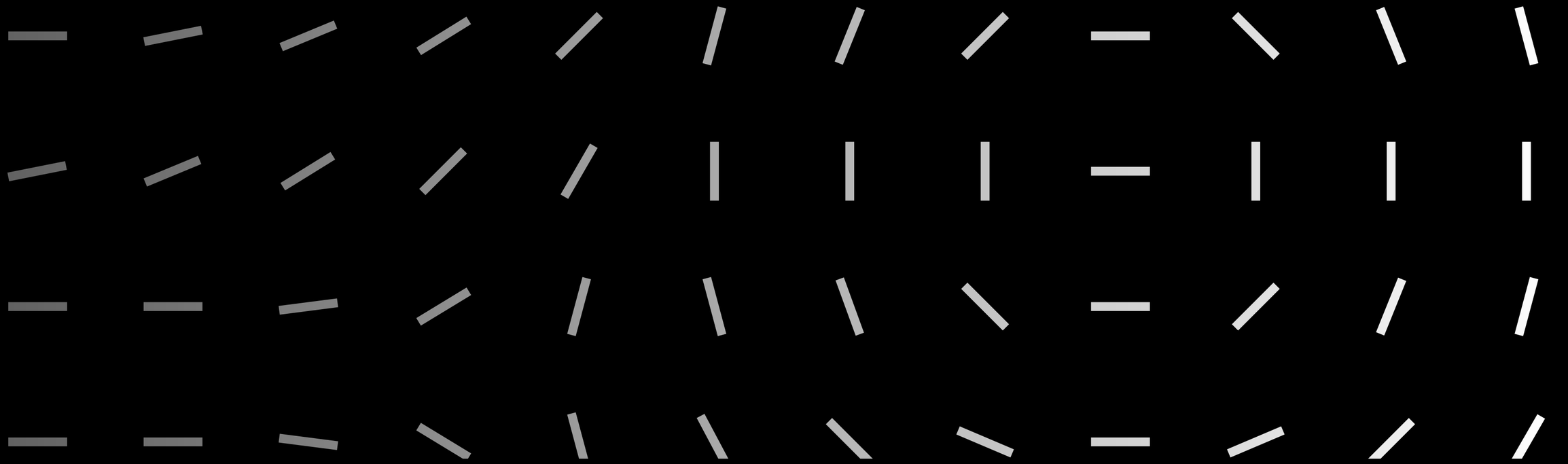


Chapter V

The C++ memory model and operations on atomic types

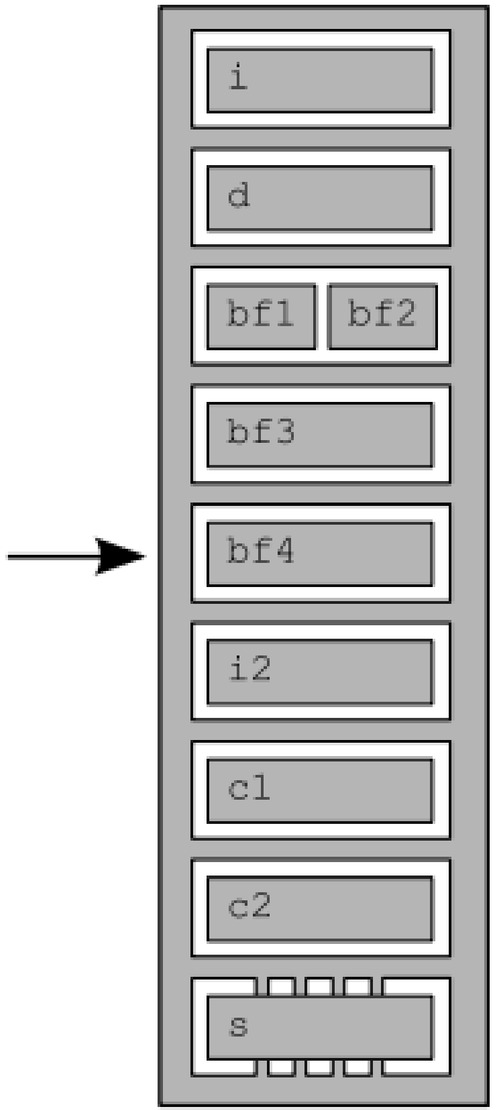
By Olexandra Olashyn PMI-33
and Nazarii Protskiv PMI-31

You will learn

- The details of the C++11 memory model
- The atomic types provided by the C++
- Standard Library
- The operations that are available on
- those types
- How those operations can be used to provide synchronization between threads

