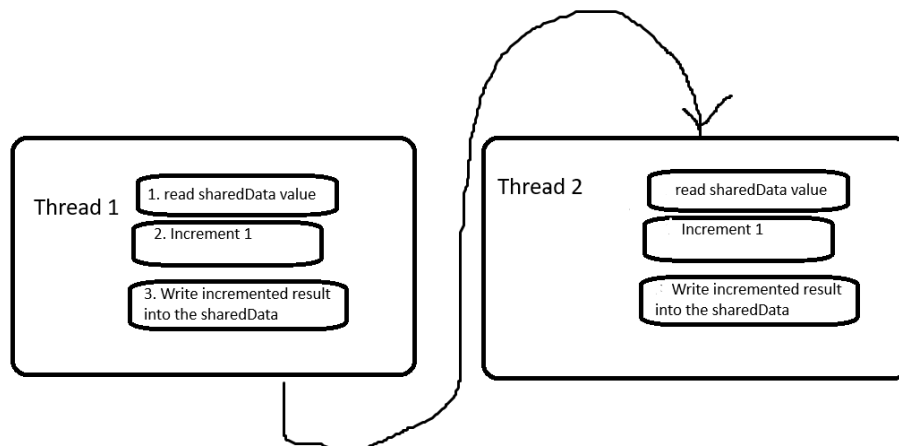


As you can imagine, data race could happen if there is no synchronization between two threads. For example, T1 just read value as 1000, T2 already started writing the value as 1001, T1 does not know the value changed, so it incremented 1 got 1001, and write 1001 again into shared value. Supposedly, the value of sharedData should be 1002, but it is currently 1001. The flow should look like this, please refer to the chart above.

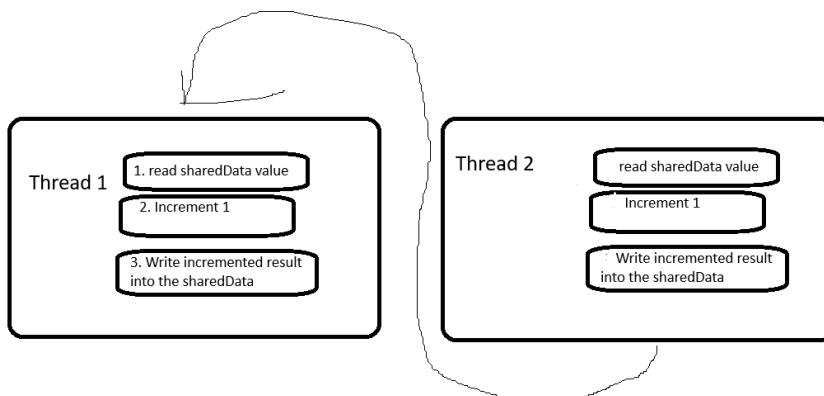
T1 1  
T2 3  
T1 2  
T1 3



Atomic class and its functions guaranteed that the 1,2,3 are done as a whole before the other thread touches the sharedValue. So, there are only two scenarios either T1 is done with 1-3 operations, or T1 is NOT started yet, so T2 can start.



Or



### Std::atomic class template at a glance:

It the template argument take almost everything even including the user defined classes. But it has to be trivially copyable but not copyable. In the human language is:

1. Continuous block of memory: so it can be copied using memcpy().
2. There is no user defined copy, move constructor, assignment operator overload.
3. No virtual functions.

Luckily, C++ has `std::is_trivially_copyable<Your_Class_Name>` to verify if the class is trivially copyable.

Also, the `std::atomic` class template provides member functions, constants, and memory orders.

