КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 24.2 — Атомарные типы

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2021

Оглавление

Атомарные типы

```
#include <atomic>
```

Атомарные типы — это типы, которые инкапсулируют значение, доступ к которому гарантированно не вызывает гонок данных и может использоваться для синхронизации доступа к памяти между различными потоками.

В этом заголовке объявляются два класса C++, **`atomic`** и **`atomic_flag`**, которые реализуют все функции атомарных типов в автономных классах.

Заголовок также объявляет полный набор типов и функций в стиле C, совместимых с атомарной поддержкой в C.

std::atomic_flag — флаг атомарного доступа

```
struct atomic_flag;
```

Атомарные флаги — это логические (boolean) атомарные объекты, которые поддерживают две операции — test-and-set (проверка-и-установка) и clear (сброс).

Атомарные флаги лишены блокировок (это единственный тип, который гарантированно свободен от блокировок во всех реализациях библиотеки).

Конструкторы

```
atomic_flag( ) noexcept = default;
atomic_flag(const atomic_flag&T) = delete;
```

Объект типа **atomic_flag** после созданиия находится в неопределенном состоянии (установленном или сброшенном), если он явно не инициализирован как **ATOMIC_FLAG_INIT**.

ATOMIC_FLAG_INIT — если объект инициализируется этим макросом, он гарантированно будет создан сброшенным.

Каким образом будет инициализироваться объект как **ATOMIC_FLAG_INIT**, простым вызовом конструктора по умолчанию или другими способами, зависит от конкретной реализации библиотеки.

Значения **atomic_flag** не могут ни копироваться, ни перемещаться (и соответсвующие конструкторы и присваивания удалены).

std::memory_order — порядок доступа к памяти

```
enum memory_order;
```

Используется в качестве аргумента для функций, которые выполняют атомарные операции, чтобы указать, как синхронизируются другие операции в разных потоках.

Порядок доступа определяется как:

```
typedef enum memory_order {
    memory_order_relaxed,   // relaxed
    memory_order_consume,   // consume
    memory_order_acquire,   // acquire
    memory_order_release,   // release
    memory_order_acq_rel,   // acquire/release
    memory_order_seq_cst    // sequentially consistent
} memory_order;
```

Все атомарные операции производят по отношению к атомарному объекту, когда к нему обращаются из нескольких потоков, имеют четко определенное поведение — каждая атомарная операция над объектом выполняется полностью, прежде чем любая другая атомарная операция сможет получить к нему доступ. Это гарантирует отсутствие гонок данных на этих объектах, и именно эта функция определяет атомарность.

Но любой поток может выполнять операции с ячейками памяти, отличными от самого атомарного объекта, и эти другие операции могут вызывать побочные эффекты, видимые в других потоках.

Аргументы данного типа позволяют указать порядок доступа к памяти для операции, который определяет, как эти (возможно, не атомарные) видимые побочные эффекты синхронизируются между потоками, с использованием атомарных операций в качестве точек синхронизации:

`memory_order_relaxed` — операция должна произойти в какой-то момент атомарно.

Это самый неопределенный порядок памяти, не дающий никаких гарантий относительно того, как доступ к памяти в разных потоках упорядочен относительно атомарной операции.

`memory_order_consume` — [относится к операциям загрузки] — операция должна выполняться после того, как произошли все обращения к памяти в освобождающем потоке, которые несут зависимость от этой операции освобождения (и которые имеют видимые побочные эффекты в загружающем потоке).

`memory_order_acquire` — [относится к операциям загрузки] — операция должна выполняться после того, как произошли все обращения к памяти в освобождающем потоке (которые имеют видимые побочные эффекты в загружающем потоке).

`memory_order_release` — [относится к операциям сохранения] — операция должна выполняться перед операцией загрузки или захвата, служа точкой синхронизации для других обращений к памяти, которые могут иметь видимые побочные эффекты в загружающем потоке.

`memory_order_acq_rel` — [относится к операциям загрузки/сохранения] — операция загружает с захватом и сохраняет с освобождением (как определено выше для **`memory_order_acquire`** и **`memory_order_release`**).

`memory_order_seq_cst` — операция упорядочивается последовательно согласованным образом — все операции, использующие данный порядок памяти, должны выполняться после того, как все обращения к памяти, которые могут иметь видимые побочные эффекты для других задействованных потоков, уже произошли.

