In C++, the `stdatomic.h` header provides a set of atomic operations and types for working with atomic variables in multi-threaded environments. The operations are designed to ensure thread safety while working with shared variables in concurrent programming. Below is a list of the APIs and declarations related to atomic operations in `stdatomic.h`(C++11 and later):

1. Atomic Types

- **`std::atomic<T>`**: A template class that provides atomic operations for a given type `T`. It is the core type for atomic variables.
- **`std::atomic_flag`**: A special atomic type for flag-style operations (typically used for simple boolean flags).

2. Atomic Operations

These operations provide atomic access to variables with specified memory order semantics:

`std::atomic::load`: Atomically loads the value of the atomic variable.

cpp

Copy

```cpp
T load(std::memory_order order = std::memory_order_seq_cst) const;
```

-

`std::atomic::store`: Atomically stores a value into the atomic variable.

cpp

Copy

```cpp
void store(T desired, std::memory_order order = std::memory_order_seq_cst);
```

-

`std::atomic::exchange`: Atomically exchanges the value of the atomic variable with a new value.

cpp

Copy

```cpp
T exchange(T desired, std::memory_order order = std::memory_order_seq_cst);
```

-

`std::atomic::compare_exchange_weak`: Performs an atomic compare-and-swap (CAS) operation. This operation checks whether the current value of the atomic variable is equal to the expected value and, if so, replaces it with a new value.

cpp

Copy

```cpp
bool compare_exchange_weak(T& expected, T desired, std::memory_order success, std::memory_order failure);
```

-
    - `compare_exchange_weak` may fail spuriously (i.e., even if the expected value is equal to the current value).

`std::atomic::compare_exchange_strong`: Similar to `compare_exchange_weak`, but does not fail spuriously.

cpp

Copy

```cpp
bool compare_exchange_strong(T& expected, T desired, std::memory_order success, std::memory_order failure);
```

-

`std::atomic::fetch_add`: Atomically adds a value to the atomic variable and returns the old value.

cpp

Copy

```cpp
T fetch_add(T arg, std::memory_order order = std::memory_order_seq_cst);
```

-

`std::atomic::fetch_sub`: Atomically subtracts a value from the atomic variable and returns the old value.

cpp

Copy

```cpp
T fetch_sub(T arg, std::memory_order order = std::memory_order_seq_cst);
```

-

`std::atomic::fetch_or`: Atomically performs a bitwise OR operation on the atomic variable and returns the old value.
cpp
Copy
```
T fetch_or(T arg, std::memory_order order = std::memory_order_seq_cst);
```

-

`std::atomic::fetch_and`: Atomically performs a bitwise AND operation on the atomic variable and returns the old value.
cpp
Copy
```
T fetch_and(T arg, std::memory_order order = std::memory_order_seq_cst);
```

-

`std::atomic::fetch_xor`: Atomically performs a bitwise XOR operation on the atomic variable and returns the old value.
cpp
Copy
```
T fetch_xor(T arg, std::memory_order order = std::memory_order_seq_cst);
```

-

3. Atomic Flag Operations (`std::atomic_flag`)

`std::atomic_flag` is a simpler atomic type used mainly for flag-based synchronization.

`std::atomic_flag::clear`: Clears the atomic flag (sets it to `false`).
cpp
Copy
```
void clear(std::memory_order order = std::memory_order_seq_cst);
```

-

`std::atomic_flag::test_and_set`: Tests whether the atomic flag is set (true), and if not, sets it to true atomically.
cpp
Copy
```
bool test_and_set(std::memory_order order = std::memory_order_seq_cst);
```

-

4. Memory Order Types

In C++11, memory ordering defines the relationship between atomic operations with respect to the visibility of memory writes between different threads. The available memory order types are:

- **`std::memory_order_relaxed`**: No synchronization or ordering constraints.
- **`std::memory_order_consume`**: Ensures that dependencies on the consumed value are synchronized.
- **`std::memory_order_acquire`**: Ensures that all previous operations on the current thread are completed before the atomic operation.
- **`std::memory_order_release`**: Ensures that all subsequent operations on the current thread are not started until after the atomic operation.
- **`std::memory_order_acq_rel`**: Combines acquire and release semantics.
- **`std::memory_order_seq_cst`**: The default memory order, providing the strongest ordering (sequential consistency).

