Basics of the Memory Model

From the concurrency perspective, there are two main aspects of the memory model:

- What is a memory location?
- What happens if two threads access the same memory location?

Let me answer both questions.

What is a memory location?

A memory location is according to cppreference.com[12]

- an object of scalar type (arithmetic type, pointer type, enumeration type, or std::nullptr_t),
- or the largest contiguous sequence of bit fields of non-zero length.

Here is an example to a memory location:

```
struct S {
char a; // memory location #1
int b : 5; // memory location #2
int c : 11, // memory location #2 (continued)
: 0,
d : 8; // memory location #3
int e; // memory location #4
double f; // memory location #5
std::string g; // several memory locations
};
```

First, the object obj consists of a seven sub-objects and the two bit fields b, and c share the same
memory location.

Here are a few import observations:

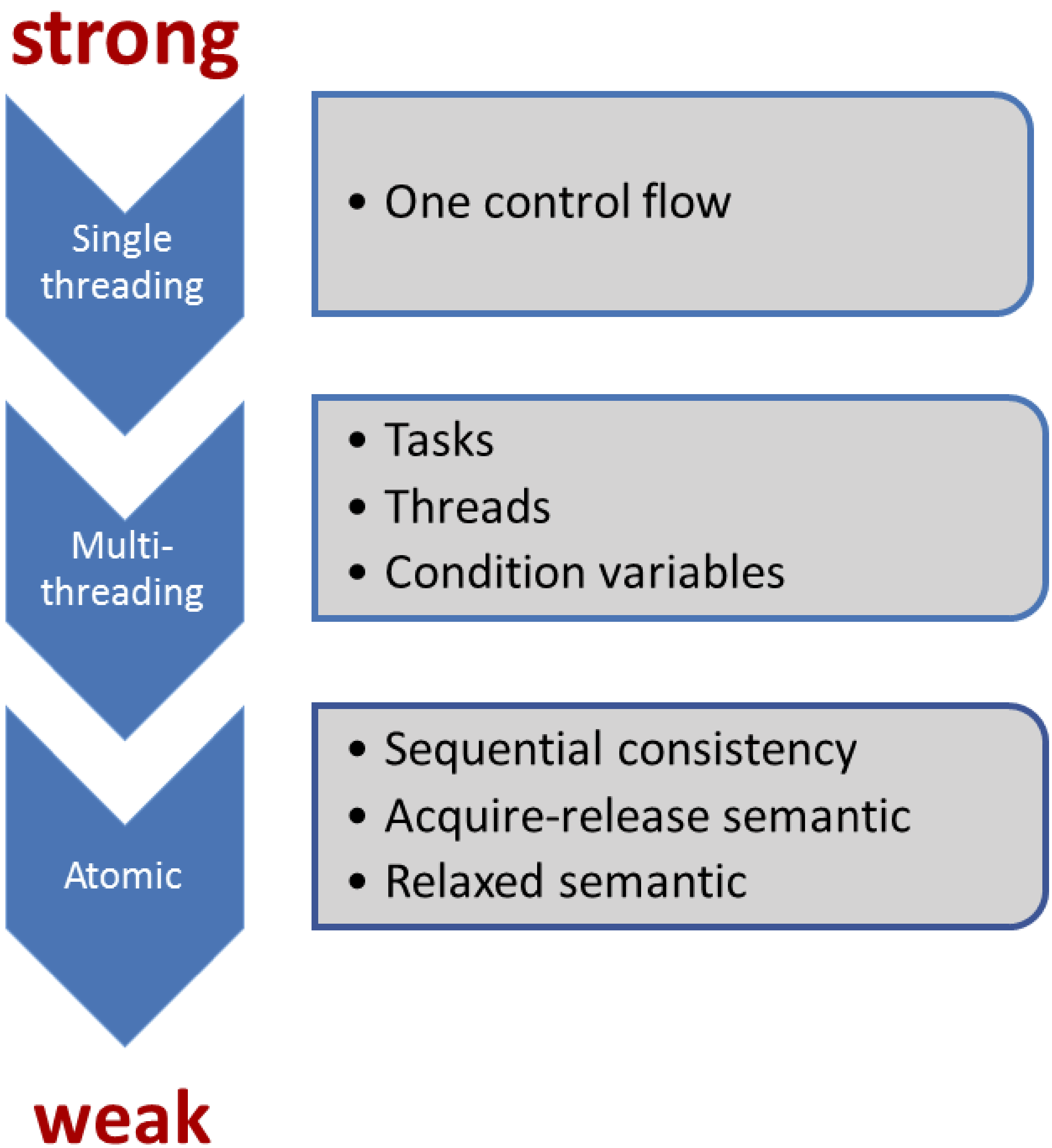
- Each variable is an object.
- Scalar types occupy one memory location.
- Adjacent bit fields (b and c) have the same memory location.
- Variables occupies at least one memory location.

