Atomics

- `std::atomic`
  - Can use a locking mechanism
  - Provides partial and full specializations for the following types

```
std::atomic<T*>
std::atomic<integral types>
std::atomic<user-defined types>
std::atomic<floating-point types> (C++20)
std::atomic<smart pointers> (C++20)
```

  - The user-defined type must be trivially copyable.

`std::atomic<bool>`

The atomic boolean `std::atomic<bool>`:

- Can explicitly set to `true` or `false`.
- Supports the function `compare_exchange_strong`.
  - Fundamental function for atomic operations
  - Compares and sets a value in an atomic operation

  - **Syntax**: `bool compare_exchange_strong(exp, des)`

  - **Strategy**: `atom.compare_exchange_strong(exp, des)`

    `*atom == exp` ⇒ `*atom = des;` returns `true`

    `*atom != exp` ⇒ `exp = *atom;` returns `false`

std::atomic<bool>

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

void setDataReady(){
  mySharedWork={1, 0, 3};
  dataReady= true;
}

void waitingForWork(){
  while (!dataReady.load()){
     sleep_for(milliseconds(5));
  }
  mySharedWork[1]= 2;
}
```

```
int main(){
  thread t1(waitingForWork);
  thread t2(setDataReady);
  t1.join();
  t2.join();
  for (auto v: mySharedWork){
    cout << v << " ";
  }
}  // 1 2 3
```

sequenced-before

synchronizes-with

atomicCondition.cpp

Atomics

Member Functions	Description
`is_lock_free`	Checks if the atomic object is lock-free.
`load`	Returns the value of the atomic.
`store`	Replaces the value of the atomic with the non-atomic.
`exchange`	Replaces the value with the new value. Returns the old value.
`compare_exchange_weak` `compare_exchange_strong`	`atom.compare_exchange_strong(expect, desir)` ▪ If `atom` is equal to `expect` returns `true`, `atom` becomes `desir`. ▪ If not returns `false`, `expect` is updated with `atom`.
`fetch_add, +=` `fetch_sub, -=`	Adds (substracts) the value and returns the previous value.
`++, --`	Increments or decrements the atomic.
`notify_one` (C++20)	Notifies one thread waiting on the atomic flag.
`notify_all` (C++20)	Notifies all threads waiting on the atomic flag.
`wait(val)` (C++20)	Waits for a notification and blocks as long as `atom == val` holds.

`fetch_mult.cpp`

`std::atomic<smart pointers>`

C++11 has `std::shared_ptr` for shared ownership.

- General rule: use smart pointers

- But:
  - The handling of the control block is thread-safe.
  - Access to the resource is not thread-safe.

`std::atomic<smart pointers>`

Three reasons for an atomic smart pointer.

- Consistency
  - `std::shared_ptr` is the only non-atomic type that supports atomic operations
- Correctness
  - The correct use of the atomic operation rests on the shoulder of the user
  - `std::atomic_store(&sharPtr, localPtr)` **!=** `sharPtr = localPtr`
- Speed
  - `std::shared_ptr` is designed for general use

`std::atomic<smart pointers>`

Partial specialization of `std::atomic`

- `std::atomic<std::shared_ptr<T>>`
- `std::atomic<std::weak_ptr<T>>`

All implementations use currently (2023) a locking mechanism.