Это самый строгий порядок памяти, гарантирующий наименее неожиданные побочные эффекты между взаимодействиями потоков при неатомарном доступе к памяти.

Для загрузок (чтение) с целью использования значения и захвата последовательно согласованные операции в памяти считаются операциями освобождения.

std::atomic_flag::test_and_set — проверить и установить (захват флага)

```
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;
```

Устанавливает **atomic_flag** и возвращает его состояние непосредственно перед вызовом.

Вся операция является атомарной (атомарная операция чтения-изменения-записи).

Между моментом чтения значения флага (которое должно быть возвращено) и моментом его изменения даной функцией никакие другие потоки на значение флага не влияют.

sync — режим синхронизации работы. Это может быть любое из возможных значений перечисляемого типа **memory_order**:

memory_order_relaxed — ослабленный;

memory_order_consume — режим использования;

memory_order_acquire — режим захвата;

memory_order_release — режим освобождения;

memory_order_acq_rel — режим захвата/освобождения — читает как при операции захвата и записывает как при операции освобождения.

memory_order_seq_cst — последовательно согласованный режим — синхронизирует все видимые побочные эффекты с другими последовательно согласованными операциями, следуя единому общему порядку.

Возвращаемое значение

истина — если флаг перед вызовом был установлен, иначе ложь.

std::atomic_flag::clear — освободить флаг

```
void clear (memory_order sync = memory_order_seq_cst) volatile noexcept;
void clear (memory_order sync = memory_order_seq_cst) noexcept;
```

Очищает **atomic_flag** (т.е. устанавливает значение **false**).

Очистка **atomic_flag** приводит к тому, что следующий вызов **atomic_flag::test_and_set** для этого объекта возвращает **false**. Операция является атомарной и следует порядку памяти, указанному в **sync**.

C++11

memory_order_relaxed — ослабленный;

memory_order_consume — режим использования;

memory_order_acquire — режим захвата;

memory_order_release — режим освобождения;

memory_order_seq_cst — последовательно согласованный режим.

C++14

memory_order_relaxed — ослабленный;

memory_order_release — режим освобождения;

memory_order_seq_cst — последовательно согласованный режим.

Пример atomic_flag в качестве спинлока (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;
std::stringstream stream;

void append_number(int x) {

  while (lock_stream.test_and_set()) {}
  stream << "thread #" << x << '\n';
  lock_stream.clear();
}

int main () {

  std::vector<std::thread> threads;
  for (int i = 0; i <= 9; ++i) threads.push_back(std::thread(append_number,i));
  for (auto& th : threads) th.join();

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

std::atomic — атомарный тип

```
template <class T> struct atomic;
```

Объекты атомарных типов содержат значение определенного типа (T).

Основная характеристика атомарных объектов заключается в том, что доступ к этому содержащемуся значению из разных потоков не может вызвать гонку данных, т.е. процесс доступа является четко определенным поведением, которое должным образом упорядочено.

Как правило, возможность вызвать гонку данных за доступ одновременно к одному и тому же объекту квалифицирует операцию как «неопределенное поведение».

Кроме того, атомарные объекты имеют возможность синхронизировать доступ к другим неатомарным объектам в своих потоках.

Общие атомарный операции

`is_lock_free` — не заблокирован ли?
`store` — изменить содержащееся значение
`load` — прочитать содержащееся значение
`operator T` — оператор приведения к типу
`exchange` — доступ к содержащемуся значению и его изменение
`compare_exchange_weak` — сравнть и обменять (слабая версия)
`compare_exchange_strong` — сравнть и обменять (сильная версия)

Операции, поддерживаемые определенными специализациями
Работают с целочисленными значениями и/или указателями.

`fetch_add` — добавить к содержащемуся значению
`fetch_sub` — вычесть из содержащегося значения
`fetch_and` — применить побитовое И к содержащемуся значению
`fetch_or` — применить побитовое ИЛИ к содержащемуся значению
`fetch_xor` — применить побитовое исключающее ИЛИ к содержащемуся значению
`operator++` — инкрементировать содержащееся значение
`operator--` — декрементировать содержащееся значение
`atomic::operator (comp. assign.)` — составные присваивания

Конструкторы

(1) по умолчанию

```
atomic() noexcept = default;
```

Оставляет атомарный объект в неинициализированном состоянии.