What happens if two threads access the same memory location?

If two threads access the same memory location - adjacent bit fields can share the same memory
location - and at least one thread wants to modify it, your program has a data races unless

1. the memory location is modified by an atomic operation.
2. one access happens-before the other.

Roughly speaking there are three contract levels in C++11.

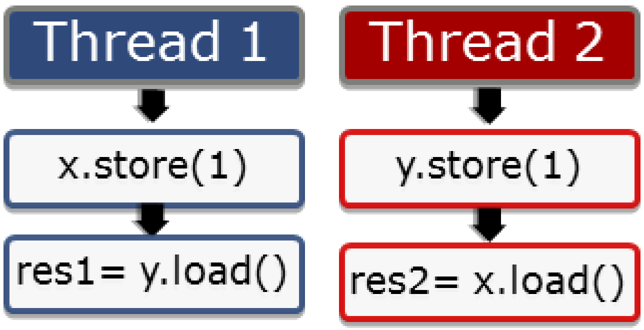


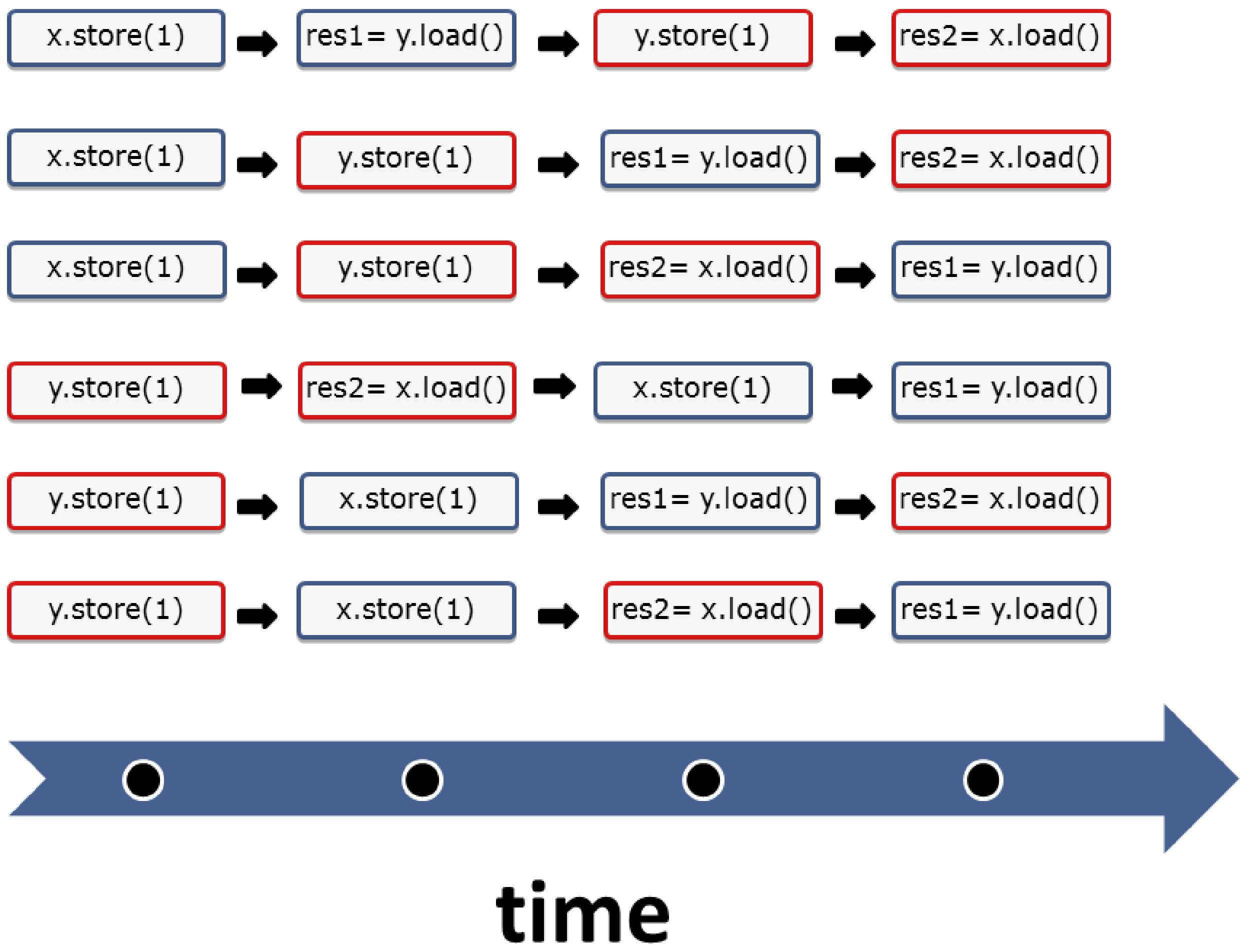
The C++ memory model has to deal with the following points:

