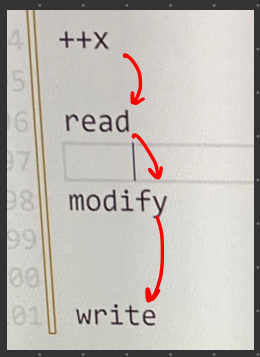
* Atomik Türler / Atomik İşlemler:

→ Atomik işlem, bölünemez işlem. Araya başka threadler giremez!



→ Bu işlem farklı threadlere verilse / araya başka işlem olsa = DATA RACE yaşanabilir!

→ Aynı şekilde ++x + ++y
(++x: atomik, ++y: atomik)
İşlem bütünün atomik olma garantisi yok!

→ Neden Atomik İşlemler kullanırız?
- non divisable

* Memory Model:

- Özellikle multithread çalışacak programların, belleğeki okuma yazma işlemleriyle ve paylaşılan değişkenlerle ilişkisini tanımlar.
- Modern Cpp ile eklendi.
- Memory model ile, birden fazla threadin olduğu uygulamalarda, threadlerin davranışı kestirilebilir hale geldi.

memory model nedir?

- Bilgisayar programının belleğe erişiminde izin verilebilen semantik yapıyı tanımlar.
- bellekten bir okuma yapıldığında hangi değer /değerler okunabilir? (hangi değerlerin okunması beklenebilir)
- hangi durumlarda tanımsız davranış oluşabilir?
- memory model, *multithread* progranmlar için kritik olan bir bileşendir.
- *thread*'ler bellek ve paylaşılan veril(ler) ile nasıl etkileşime giriyor? Bellek programa ve programda kullanılan *thread*'lere nasıl görünüyor?
- multi-thread programların yürütülmesine ilşkin kuralların belirlenebilmesi için programlama dilinin bir memory model oluşturulması gerekiyor.

thread'ler, programın çalışma zamanında paylaşılan verileri farklı değerlerde görebilir. bir *memory model* oluşturmadan derleyici optimizasyonları ve donanım tarafında yapılan optimizasyonlar sorunlar oluşturabilir. *thread*'lerin paylaşılan değişkenlerde yapılan değişiklikleri anında gözlemleyebilmeleri verim *(efficiency)* açısından yüksek bir maliyet oluşturur.

kaynak kodumuzdaki işlemler

- derleyici tarafından
- CPU tarafından;
- bellek (caching) tarafından yeniden sıralanabilir