Member Functions at a Glance		
	Member Function	Functionality
Assignment	<code>operator=</code>	stores a value into an atomic object
	<code>is_lock_free</code>	checks if the atomic object is lock-free
Memory Read and Write	<code>store</code>	atomically replaces the value of the atomic object with a non-atomic argument
	<code>load</code>	atomically obtains the value of the atomic object
	<code>exchange</code>	atomically replaces the value of the atomic object and obtains the value held previously
Read, Modify, Write	<code>compare_exchange_weak</code>	atomically compares the value of the atomic object with non-atomic argument and performs atomic exchange if equal or atomic load if not
	<code>compare_exchange_strong</code>	
Conditional Variable Functionality	<code>wait</code>	Synchronization operators like <code>conditional_variable</code> . (C++20)
	<code>notify_one</code>	
	<code>notify_all</code>	

(Arisaif,2020)

Since there `std::atomic` class objects are not copy assignable, you can not have this type of expression

```
std::atomic<int> sharedData1 = 0;
std::atomic<int> sharedData2 = 0;
sharedData1 = sharedData2;
```

But you have to use member functions to assign values.

```
std::atomic<int> sharedData1 = 0;
std::atomic<int> sharedData2 = 0;
sharedData1.store(sharedData2.load());
```

Because this making sure load and store are atomically executed.

And the following are the specialized member functions of `std::atomic` class:

Specialized Member Functions at a Glance		
Member Function	Type	Functionality
<code>fetch_add</code> <code>fetch_sub</code>	Integral Floating point Pointers: *T	1. Obtains the value held previously 2. Applies the operation on atomic variable and the argument
<code>fetch_and</code> <code>fetch_or</code> <code>fetch_xor</code>	Integral only	
<code>operator++</code> <code>operator++(int)</code> <code>operator--</code> <code>operator--(int)</code>	Integral Pointers: *T	increments or decrements the atomic value by one
<code>operator+=</code> <code>operator-=</code>	Integral Floating point Pointers: *T	adds, subtracts, or performs bitwise AND, OR, XOR with the atomic value
<code>operator&amp;=</code> <code>operator =</code> <code>operator^=</code>	Integral	

(Arisaif,2020)

As you noticed that ++ is overloaded operator of the atomic class, so this is why the above atomic solution works because the ++ operator overload handled it in an atomic way internally. `sharedData++` is definitely NOT equal to

```
sharedData = sharedData + 1
```

Now you must be think, if the C++ committee made something redundant. They have the operator += overloaded, and they also have the `fetch_add` implemented. Why? Isn't this overlapped with each other?

Because `std::atomic` is far more than just making the operation atomic. It could pass memory order objects into those atomic operations. And this is the most hardcore part of the `std::atomic` template class.

```
std::atomic<int> sharedData1 = 0;
std::atomic<int> sharedData2 = 0;
sharedData1.store(sharedData2.load(), );
std::thread thread1(incrementData, &sharedData1, &sharedData2);
```

▲ 4 of 4 ▼ inline void store(const int \_Value, const std::memory\_order Order)



## Memory Orders:

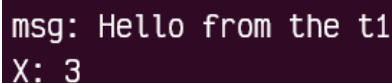
In this section, the program order and `std::memory_order` will be introduced.

### Program order - Sequence before:

C++ official manual contains 20 rules to explain this concept in detail, but it is extremely difficult to understand it because of the precise language. But really, it is like I wrote this sequence in visual studio, and I expect the program output is based on what I wrote.

```
1  x = 3;
2  y += x;
3  msg = "Hello from the t1";
4  flag = 1;
5  std::cout << "msg: " << msg << std::endl;
6  std::cout << "X: " << x << std::endl;
```

Therefore, the output is like here:

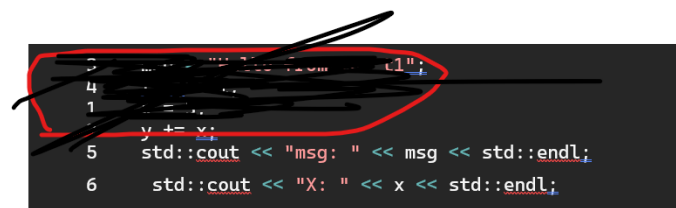


```
msg: Hello from the t1
X: 3
```

But you might think that this is correct nonsense because you know this from the first day you code. However, what you might not know is that the compiler could potentially change the execution sequence of what you wrote to optimize. And every time the sequence could be different.

```
3  msg = "Hello from the t1";
4  flag = 1;
1  x = 3;
2  y += x;
5  std::cout << "msg: " << msg << std::endl;
6  std::cout << "X: " << x << std::endl;
```