- Atomic operations: operations that can be performed without interruption.
- Partial ordering of operations: sequences of operations that must not be

reordered.

• Visible effects of operations: guarantees when operations on shared variables are visible to
other threads.





The two threads can overlap their instruction operations in many different ways, leading to race condition.

The Atomic Flag:

Atomic Flag: two states - Being set (true) or cleared (false)

The two member functions, test and set, and clear, are both atomic.

test_and_set:

Sets the atomic_flag (to true) and returns whether it was already set immediately before the call.

clear:

Clears the atomic_flag (to false).

Atomic flag is used to design spinlock (a CPU intense lock).

A spinlock with std::atomic_flag

```cpp
// spinLock.cpp

#include <atomic>
#include <thread>

class Spinlock{
std::atomic_flag flag = ATOMIC_FLAG_INIT;
public:

void lock(){
while( flag.test_and_set() );
}

void unlock(){
flag.clear();
}

};

Spinlock spin;

void workOnResource(){
spin.lock();
// shared resource
spin.unlock();
}
```

```
int main(){

std::thread t(workOnResource);
std::thread t2(workOnResource);

t.join();
t2.join();

}
```

Spinlock vs mutex

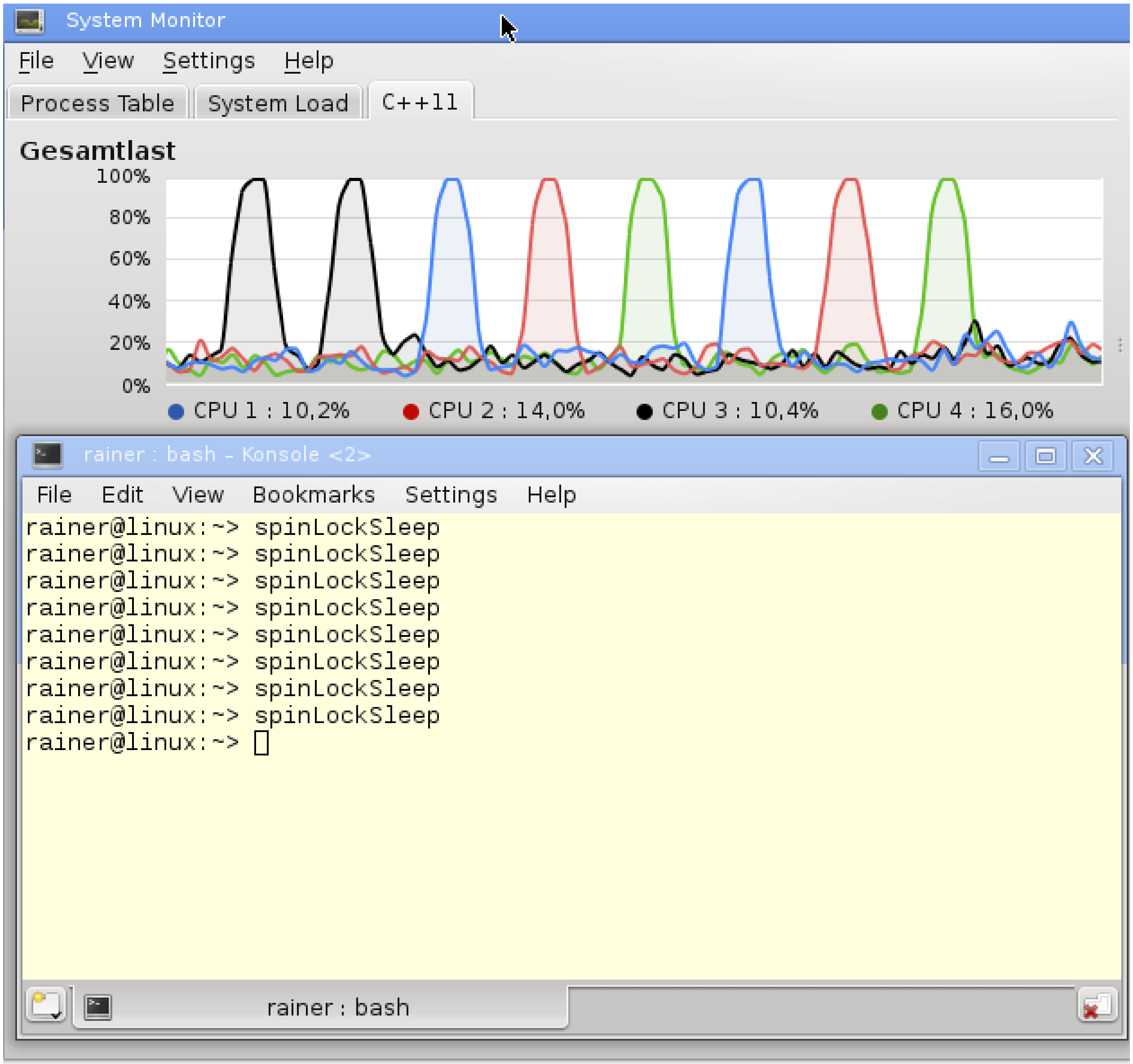
Waiting with a spinlock

```
// spinLockSleep.cpp

#include <atomic>
#include <thread>

class Spinlock{
std::atomic_flag flag = ATOMIC_FLAG_INIT;
public:

void lock(){
while( flag.test_and_set() );
}

void unlock(){
flag.clear();
}

};
```



```

Spinlock spin;

void workOnResource(){
```

```
spin.lock();
std::this_thread::sleep_for(std::chrono::milliseconds(2000));
spin.unlock();
}


int main(){

std::thread t(workOnResource);
std::thread t2(workOnResource);

t.join();
t2.join();

}
```