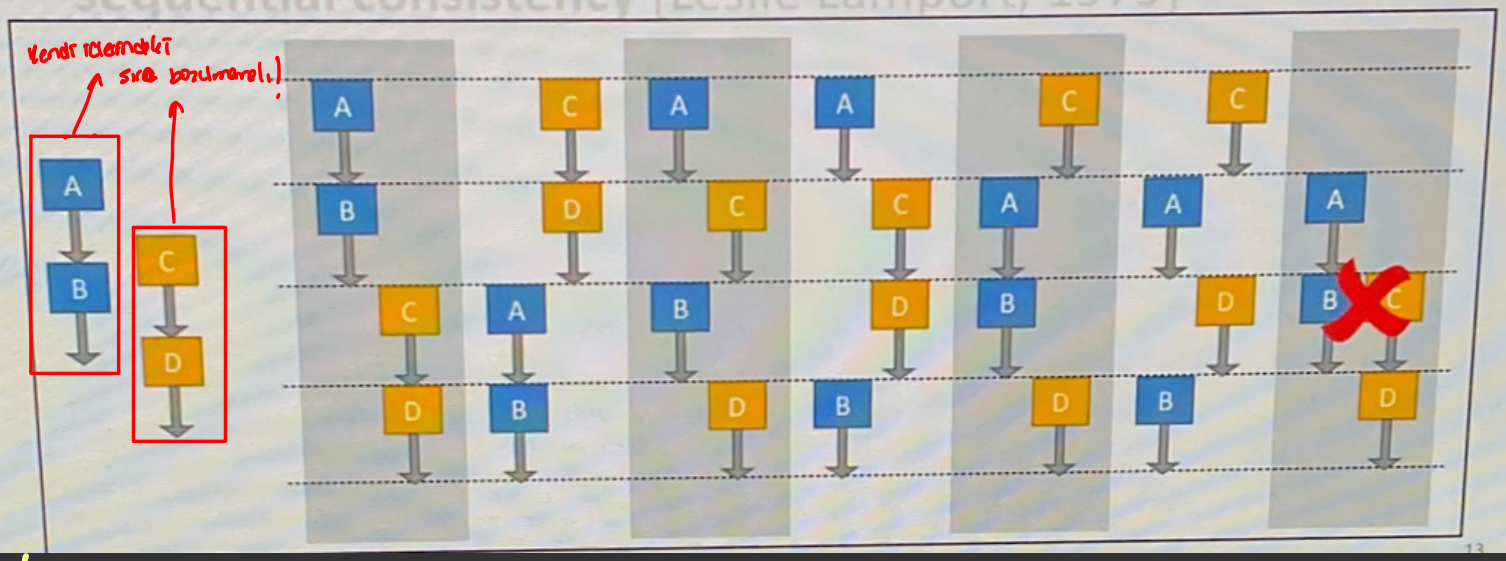
memory model hangi durumlarda yeniden sıralamaya *(reordering)* izin verilip verilmediğini belirler. Birden fazla *thread* hangi durumlarda paylaşılan değişkenlere erişebilir? Bir *thread* tarafından bir değişkene yapılan atama eş zamanlı çalışan *thread*'ler tarafından ne zaman görünür olacak? Programcılar *multi thread* programların nasıl çalışacağını (belirli garantiler altında) kestirebilmeliler. memory model buna yardımcı oluyor.

- C++11 öncesinde C++ dilinden formal bir bir memory model yoktu. C++ soyut makinesi *(abstract machine) single-thread* olarak tasarlanmıştı. java memory model 1995 yılında oluşturuldu. Yaygın kullanılan programlama dilleri içinde bu açıdan java bir ilk. *pthreads* kütüphanesi ilk olarak 1995 yılında, bir *memory modele* dayanmadan geliştirildi. Posix memory model oluşturmuyor. Tanımlar kesin *(precise)* değil. Yani, "program doğru mu" sorusunun formal bir cevabı yok. İlk olarak C++ dilinde java'nın memory modelinin kullanılması düşünüldü, ancak bunun uygun olmadığı görüldü (fazla kısıtlayıcı). Java memory modeli belirli veri türlerinin atomik olmasını gerektiriyor. Java memory model C++ için çok pahalı. Bir thread library oluşturabilemek için derleyici üstünde bazı kısıtlamaların olması gerekiyor. Bunun için de bir memory model'e ihtiyaç duyuluyor.

Bir programın çalışma zamanında nasıl davranacağını kestirebilmemiz için şunları bilebilmemiz gerekir:

- programdaki (farklı *thread*'ler tarafından gerçekleştirilebilecek) işlemlerin *(operasyonların)* hangi sırayla yürütüleceği *(ordering)*
- programdaki bir işlemin/işlemlerin sonuçlarının (başka *thread*'ler tarafından yürütülebilecek) diğer bir işlem/işlemler yapılmadan görülür olup olmadığı *(visibility)*

sequential consistency [Leslie Lamport, 1979]



↳ Yani, farklı threadlerin yaptığı farklı işlemler, tek bir threadde sıralı çalışıyormuş gibi olmalı ⟶ Sequential Consistency

fakat sırası aynı olmak zorunda değil

happens-before ilişkisi

A ve *B* iki operasyon olsun. Bu operasyonlar aynı *thread*'de gerçekleştirilen operasyonlar olabildiği gibi farklı *thread*'lerde gerçekleştirilen operasyonlar da olabilir. Eğer *A* operasyonunun etkileri *B* operasyonunu yürütecek *thread*'de *B* operasyonu yürütüldüğünde görülür durumda ise

A happens before B

garantisi söz konusudur.