5. Special Operations for Atomic Types (`std::atomic<T>`)

`std::atomic<T>::is_lock_free`: Determines if atomic operations on a given type are lock-free.
cpp
Copy
```
bool is_lock_free() const noexcept;
```

-

6. Atomic Utilities

`std::atomic::operator=`: Atomically assigns a value to an atomic variable.
cpp
Copy

```
std::atomic<T>& operator=(T desired);
```

-

`std::atomic::operator T()`: Converts the atomic variable to a regular value.
cpp
Copy

```
operator T() const;
```

-

7. `std::atomic_thread_fence` (Global Function)

This function creates a fence, enforcing a memory ordering between atomic operations in different threads. It acts as a barrier to control memory ordering for operations across threads.

cpp
Copy

```
void atomic_thread_fence(std::memory_order order);
```

8. `std::atomic_signal_fence` (Global Function)

This function acts as a barrier for operations that are related to signals and asynchronous events. It prevents certain kinds of optimizations that could cause issues with signal handling.

cpp
Copy

```
void atomic_signal_fence(std::memory_order order);
```

Summary of Key API Functions in `stdatomic.h`:

- **`std::atomic<T>`**: Atomic wrapper class for any type `T`.
- **Atomic operations** like `load()`, `store()`, `exchange()`, and `compare_exchange_*`.
- **`std::atomic_flag`**: A simpler atomic flag type for basic synchronization.
- **Atomic arithmetic and bitwise operations**: `fetch_add()`, `fetch_sub()`, `fetch_or()`, `fetch_and()`, `fetch_xor()`.
- **Memory ordering**: `std::memory_order_*` options define how memory operations are ordered with respect to atomic operations.
- **Atomic fences**: `atomic_thread_fence()` and `atomic_signal_fence()` for enforcing memory order constraints.

These atomic operations and types provide a foundation for building thread-safe, high-performance applications where shared resources are accessed concurrently by multiple threads.

You said:
List all APIs/declarations related to atoimic operation in stdatomic.h in c

ChatGPT said:

In C, the header `<stdatomic.h>` provides a set of atomic operations and types designed to ensure safe access to shared variables in multi-threaded environments. Below is a list of all the APIs and declarations related to atomic operations in the `<stdatomic.h>` header for C (C11 and later).

1. Atomic Types

`atomic`: A generic atomic type that provides atomic operations for a given type.
c

Copy

```
_Atomic(type) var;
```

-

`atomic_flag`: A special atomic type used for flag-based synchronization, typically for spinlocks.

c

Copy

```
atomic_flag flag;
```

-

2. Atomic Operations

These are functions for performing atomic operations on variables of atomic types.

`atomic_load`: Atomically loads the value of an atomic variable.

c

Copy

```
T atomic_load(const atomic_type *obj);
```

-

`atomic_store`: Atomically stores a value into an atomic variable.

c

Copy

```
void atomic_store(atomic_type *obj, T value);
```

-

`atomic_exchange`: Atomically exchanges the value of the atomic variable with a new value.

c

Copy

```
T atomic_exchange(atomic_type *obj, T value);
```

-

`atomic_compare_exchange_weak`: Atomically compares and exchanges values. If the current value matches the expected value, it is replaced with a new value.

c

Copy

```
bool atomic_compare_exchange_weak(atomic_type *obj, T *expected, T desired);
```

-

`atomic_compare_exchange_strong`: Similar to `atomic_compare_exchange_weak`, but less likely to fail spuriously.

c

Copy

```
bool atomic_compare_exchange_strong(atomic_type *obj, T *expected, T desired);
```

-

`atomic_fetch_add`: Atomically adds a value to the atomic variable and returns the old value.

c

Copy

```
T atomic_fetch_add(atomic_type *obj, T value);
```

-

`atomic_fetch_sub`: Atomically subtracts a value from the atomic variable and returns the old value.