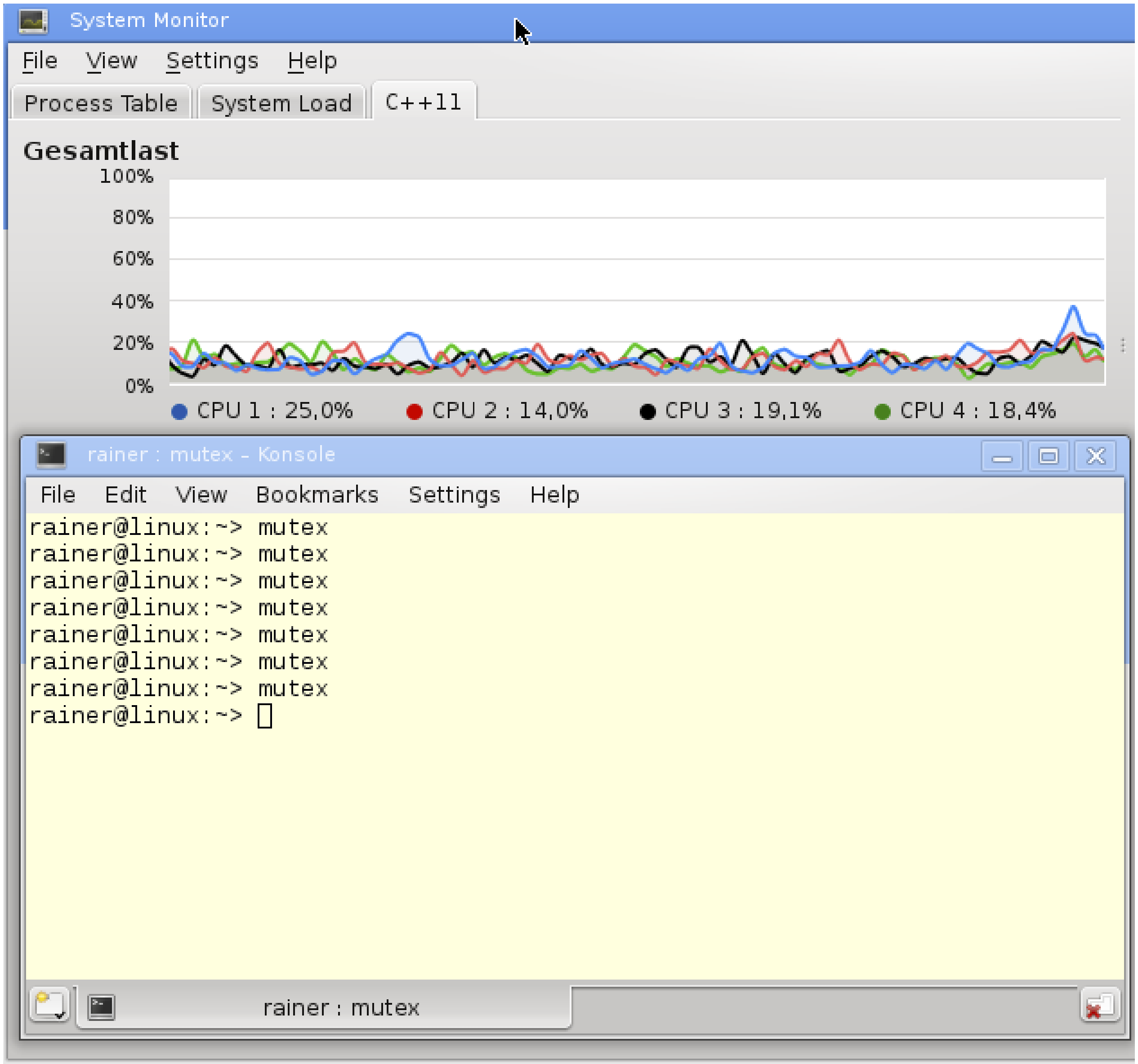
Waiting with a mutex

```
// mutex.cpp

#include <mutex>
#include <thread>

std::mutex mut;

void workOnResource(){
```

```
mut.lock();
std::this_thread::sleep_for(std::chrono::milliseconds(5000));
mut.unlock();
}

int main(){

std::thread t(workOnResource);
std::thread t2(workOnResource);
```



```

t.join();
t2.join();

}
```



Another example of spinlock

```
// atomic_flag as a spinning lock
#include <iostream>          // std::cout
#include <atomic>            // std::atomic_flag
#include <thread>            // std::thread
```

```
#include <vector>               // std::vector
#include <sstream>              // std::stringstream

std::atomic_flag lock_stream = ATOMIC_FLAG_INIT;
//ATOMIC_FLAG_INIT: This macro is defined in such a way that it can be
used to initialize an object of type atomic_flag to the clear state.

std::stringstream stream;

void append_number(int x) {
	while (lock_stream.test_and_set()) {}
	//Testand set flag
		// Sets the atomic_flagand returns whether it was already set
immediately before the call.

	std::cout << "thread #" << x << '¥n';
	lock_stream.clear();
}

int main()
{
	std::vector<std::thread> threads;
	for (int i = 1; i <= 10; ++i)
threads.push_back(std::thread(append_number, i));
	for (auto& th : threads) th.join();

	std::cout << stream.str();
	return 0;
}
```

Memory order, shown in the following table, is another major issues that will be discussed.

public member function

std::atomic_flag::test_and_set

<atomic>

```
bool test_and_set (memory_order sync = memory_order_seq_cst) volatile noexcept;
bool test_and_set (memory_order sync = memory_order_seq_cst) noexcept;
```

Test and set flag

Sets the atomic_flag and returns whether it was already set immediately before the call.