***happens-before* ilişkisi "zamansal olarak daha önce gerçekleşme" ilişkisine göre çok daha güçlü bir durumdur.**

Eğer *A* operasyonu *B* operasyonuna göre zamansal açıdan daha önce gerçekleştiriliyor ise bu *A*'nın etkilerinin *B*'yi yürütecek *thread*'de *B* yürütülmeden önce görülür olma garantisi değildir. *caching*, *store buffer* vs gibi mekanizmalar operasyonun etkilerinin diğer *thread*'lerde görülmesini geciktirebilir.

sequenced before ilişkisi

sequenced-before ilişkisi aynı *thread*'deki işlemlere ilişkindir.

- *Sequenced-before* ilişkisi geçişkendir *(transitive)* A *sequenced before* B doğru ise B *sequenced before* C doğru ise A *sequenced before* C doğrudur.

```
x = 1;          // A
y = 2;          // B
z = x + 1;      // C
```

Aşağıdaki kodu ele alalım:

```
y = a * x + b; // (y = ((a * x) + b);
```

Bu ifadede 3 işlem var: çarpma toplama ve atama. Burada
çarpma işlemi *sequenced before* toplama işlemi
toplama işlemi *sequenced before* atama işlemi
dolayısıyla
atama işlemi *sequenced before* toplama işlemi

Eğer bir *thread* için
A *sequenced before* B doğru ise
A *happens before* B doğrudur.

Bu şu anlama geliyor:
sequenced-before ilişkisi aynı zamanda *thread* içindeki *(intra-thread)* happens before ilişkisine karşılık geliyor.

- *happens-before* ilişkisi zamana bağlı değildir, görünürlüğe *(visibility)* bağlıdır.
- *A happens before B* ise, bu A'nın B'de önce yapılması anlamına gelmez (böyle bir garanti yoktur).
- A B'den önce yapılmış ise *A happens-before B* olmak değildir.
- happens-before ilişkisi *acquire-release* semantiği ile gerçekleştirilebilir. (Daha sonra ele alacağım)

```
#include <iostream>

int is_ready{ 0 };
int value{0};

void producer()
{
    value = 42;                          // (1)
    is_ready = 1;                        // (2)
}

void consumer()
{
    if (is_ready)                        // (3) Burada okunan değerin 1 olduğunu düşünelim
        std::cout << value;              // (4)
}
```

Buradaki fonksiyonlar iki ayrı *thread* tarafından çalıştırılıyor olsun. Değişkenlere yapılan atamalar *(stores)* ve değişkenlerden yapılan okumalar *(loads)* atomik olsun. Programın çalışma zamanında *consumer thread*'inin (3) noktasına geldiğini ve *is_ready* değişkeninin okunan değerinin 1 olduğunu düşünelim. Bu değer *producer thread*'inde (2) noktasında *is_ready* değişkenine atanan değer.
(2) (3)'ten önce olmuş olmalı. Ama bu (2) ve (3) arasında *happens-before* ilişkisi olduğu anlamına gelmez.
(2) ve (3) arasında *happens-before* ilişkisi olmadığı gibi (1) ile (4) arasında da *happens-before* ilişkisi yoktur. Bu yüzden (1) (4) arasındaki bellek işlemleri farklı şekilde sıralanabilir (reordering). derleyici tarafından oluşturulan *instruction*'lar işlemci ya da bellek tarafından farklı şekilde sıralanabilir *(instruction reordering / memory reordering)*. *consumer thread*'i (4) noktasına geldiğinde ekrana 0 değeri yazılabilir.

- Inter-thread happens-before ilişkisi farklı thread'ler arasındaki happens-before ilişkisidir.
- A ve B farklı threadler'deki operasyonlar olsun. Eğer A ve B arasında *inter-thread-happens_before* ilişkisi varsa A ve B arasında *happens-before* ilişkisi vardır. Yani A'daki operasyonların sonucu B'de görülebilir olmak zorundadır.
- Inter-thread happens-before ilşkisi geçişkendir *(transitive)*
- Inter-thread-happens-before ilişkisinin oluşması için (dil tarafından tanımlanan) bir senkronizasyonun söz konusu olması gerekir.
- A ve B arasında happens-before ilişkisi olsun. Bu durumda
  - A ve B aynı thread'de olabilir ve aralarında sequenced-before ilişkisi vardır.
  - A ve B aynı farklı thread'lerde olabilir ve aralarında intra-thread-happens-before ilişkisi vardır.