Would this change the output? No because you only care about the output of line 5 and 6. So the sequence change would not affect the result. And the sequence of the reordering is private in the single thread environment. So, essentially it is like this, as long as I give you the correct output matched what you coded, you should not care what I am doing inside this censored area.



```
3  msg = "Hello from the t1";
4  flag = 1;
1  x = 3;
2  y += x;
5  std::cout << "msg: " << msg << std::endl;
6  std::cout << "X: " << x << std::endl;
```

Now we have two threads, privacy is gone because all the threads can see what is inside the other thread. Take the below code as an example, the t2 can see t1 's reordered version, which could cause t2 not working as expected.

Inside the t1:

```
1  x = 3;
2  y += x;
3  msg = "Hello from the t1";
4  flag = 1;
5  std::cout << "msg: " << msg << std::endl;
6  std::cout << "X: " << x << std::endl;
```

Inside the t2:

```
a  txt = "Hello I am T2";
b  while (flag != 1);
c  txt = msg;
d  std::cout << "text: " << txt << std::endl;
```

If you still cannot see the problem here is a detailed version of reordering caused problem.

```
4  flag = 1;
1  x = 3;
2  y += x;
3  msg = "Hello from the t1";
5  std::cout << "msg: " << msg << std::endl;
6  std::cout << "X: " << x << std::endl;
```

```
a  txt = "Hello I am T2";
b  while (flag != 1);
c  txt = msg;
d  std::cout << "text: " << txt << std::endl;
```

if 4 happens before 3 , it does not affect Thread 1, but thread 2 's output would be text:

Because 3 is not executed yet, 4 happens before 3. And after 4 executed, c is executed because b unlocked. As a result, the msg is assigned to the txt. But 3 is not executed yet. So msg is empty. So, the output from thread 2 would be "text:"

(Don't try to replicate this because visual C++ compiler can easily optimize this issue. You won't even catch the bug)

How to resolve this potential bug? Happens-before.

[Program order Happens-Before:](#)

Sequence before from above only resolve the sequence from an individual thread point of view.

But happens before here could potentially resolved the dilemma in the multithreaded environment because it synchronizes the order between the threads.

Inside the t1:

```
1  x = 3;
2  y += x;
3  msg = "Hello from the t1";
4  flag = 1;
5  std::cout << "msg: " << msg << std::endl;
6  std::cout << "X: " << x << std::endl;
```

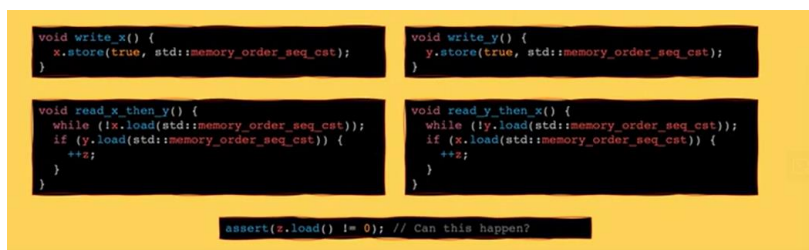
Inside the t2:

```
a  txt = "Hello I am T2";
b  while (flag != 1);
c  txt = msg;
d  std::cout << "text: " << txt << std::endl;
```

In this case, you tell the compiler 3 happens before 4, 4 happens before b. So, 3 happens before b. So, the msg will contain values before the step c.

But it is easy to say, how to implement this in C++? Well, there is no way so far in the std::atomic way because the std::atomic require the type of argument to be trivially copyable. std::string does have copy constructor, copy assignment, and move implemented. So I cannot use std::atomic<std::string> msg().

Apart from the synchronization, in the next code example, you will be able to see the implementation by using the methods like load and store from the std::atomic template class to specify the memory order to create a order of operations and make all threads agree on the order of operations. This is code example from C++ website. You have 4 threads in the example, we assume x, y were false, and z was 0. Would the assert to be triggered?