The entire operation is atomic (an *atomic read-modify-write* operation): the value is not affected by other threads between the instant its value is read (to be returned) and the moment it is modified by this function.

Parameters

sync

Synchronization mode for the operation.
This can be any of the possible values of the enum type memory_order:

value	memory order	description
memory_order_relaxed	Relaxed	No synchronization of side effects.
memory_order_consume	Consume	Synchronizes the visible side effects on values *carrying dependencies* from the last *release* or *sequentially consistent* operation.
memory_order_acquire	Acquire	Synchronizes all visible side effects from the last *release* or *sequentially consistent* operation.
memory_order_release	Release	Synchronizes side effects with the next *consume* or *acquire* operation.
memory_order_acq_rel	Acquire/Release	Reads as an *acquire* operation and writes as a *release* operation (as described above).
memory_order_seq_cst	Sequentially consistent	Synchronizes all visible side effects with the other *sequentially consistent* operations, following a single total order.

Return value

true if the flag was set before the call.
false otherwise.

More example on atomic vs mutex

std::atomic<bool>

Usage of a condition variable

```cpp
// conditionVariable.cpp

#include <condition_variable>
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> mySharedWork;
std::mutex mutex_;
std::condition_variable condVar;

bool dataReady{false};

void waitingForWork(){
std::cout << "Waiting " << std::endl;
std::unique_lock<std::mutex> lck(mutex_);
condVar.wait(lck, []{ return dataReady; });
mySharedWork[1] = 2;
std::cout << "Work done " << std::endl;
}

void setDataReady(){
mySharedWork = {1, 0, 3};
{
std::lock_guard<std::mutex> lck(mutex_);
dataReady = true;
}
```

```
std::cout << "Data prepared" << std::endl;
condVar.notify_one();
}
int main(){
std::cout << std::endl;
std::thread t1(waitingForWork);
std::thread t2(setDataReady);
t1.join();
t2.join();
for (auto v: mySharedWork){
std::cout << v << " ";
}
std::cout << "\n\n";
}
```

Implementation of a condition variable with std::atomic<bool>

```cpp
// atomicCondition.cpp

#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> mySharedWork;
std::atomic<bool> dataReady(false);

void waitingForWork(){
std::cout << "Waiting " << std::endl;
while (!dataReady.load()){
std::this_thread::sleep_for(std::chrono::milliseconds(5));
}
mySharedWork[1] = 2;
std::cout << "Work done " << std::endl;
}

void setDataReady(){
mySharedWork = {1, 0, 3};
dataReady = true;
std::cout << "Data prepared" << std::endl;
}

int main(){

std::cout << std::endl;

```

```
std::thread t1(waitingForWork);
std::thread t2(setDataReady);

t1.join();
t2.join();

for (auto v: mySharedWork){
std::cout << v << " ";
}


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

}
```

User Defined Atomics std::atomic<user-defined type>

std::atomic<T*>

```
int intArray[5];
std::atomic<int*> p(intArray);
p++;
assert(p.load() == &intArray[1]);
p+=1;
assert(p.load() == &intArray[2]);
--p;
assert(p.load() == &intArray[1]);
```

std::atomic<integral type>

• character types: char, char16_t, char32_t, and wchar_t

• standard signed integer types: signed char, short, int, long, and long long

• standard unsigned integer types: unsigned char, unsigned short, unsigned int, unsigned long, and unsigned long long

• additional integer types, defined in the header <cstdint>[20] :

– int8_t, int16_t, int32_t, and int64_t (signed integer with exactly 8, 16, 32, and 64 bits)

– uint8_t, uint16_t, uint32_t, and uint64_t (unsigned integer with exactly 8, 16, 32, and 64 bits)

– int_fast8_t, int_fast16_t, int_fast32_t, and int_fast64_t (fastest signed integer with at least 8, 16, 32, and 64 bits)