Examples of Synchronizes-With Relationships

- ***Thread creation.*** The completion of the constructor for a `thread` object T synchronizes with the start of the invocation of the thread function for T. [C++17 §33.3.2.2/6]
- ***Thread join.*** The completion of the execution of a thread function for a `thread` object T synchronizes with (the return of) a `join` operation on T. [C++17 §33.3.2.5/4]
- ***Mutex unlock/lock.*** All prior `unlock` operations on a mutex M synchronize with (the return of) a `lock` operation on M. [C++17 §33.4.3.2/11]
- ***Atomic.*** A suitably tagged atomic write operation W on a variable `x` synchronizes with a suitably tagged atomic read operation on `x` that reads the value stored by W (where the meaning of "suitably tagged" will be discussed later).

→ Michael D Adams !!
↳ slaytları bul ve oku!

Atomic kütüphanesi:

```cpp
#include <atomic>
#include <iostream>

int main()
{
    std::cout.setf(std::ios::boolalpha);
    std::atomic<int> x;
    //std::atomic<long long int> x;

    std::cout << x.is_lock_free() << "\n";
}
```

→ true

→ true

x.is_lock_free(): implementasyonunda lock / mutex kullanılmıyor. ⇒ Doğrudan instruction'ları atomic

Atomic flag: - bu her yerde lock free olmak zorunda
- her tür her zaman lock-free değilmiş?

```cpp
#include <iostream>

int main()
{
    using namespace std;

    cout << boolalpha;
    // atomic_flag flag_x{ false }; //gecersiz
    // atomic_flag flag_y{ true };  //gecersiz
    atomic_flag flag_z; //C++ 17'de belirsiz deger C++20'de false değeri
    cout << "flag_z = " << flag_z.test() << "\n"; //C++20
    atomic_flag flag = ATOMIC_FLAG_INIT;  //gecerli
    cout << "flag = " << flag.test() << "\n"; //C++20
    auto b = flag.test_and_set();
    cout << "b = " << b << "\n";
    cout << "flag = " << flag.test() << "\n";
    flag.clear();
    cout << "flag = " << flag.test() << "\n";
    b = flag.test_and_set();
    cout << "b = " << b << "\n";
    cout << "flag = " << flag.test() << "\n";
}
```

```cpp
#include <iostream>

class SpinLockMutex {
public:
    SpinLockMutex()
    {
        m_f.clear();
    }
    void lock()
    {
        while (m_f.test_and_set())
            ; //null statement
    }

    void unlock()
    {
        m_f.clear();
    }
private:
    std::atomic_flag m_f;
};

SpinLockMutex mtx;
unsigned long long counter{};

void func()
{
    for (int i{ 0 }; i < 100'000; ++i) {
        mtx.lock();
        ++counter;
        mtx.unlock();
    }
}
```

```cpp
int main()
{
    std::vector<std::thread> tvec;

    for (int i = 0; i < 10; ++i) {
        tvec.emplace_back(func);
    }

    for (auto& th : tvec) {
        th.join();
    }

    std::cout << "counter = " << counter << "\n";
}
```

* atomic<bool>:

```cpp
#include <atomic>
#include <iostream>

int main()
{
    using namespace std;

    cout << boolalpha;
    atomic<bool> flag_1;
    atomic<bool> flag_2; //indetermined value before before C++20. false value since C++20
    cout << flag_1 << '\n';
    cout << flag_2 << '\n';
    ///atomic<bool> flag_3{flag_2}; //geçersiz
    //flag_1 = flag_2; //geçersiz

    flag_1 = true;
    flag_2 = false;
    flag_1.store(false);
    flag_2.store(true);

    cout << "flag_1 = " << flag_1 << '\n';  // operator T
    cout << "flag_2 = " << flag_2 << '\n';  // operator T

    auto b = flag_1.exchange(true); önceki değer false ise, b=false flg=true olur.
    cout << "b = " << b << '\n';
    cout << "flag_1 = " << flag_1 << '\n';  // operator T
    cout << "flag_1.load() = " << flag_1.load() << "\n";
    cout << "flag_2.load() = " << flag_2.load() << "\n";
}
```

