ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

ATOMIC

- Implement indivisible operations.
- Set constraints on execution order of memory operations in the thread.
- Synchronize memory in two or more threads.

ATOMIC

```
static std::atomic<T>::is_always_lock_free
std::atomic<T>::is_lock_free
```

```
struct A { int a[100]; };
struct B { int x, y; };

void main()
{
   static_assert(std::atomic<long[2]>::is_always_lock_free);
   static_assert(!std::atomic<long[3]>::is_always_lock_free);

   std::cout << std::atomic<A>{}.is_lock_free() << '\n'
             << std::atomic<B>{}.is_lock_free() << '\n';
}
```

ATOMIC

```
std::atomic<T>::store
std::atomic<T>::load
```

```
std::atomic<bool> atomic_variable;

//Thread 1
data_queue.push(prepare_data());
atomic_variable.store(true); //or atomic_variable = true;

//Thread 2
while (!atomic_variable.load()); //or while(!atomic_variable);
process_data(data_queue.pop());
```

ATOMIC

std::atomic<T>::exchange

```cpp
const std::size_t ThreadNumber = 5;
const int Sum = 5;
std::atomic<int> atom{0};
std::atomic<int> counter{0};

auto lambda = [&](const int id){
    for (int next = 0; next < Sum;){
        const int current = atom.exchange(next);
        counter++;

        std::osyncstream(std::cout)
           << '#' << id
           << " (" << std::this_thread::get_id()
           << ") wrote " << next
           << " replacing the old value "
           << current << '\n';
        next = std::max(current, next) + 1;
    }
};

std::vector<std::thread> v;
for (size_t i = 0; i < ThreadNumber; ++i){
    v.emplace_back(lambda, i);
}
```

Possible output:

```
#1 (140552371918592) wrote 0 replacing the old value 0
#2 (140552363525888) wrote 0 replacing the old value 0
#1 (140552371918592) wrote 1 replacing the old value 0
#1 (140552371918592) wrote 2 replacing the old value 1
#2 (140552363525888) wrote 1 replacing the old value 1
#1 (140552371918592) wrote 3 replacing the old value 2
#1 (140552371918592) wrote 4 replacing the old value 2
#2 (140552363525888) wrote 2 replacing the old value 3
#2 (140552363525888) wrote 4 replacing the old value 0
#3 (140552355133184) wrote 0 replacing the old value 4
#0 (140552380311296) wrote 0 replacing the old value 0
#0 (140552380311296) wrote 1 replacing the old value 4
#4 (140552346740480) wrote 0 replacing the old value 1
#4 (140552346740480) wrote 2 replacing the old value 0
#4 (140552346740480) wrote 3 replacing the old value 2
#4 (140552346740480) wrote 4 replacing the old value 3
```

ATOMIC

```
std::atomic<T>::compare_exchange_weak
std::atomic<T>::compare_exchange_strong
```

Spurious failure

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

...
expected = current.load();
do desired = function(expected);
while (!current.compare_exchange_weak(expected, desired));
```

ATOMIC

```
std::atomic<Integral>::fetch_add  -> object + value
std::atomic<Integral>::fetch_sub  -> object - value
std::atomic<Integral>::fetch_and  -> object & value
std::atomic<Integral>::fetch_or   -> object | value
std::atomic<Integral>::fetch_xor  -> object ^ value
```

```
std::atomic<FloatingPoint>::fetch_add -> object + value
std::atomic<FloatingPoint>::fetch_sub -> object - value
```

```
std::atomic<T*>::fetch_add -> object + value
std::atomic<T*>::fetch_sub -> object - value
```

MEMORY ORDER

```
enum class memory_order
{
    relaxed, consume, acquire, release, acq_rel, seq_cst
};

inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
inline constexpr memory_order memory_order_consume = memory_order::consume;
inline constexpr memory_order memory_order_acquire = memory_order::acquire;
inline constexpr memory_order memory_order_release = memory_order::release;
inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;
```

Sequentially-consistent ordering

```
void std::atomic<T>::store(T desired,
          std::memory_order order = std::memory_order_seq_cst) noexcept;
```

MEMORY ORDER

```
std::string data;
std::atomic<bool> ready{ false };

void thread1() {
   data = "very important bytes";
   ready.store(true, std::memory_order_relaxed);
}

void thread2() {
   while (!ready.load(std::memory_order_relaxed));
   std::cout << "data is ready: " << data << "\n";
}
```

MEMORY ORDER

```cpp
std::string data;
std::atomic<bool> ready{ false };

void thread1() {
   data = "very important bytes";
   ready.store(true, std::memory_order_relaxed);
}

void thread2() {
   while (!ready.load(std::memory_order_relaxed));
   std::cout << "data is ready: " << data << "\n";
}
```

Compiler can change execution order:

```cpp
ready.store(true, std::memory_order_relaxed);
data = "very important bytes";
```

MEMORY ORDER

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

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

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

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

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

z = ???

VOLATILE

Volatile variables are not cached or optimized.

```
int x;

auto y = x;
y = x;
x = 10;
x = 20;
```

optimize →

```
int x;

auto y = x;
x = 20;
```

```
volatile int x;

auto y = x;
y = x;
x = 10;
x = 20;
```

doesn't optimize →

```
volatile int x;

auto y = x;
y = x;
x = 10;
x = 20;
```