– uint_fast8_t, uint_fast16_t, uint_fast32_t, and uint_fast64_t (fastest unsigned integer with at least 8, 16, 32, and 64 bits)

– int_least8_t, int_least16_t, int_least32_t, and int_least64_t (smallest signed integer with at least 8, 16, 32, and 64 bits)

– uint_least8_t, uint_least16_t, uint_least32_t, and uint_least64_t (smallest unsigned integer with at least 8, 16, 32, and 64 bits)

– intmax_t, and uintmax_t (maximum signed and unsigned integer)

– intptr_t, and uintptr_t (signed and unsigned integer for holding a pointer)

std::atomic<integral type> supports the composite assignment operators +=, -=, &=, |= and ^= and their fetch pedants: fetch_add, fetch_sub, fetch_and, fetch_or and fetch_xor.

There is a small difference in the composite assignment and the fetch version. The composite assignment operators return the new value; the fetch variations returns the old value. Additionally, the pre- and postincrement and pre- and post-decrement (++x, x++, --x, and x--) are available.
A more in-depth look provides more insight: there is no atomic multiplication, atomic division,
nor atomic shift operation available. This is not a significant limitation, because these operations
are seldom needed and can easily be implemented. Here is an example of an atomic fetch_mult
function.

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

std::atomic<long long> data;
void do_work()
{
	data.fetch_add(1, std::memory_order_relaxed);
}

int main()
{
	std::thread th1(do_work);
	std::thread th2(do_work);
	std::thread th3(do_work);
	std::thread th4(do_work);
	std::thread th5(do_work);

	th1.join();
	th2.join();
```

```
    th3.join();
    th4.join();
    th5.join();

    std::cout << "Result:" << data << '¥n';
}
```

```
Some common operations on std::atomic<T> x;
//read and write
T y = x.load();    //y = x; OK
x.store(y); //x = y;    OK

//atomic exchange
T z = x.exchange(y); //z = x; x=y;    One atomic operation.
//x = x+5; //two atomic operations


//compare-and-swap
bool Bool = x.compare_exchange_strong(y,z);
//if x==y, make x=z and return true;
//Else, set y = x and return false;


//Example: atomic increment with CAS
std::atomic<int> x {10};
int x0 = x;
while (!x.compare_exchange_strong(x0, x0+1)) {}

//x becomes x+1, whichis an atomic operation
//atomic increment used to be only valid for int;
//Now, increment operations can be performed atomically even if it is not of type int.
//The concept can be used for increment more general type (such as doubles),
//multiply integers, and many more.

while (!x.compare_exchange_strong(x0, x0*2)){}
//x becomes x*2.    An atomic operation.
```

An atomic multiplication with compare_exchange_strong

```cpp
// fetch_mult.cpp

#include <atomic>
#include <iostream>

template <typename T>
T fetch_mult(std::atomic<T>& shared, T mult){
T oldValue = shared.load();
while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
return oldValue;
}