The image shows a screenshot of four C++ code blocks on a yellow background. The first block defines 'write\_x()' which sets 'x' to true using 'store(true, std::memory\_order\_seq\_cst)'. The second block defines 'write\_y()' which sets 'y' to true using 'store(true, std::memory\_order\_seq\_cst)'. The third block defines 'read\_x\_then\_y()' which loops until 'x' is true, then increments 'z' and loops until 'y' is true. The fourth block defines 'read\_y\_then\_x()' which loops until 'y' is true, then increments 'z' and loops until 'x' is true. At the bottom, an assertion 'assert(z.load() != 0); // Can this happen?' is shown.

```
void write_x() {
    x.store(true, std::memory_order_seq_cst);
}

void write_y() {
    y.store(true, std::memory_order_seq_cst);
}

void read_x_then_y() {
    while (!x.load(std::memory_order_seq_cst));
    if (y.load(std::memory_order_seq_cst)) {
        ++z;
    }
}

void read_y_then_x() {
    while (!y.load(std::memory_order_seq_cst));
    if (x.load(std::memory_order_seq_cst)) {
        ++z;
    }
}

assert(z.load() != 0); // Can this happen?
```

(Arisaif,2020)

The fact is no because here all threads used memory\_order\_seq\_cst and they agree either write\_x happens before write\_y or write\_y happens before write\_x. So, the assertion will never happen. In the other word, there is a unified order agreed by those 4 code blocks by using the std::memory\_order\_seq\_cst.

As a matter of fact, there are totally 6 memory orders from the `std::atomic`.

`std::memory_order_seq_cst` is just one of them, and it is the commonly used memory order, which is default memory order while using the atomic methods. Due to the scope of this report, the rest of memory orders will not be covered in this report.

## Summary:

The research report initiates by discussing the data race condition in C++ multi-threading programming. It introduces two primary solutions to mitigate this issue: the use of locks and programming abstractions in order to prevent race conditions. Subsequently, the report explores the third solution—lock-free programming. This section provides a comprehensive exploration of various aspects, including atomic operations, the utilization of `std::atomic`, and the associated syntax. As the report progresses towards its conclusion, the focus is placed on the memory order in C++. Although not entirely discussed due to the scope concerns, the report briefly covers certain parts of the memory order.

## Discussion:

Why use you want to learn this? To show off front of the other coders. This is just like Chi master can control anything including needles and leaves to kill their enemies. You are not just coding; you are controlling the machine from a very low level to get the results you want. C++ empowers the coders to do this type of lower level controlling in order to fully unleash the power of the machine. Nowadays, especially front-end web development, there are overwhelmingly number of frameworks available. Many coders decide to focus on how to use some frameworks to produce the similar solutions again and again. Life is short. We should all code something that makes yourself thrilled every time you see it.

On the other hand, should we always use lock free then? No, lock free does not always mean fast. (Pikus,2015). I can tell that how complicated `std::atomic` is to be used in the production environment. Also, I want to remind my classmates here: 99.99% of chance lock solution and high abstraction solutions are not the performance bottle neck in your current projects, at least not for now. Therefore, don't even think about implementing lock-free unless you indeed understand the concepts throughout.

# Reference List

Arisaif. (2020, August 25). *C++ Multi Threading Part 3: Atomic Variables and Memory Models*

[Video]. Youtube. <https://www.youtube.com/watch?v=IE6EpKT7cJ4&t=2405s>

Pikus, F. (2015, October 9). *CppCon 2015: Fedor Pikus PART 1 "Live Lock-Free or Deadlock*

*(Practical Lock-free Programming)* [Video]. Youtube.

<https://www.youtube.com/watch?v=IVBvHbJsg5Y>

Sutter, H. (2012, November 6). *C++ and Beyond 2012: Herb Sutter - atomic Weapons 1 of 2*

[Video]. Youtube. <https://www.youtube.com/watch?v=A8eCGOqgvH4>

Williams, A. (2012). *C++ Concurrency in Action: Practical Multithreading*.

<http://ci.nii.ac.jp/ncid/BB11831342>