Objects and memory locations

```
struct my_data
{
    int i;
    double d;
    unsigned bf1:10;
    int bf2:25;
    int bf3:0;
    int bf4:9;
    int i2;
    char c1,c2;
    std::string s;
};
```



Object

Memory Location

Modification orders

Atomic operations and types in C++

The standard atomic types

<atomic>

std::atomic_flag

is_lock_free()

test_and_set()

clear()

load() store()

compare_exchange_weak()

compare_exchange_strong()

Atomic type	Corresponding specialization
`atomic_bool`	`std::atomic<bool>`
`atomic_char`	`std::atomic<char>`
`atomic_schar`	`std::atomic<signed char>`
`atomic_uchar`	`std::atomic<unsigned char>`
`atomic_int`	`std::atomic<int>`
`atomic_uint`	`std::atomic<unsigned>`
`atomic_short`	`std::atomic<short>`
`atomic_ushort`	`std::atomic<unsigned short>`

atomic_long	std::atomic<long>
atomic_ulong	std::atomic<unsigned long>
atomic_llong	std::atomic<long long>
atomic_ullong	std::atomic<unsigned long long>
atomic_char16_t	std::atomic<char16_t>
atomic_char32_t	std::atomic<char32_t>
atomic_wchar_t	std::atomic<wchar_t>

Atomic `typedef`	Corresponding Standard Library `typedef`
`atomic_int_least8_t`	`int_least8_t`
`atomic_uint_least8_t`	`uint_least8_t`
`atomic_int_least16_t`	`int_least16_t`
`atomic_uint_least16_t`	`uint_least16_t`
`atomic_int_least32_t`	`int_least32_t`
`atomic_uint_least32_t`	`uint_least32_t`
`atomic_int_least64_t`	`int_least64_t`
`atomic_uint_least64_t`	`uint_least64_t`
`atomic_int_fast8_t`	`int_fast8_t`
`atomic_uint_fast8_t`	`uint_fast8_t`
`atomic_int_fast16_t`	`int_fast16_t`
`atomic_uint_fast16_t`	`uint_fast16_t`

atomic_int_fast32_t	int_fast32_t
atomic_uint_fast32_t	uint_fast32_t
atomic_int_fast64_t	int_fast64_t
atomic_uint_fast64_t	uint_fast64_t
atomic_intptr_t	intptr_t
atomic_uintptr_t	uintptr_t
atomic_size_t	size_t
atomic_ptrdiff_t	ptrdiff_t
atomic_intmax_t	intmax_t
atomic_uintmax_t	uintmax_t

load()
store()
exchange()
compare_exchange_weak()
compare_exchange_strong()

- збереження
- завантаження
- читання-модифікація-запис

memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_seq_cst
memory_order_release
memory_order_acq_rel

```cpp
#include <iostream>
#include <atomic>
#include <thread>
using namespace std;
atomic<int> sharedValue(0);

void threadFunction() {
    int expected = 0;
    int newValue = 42;
    bool exchanged = sharedValue.compare_exchange_weak(expected, newValue);

    if (exchanged)
        cout << "Value exchanged successfully by thread" << endl;
    else
        cout << "Value was not exchanged by thread" << endl;
}

int main() {
    std::thread t(threadFunction);
    t.join();
    cout << "Final value: " << sharedValue.load() << endl;
    return 0;
}
```

```cpp
#include <iostream>
#include <atomic>
#include <thread>
using namespace std;
atomic<int> sharedValue(0);

void threadFunction() {
    int expected = 0;
    int newValue = 42;
    bool exchanged = sharedValue.compare_exchange_strong(expected, newValue);

    if (exchanged)
        cout << "Value exchanged successfully by thread." << endl;
    else
        cout << "Value was not exchanged by thread." << endl;
}

int main() {
    thread t(threadFunction);
    t.join();
    cout << "Final value: " << sharedValue.load() << endl;
}
```

Atomic flag

std::atomic_flag

clear()
test_and_set()

x.is_lock_free()
! x.is_lock_free()

```
std::atomic_flag f=ATOMIC_FLAG_INIT;
```

<atomic>

```
class spinlock_mutex
{
    std::atomic_flag flag;
public:
    spinlock_mutex():
        flag(ATOMIC_FLAG_INIT)
    {}
    void lock()
    {
        while(flag.test_and_set(std::memory_order_acquire));
    }

    void unlock()
    {
         flag.clear(std::memory_order_release);
    }
};
```