c

Copy

```
T atomic_fetch_sub(atomic_type *obj, T value);
```

-

`atomic_fetch_or`: Atomically performs a bitwise OR operation on the atomic variable and returns the old value.
c
Copy

```
T atomic_fetch_or(atomic_type *obj, T value);
```

-

`atomic_fetch_and`: Atomically performs a bitwise AND operation on the atomic variable and returns the old value.
c
Copy

```
T atomic_fetch_and(atomic_type *obj, T value);
```

-

`atomic_fetch_xor`: Atomically performs a bitwise XOR operation on the atomic variable and returns the old value.
c
Copy

```
T atomic_fetch_xor(atomic_type *obj, T value);
```

-

3. Atomic Flag Operations (`atomic_flag`)

The atomic flag provides a simple atomic flag-style mechanism (often used for spinlocks).

`atomic_flag_clear`: Clears the atomic flag (sets it to `false`).
c
Copy

```
void atomic_flag_clear(atomic_flag *flag);
```

-

`atomic_flag_test_and_set`: Tests if the atomic flag is set (true). If not, it sets it to true atomically.
c
Copy

```
bool atomic_flag_test_and_set(atomic_flag *flag);
```

-

4. Memory Ordering

These functions and types are used to control the memory ordering of atomic operations. Memory order determines the visibility and ordering of operations across different threads.

- **Memory Order Types**:
  - **`memory_order_relaxed`**: No ordering or synchronization constraints.
  - **`memory_order_consume`**: Ensures proper ordering of operations that depend on the consumed value.
  - **`memory_order_acquire`**: Ensures that all operations before the atomic operation in the program are completed before the atomic operation itself.
  - **`memory_order_release`**: Ensures that all operations after the atomic operation in the program are completed after the atomic operation.
  - **`memory_order_acq_rel`**: Combines `acquire` and `release` semantics.
  - **`memory_order_seq_cst`**: The strongest memory order; ensures sequential consistency.

`atomic_thread_fence`: Enforces a memory barrier between atomic operations to ensure ordering.
c
Copy

```
void atomic_thread_fence(memory_order order);
```

-

`atomic_signal_fence`: Similar to `atomic_thread_fence`, but specifically for operations affected by signals.

c

Copy

```c
void atomic_signal_fence(memory_order order);
```

-

5. Atomic Utilities

`atomic_is_lock_free`: Checks if atomic operations on a given atomic type are lock-free.

c

Copy

```c
bool atomic_is_lock_free(const atomic_type *obj);
```

-

6. Atomic Initializer Macros

`ATOMIC_VAR_INIT`: Initializes an atomic variable with a specified value.

c

Copy

```c
#define ATOMIC_VAR_INIT(value) value
```

-

`ATOMIC_FLAG_INIT`: Initializes an atomic flag.

c

Copy

```c
#define ATOMIC_FLAG_INIT (atomic_flag){0}
```

-

Summary of Key API Functions in `<stdatomic.h>` (C):

- **`atomic_load`**: Loads an atomic value.
- **`atomic_store`**: Stores a value atomically.
- **`atomic_exchange`**: Atomically exchanges a value.
- **`atomic_compare_exchange_*`**: Performs atomic compare-and-swap operations (`weak` or `strong`).
- **`atomic_fetch_*`**: Atomically fetches and updates the value (e.g., `fetch_add`, `fetch_sub`, `fetch_or`, etc.).
- **`atomic_flag_clear`**: Clears an atomic flag.
- **`atomic_flag_test_and_set`**: Tests and sets an atomic flag.
- **`atomic_is_lock_free`**: Checks if an atomic variable's operations are lock-free.
- **`atomic_thread_fence`**: Memory barrier to enforce ordering of atomic operations.
- **`atomic_signal_fence`**: Memory barrier specifically for signal-related atomic operations.
- **`ATOMIC_VAR_INIT`**: Initializes an atomic variable.
- **`ATOMIC_FLAG_INIT`**: Initializes an atomic flag.

These atomic operations allow C programs to perform thread-safe operations on shared variables, ensuring proper synchronization and memory ordering in concurrent environments.