int main(){
std::atomic<int> myInt{5};
std::cout << myInt << std::endl;
fetch_mult(myInt,5);
std::cout << myInt << std::endl;
}
```

compare_exchange_strong and compare_exchange_weak

compare_exchange_strong has the syntax: bool compare_exchange_strong(T& expected, T& desired). Because this operation compares and exchanges its values in one atomic operation, it is often called compare and swap (CAS). This kind of operation is available in many programming languages and is the foundation of non-blocking algorithms. Of course, the behaviour may vary a little.

atomicValue.compare_exchange_strong(expected, desired) has the following behaviour.

- If the atomic comparison of **atomicValue** with **expected** returns **true**, **atomicValue** is set in the same atomic operation to **desired**.
- If the comparison returns **false**, **expected** is set to **atomicValue**.

compare_exchange_weak.

The weak version can fail spuriously. That means, although *atomicValue == expected holds, atomicValue was not set to desired and the function call returns false, so you have to check the condition in a loop: while (!atomicValue.compare_exchange_weak(expected, desired)). The weak form exists because some processor doesn't support an atomic compare-exchange instruction. When called in a loop the weak form should be preferred. On some platforms, the weak form can run faster.

Type aliases for std::atomic<bool> and std::atomic<integral type>

Type alias Definition

std::atomic_bool std::atomic<bool>

std::atomic_char std::atomic<char>

std::atomic_schar std::atomic<signed char>

std::atomic_uchar std::atomic<unsigned char>

std::atomic_short std::atomic<short>

std::atomic_ushort std::atomic<unsigned short>

std::atomic_int std::atomic<int>

std::atomic_uint std::atomic<unsigned int>

std::atomic_long std::atomic<long>

std::atomic_ulong std::atomic<unsigned long>
std::atomic_llong std::atomic<long long>
std::atomic_ullong std::atomic<unsigned long long>
std::atomic_char16_t std::atomic<char16_t>
std::atomic_char32_t std::atomic<char32_t>
std::atomic_wchar_t std::atomic<wchar_t>
std::atomic_int8_t std::atomic<std::int8_t>
std::atomic_uint8_t std::atomic<std::uint8_t>
std::atomic_int16_t std::atomic<std::int16_t>
std::atomic_uint16_t std::atomic<std::uint16_t>
std::atomic_int32_t std::atomic<std::int32_t>
std::atomic_uint32_t std::atomic<std::uint32_t>
std::atomic_int64_t std::atomic<std::int64_t>
std::atomic_uint64_t std::atomic<std::uint64_t>
std::atomic_int_least8_t std::atomic<std::int_least8_t>
std::atomic_uint_least8_t std::atomic<std::uint_least8_t>‘
std::atomic_int_least16_t std::atomic<std::int_least16_t>
std::atomic_uint_least16_t std::atomic<std::uint_least16_t>
std::atomic_int_least32_t std::atomic<std::int_least32_t>
std::atomic_uint_least32_t std::atomic<std::uint_least32_t>
std::atomic_int_least64_t std::atomic<std::int_least64_t>
std::atomic_uint_least64_t std::atomic<std::uint_least64_t>
std::atomic_int_fast8_t std::atomic<std::int_fast8_t>
std::atomic_uint_fast8_t std::atomic<std::uint_fast8_t>
std::atomic_int_fast16_t std::atomic<std::int_fast16_t>
std::atomic_uint_fast16_t std::atomic<std::uint_fast16_t>
std::atomic_int_fast32_t std::atomic<std::int_fast32_t>
std::atomic_uint_fast32_t std::atomic<std::uint_fast32_t>
std::atomic_int_fast64_t std::atomic<std::int_fast64_t>
std::atomic_uint_fast64_t std::atomic<std::uint_fast64_t>
std::atomic_intptr_t std::atomic<std::intptr_t>

std::atomic_uintptr_t std::atomic<std::uintptr_t>
std::atomic_size_t std::atomic<std::size_t>
std::atomic_ptrdiff_t std::atomic<std::ptrdiff_t>
std::atomic_intmax_t std::atomic<std::intmax_t>
std::atomic_uintmax_t std::atomic<std::uintmax_t>

All atomic operations

Method Description

test_and_set Atomically set the flag to true and returns the previous value.
clear Atomically sets the flag to false.
is_lock_free Checks if the atomic is lock-free.
load Atomically return the value of the atomic.
store Atomically replaces the value of the atomic with a non-atomic.
exchange Atomically replaces the value with the new value. Returns the old value.
compare_exchange_strong Atomically compares and eventually exchanges the value. Details are here.
compare_exchange_weak
fetch_add, += Atomically adds(subtracts) the value.
fetch_sub, -=
fetch_or, |= Atomically performs bitwise (OR, AND, and XOR) operation with the value.
fetch_and, &=
fetch_xor, ^=
++, -- Increments or decrements (pre- and post-increment) the atomic.

All atomic operations depeding on the atomic type

Method atomic_flag atomic<bool> atomic<user> atomic<T*>

atomic<integral>

test_and_set yes

clear yes

is_lock_free yes yes yes yes

load yes yes yes yes

store yes yes yes yes

exchange yes yes yes yes

compare_exchange_strong yes yes yes yes

compare_exchange_weak

fetch_add, += yes yes

fetch_sub, -=

fetch_or, |= yes

fetch_and, &=

fetch_xor, ^=

++, -- yes yes