Atomic bool

std::atomic<bool>

```
std::atomic<bool> b(true);
b=false;
```

clear() -> store()

test_and_set() -> exchange()

std::atomic<bool>

```
std::atomic<bool> b;
bool x=b.load(std::memory_order_acquire);
b.store(true);
x=b.exchange(false,std::memory_order_acq_rel);
```

std::atomic<T*>

std::atomic<T*>

is_lock_free()
load()
store()
exchange()
compare_exchange_weak()
compare_exchange_strong()

fetch_add()
fetch_sub()

```cpp
#include <atomic>

std::atomic<int> atomicVariable(0);

int previousValue = atomicVariable.fetch_add(5);
```

Atomic integral types

fetch_add()	+=
fetch_sub()	-=
fetch_and()	&=
fetch_or()	|=
fetch_xor()	^=

The std::atomic<> primary class template

std::atomic<User-defined type>

memcpy(); memcmp();

compare_exchange_strong

Free functions for atomic operations

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	✓				
clear	✓				
is_lock_free		✓	✓	✓	✓
load		✓	✓	✓	✓
store		✓	✓	✓	✓
exchange		✓	✓	✓	✓
compare_exchange_weak, compare_exchange_strong		✓	✓	✓	✓
fetch_add, +=			✓	✓	
fetch_sub, -=			✓	✓	
fetch_or, |=				✓	
fetch_and, &=				✓	
fetch_xor, ^=				✓	
++, --			✓	✓	

load() ⟶ std::atomic_load()

a.load(std::memory_order_acquire)

⟶

std::atomic_load_explicit(&a,
std::memory_order_acquire).

std::atomic_store(&atomic_var,new_value)

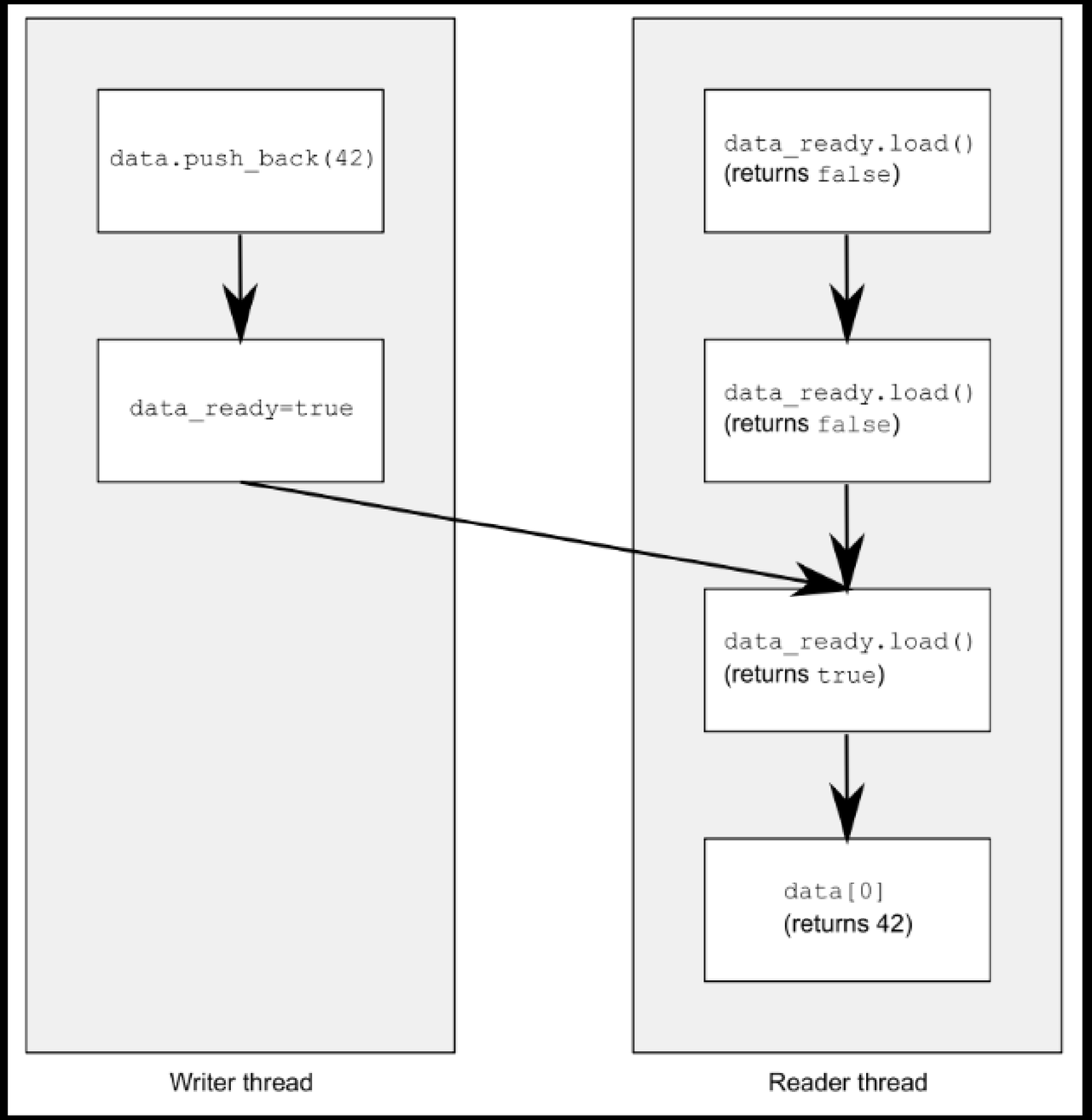
std::atomic_ store_explicit
(&atomic_var,new_value,std::memory_order_release)