*atomic<int>:

```cpp
// int expected
// // desired
// expected = 4
// bool result = a.compare_exchange_strong(expected, 50);

int main()
{
    using namespace std;

    cout << boolalpha;
    atomic<int> a;

    cout << "a = " << a << "\n";
    cout << "a.load() = " << a.load() << "\n";
    a.store(10);
    cout << "a = " << a << "\n";
    int expected = 20;
    cout << "expected = " << expected << "\n";
    bool result = a.compare_exchange_strong(expected, 50);
    // a has not the expected value and will not be set
    cout << "a = " << a << "\n";
    //result will be false
```

a'nın değeri expected ise, a'nın yeni değeri 50 ile değişir. result=true! (changed)

eğer " " " değil ise, expected = changed olur, result=false!

↳ böyle bir yapının temel nedeni, biz a'nın değeri üzerinden işlem yapıken, eğer değeri değiştirildiyse, yeni değer üzerinden işlem yapmak

```cpp
atomic<int> a = 10;

int temp = a.load();
//a = 200
while (!a.compare_exchange_weak(temp, temp * 50))
    ;
```

her türlü modifiye edilmiş değer!

```cpp
using namespace std;

int main()
{
    atomic<int> a = 10;

    ++a;
    a++;
    a = 5;
    //a.load()
    a += 5;
    a -= 10;
    a &= 4;
    a ^= 4;
    a |= 4;

    auto result = a.exchange(450)
```

→ Atomic(int) class template features

→ compare_exchange_weak:

For `compare_exchange_weak()`, the store might not be successful even if the original value was equal to the expected value, in which case the value of the variable is unchanged and the return value of `compare_exchange_weak()` is `false`. This is most likely to happen on machines that lack a single compare-and-exchange instruction, if the processor can't guarantee that the operation has been done atomically—possibly because the thread performing the operation was switched out in the middle of the necessary sequence of instructions and another thread scheduled in its place by the operating system where there are more threads than processors. This is called a *spurious failure*, because the reason for the failure is a function of timing rather than the values of the variables.

```cpp
#include <atomic>
#include <thread>
#include <iostream>

class AtomicCounter {
public:
    AtomicCounter() : m_c(0) {}
    AtomicCounter(int val) : m_c{ val } {}
    int operator++() { return ++m_c; }
    int operator++(int) { return m_c++; }
    int operator--() { return --m_c; }
    int operator--(int) { return m_c--; }
    int get() const { return m_c.load(); }
    operator int()const { return m_c.load(); }
private:
    std::atomic<int> m_c;
};

AtomicCounter cnt;

void foo()
{
    for (int i = 0; i < 1'000'000; ++i) {
        ++cnt;
    }
```

→ Atomic Pointer:

Table 5.3 The operations available on atomic types

Operation	atomic_flag	atomic <bool>	atomic <T*>	atomic <integral-type>	atomic <other-type>
test_and_set	Y				
clear	Y				
is_lock_free		Y	Y	Y	Y
load		Y	Y	Y	Y
store		Y	Y	Y	Y
exchange		Y	Y	Y	Y
compare_exchange_weak, compare_exchange_strong		Y	Y	Y	Y
fetch_add, +=			Y	Y	
fetch_sub, -=			Y	Y	
fetch_or, |=				Y	
fetch_and, &=				Y	
fetch_xor, ^=				Y	
++, --			Y	Y	

```cpp
using namespace std;

int main()
{
    int a[] = { 1, 3, 5, 7, 9 ,11 };
    atomic<int *> ax{a};

    ++ax;

    std::cout << *ax << "\n";

}
```