Synchronizing operations and enforcing ordering

```
#include <vector>
#include <atomic>
#include <iostream>

std::vector<int> data;
std::atomic<bool> data_ready(false);

void reader_thread()
{
    while(!data_ready.load())    <-1
    {
        std::this_thread::sleep(std::milliseconds(1));
    }
    std::cout<<"The answer="<<data[0]<<"\n";    <-2
}
void writer_thread()
{
    data.push_back(42);          <-3
    data_ready=true;             <-4
}
```


data.push_back(42)
data_ready=true
data_ready.load()
(returns false)
data_ready.load()
(returns false)
data_ready.load()
(returns true)
data[0]
(returns 42)
Writer thread
Reader thread

The synchronizes-with relationship

The happens-before relationship

Listing 5.3 Order of evaluation of arguments to a function call is unspecified

```
#include <iostream>

void foo(int a,int b)
{
    std::cout<<a<<","<<b<<std::endl;
}

int get_num()
{
    static int i=0;
    return ++i;
}

int main()
{
    foo(get_num(),get_num());
}
```

Calls to get_num() are unordered

memory_order_relaxed
memory_order_acquire
memory_order_release
memory_order_acq_rel
memory_order_seq_cst

SEQUENTIALLY CONSISTENT ORDERING

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

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;
}
```

```
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);
}
```

RELAXED ORDERING

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);
    y.store(true,std::memory_order_relaxed);
}

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

```
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);
}
```

ACQUIRE-RELEASE ORDERING

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

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

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

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

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

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);    <-- 3
}
```

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);
    y.store(true,std::memory_order_release);
}

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

```
int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);
}
```

Release sequences and synchronizes-with

Listing 5.11 Reading values from a queue with atomic operations

```
#include <atomic>
#include <thread>

std::vector<int> queue_data;
std::atomic<int> count;

void populate_queue()
{
    unsigned const number_of_items=20;
    queue_data.clear();
    for(unsigned i=0;i<number_of_items;++i)
    {
        queue_data.push_back(i);
    }

    count.store(number_of_items,std::memory_order_release);
}

void consume_queue_items()
{
    while(true)
    {
        int item_index;
        if((item_index=count.fetch_sub(1,std::memory_order_acquire))<=0)
        {
            wait_for_more_items();
            continue;
```

1 The initial store

2 An RMW operation

3 Wait for more items

```
            }
            process(queue_data[item_index-1]);
        }
    }

int main()
{
    std::thread a(populate_queue);
    std::thread b(consume_queue_items);
    std::thread c(consume_queue_items);
    a.join();
    b.join();
    c.join();
}
```

4 **Reading queue_data is safe**

Fences

Listing 5.12 Relaxed operations can be ordered with fences

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);                    ←1
    std::atomic_thread_fence(std::memory_order_release);        ←2
    y.store(true,std::memory_order_relaxed);                    ←3
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));                  ←4
    std::atomic_thread_fence(std::memory_order_acquire);        ←5
    if(x.load(std::memory_order_relaxed))                       ←6
        ++z;
}

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);    ←7
}
```

```
#include <atomic>
#include <thread>
#include <assert.h>

bool x=false;
std::atomic<bool> y;
std::atomic<int> z;

void write_x_then_y()
{
    x=true;
    std::atomic_thread_fence(std::memory_order_release);
    y.store(true,std::memory_order_relaxed);
}

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

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);
}
```

x is now a plain nonatomic variable

1 Store to x before the fence

2 Store to y after the fence

3 Wait until you see the write from #2

4 This will read the value written by #1

5 This assert won't fire

SUMMARY