Неинициализированный атомарный объект может быть позже инициализирован вызовом **`atomic_init()`**.

(2) с инициализацией

```
constexpr atomic (T val) noexcept;
```

Инициализирует объект с помощью **`val`**.

(3) копирования (исключен, как таковой)

```
atomic (const atomic&) = delete;
```

Атомарные объекты нельзя ни копировать, ни перемещать.

std::atomic::operator= — оператор присваивания

(1) установить значение

```
T operator= (T val) noexcept;
T operator= (T val) volatile noexcept;
```

Заменяет хранящееся значение на **`val`**.

Эта операция является атомарной и использует последовательную согласованность (**`memory_order_seq_cst`**). Чтобы изменить значение, используя другой порядком памяти, см. **`atomic::store()`**.

(2) оператор копирования (deleted)

```
atomic& operator= (const atomic&) = delete;
atomic& operator= (const atomic&) volatile = delete;
```

Для атомарных объектов не определено присваивание копированием.

`val` — значение, которое копируется в содержащийся объект.

`T` — это параметр шаблона атомарного объекта (тип содержащегося значения).

Пример использования оператора присваивания

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic
#include <thread>         // std::thread, std::this_thread::yield

std::atomic<int> foo = 0;

void set_foo(int x) { foo = x; }

void print_foo() {
  while (foo == 0) {              // ожидать, пока foo равен 0
    std::this_thread::yield();    // освобождаем процессор
  }
  std::cout << "foo: " << foo << '\n';
}

int main () {

  std::thread first(print_foo);
  std::thread second(set_foo, 10);
  first.join();
  second.join();
  return 0;
}
```

Вывод

```
foo: 10
```

is_lock_free — не заблокирован ли?

```
bool is_lock_free( ) const volatile noexcept;
bool is_lock_free( ) const noexcept;
```

Указывает, свободен ли объект от блокировки.

Объект без блокировки не вызывает блокировку других потоков при доступе (возможно, с использованием какой-то транзакционной памяти для этого типа).

Значение, возвращаемое этой функцией, совместимо со значениями, возвращаемыми для всех других объектов того же типа.

Возвращаемое значение

true, если не заблокирован.

store — изменить содержащееся значение

```
void store (T val, memory_order sync = memory_order_seq_cst) volatile noexcept;
void store (T val, memory_order sync = memory_order_seq_cst) noexcept;
```

Заменяет содержащееся значение на **val**.

Операция является атомарной и следует порядку памяти, указанному в sync.

val — значение для копирования в содержащийся объект.

T — это параметр шаблона атомарного объекта (тип содержащегося значения).

sync — режим синхронизации операции. Режим должен быть одним из следующих возможных значений перечислимого типа **memory_order**:

memory_order_relaxed — ослабленный;

memory_order_release — режим освобождения;

memory_order_seq_cst — последовательно согласованный режим.

load — прочитать содержащееся значение

```
T load (memory_order sync = memory_order_seq_cst) const volatile noexcept;
T load (memory_order sync = memory_order_seq_cst) const noexcept;
```

Возвращает содержащееся значение.
Операция является атомарной и следует порядку памяти, указанному в **sync**.

memory_order_relaxed — ослабленный;
memory_order_consume — режим использования;
memory_order_acquire — режим захвата;
memory_order_seq_cst — последовательно согласованный режим

Возвращаемое значение
Значение объекта

Пример atomic::load/store

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic, std::memory_order_relaxed
#include <thread>         // std::thread

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

void set_foo(int x) {
  foo.store(x, std::memory_order_relaxed);     // set value atomically
}

void print_foo() {
  int x;
  do {
    x = foo.load(std::memory_order_relaxed);  // get value atomically
  } while (x == 0);
  std::cout << "foo: " << x << '\n';
}

int main () {

  std::thread first(print_foo);
  std::thread second(set_foo, 10);
  first.join();
  second.join();
  return 0;
}
```

exchange — доступ к содержащемуся значению и его изменение

```
T exchange (T val, memory_order sync = memory_order_seq_cst) volatile noexcept;
T exchange (T val, memory_order sync = memory_order_seq_cst) noexcept;
```

Заменяет содержащееся значение на **val** и возвращает значение, которое было непосредственно перед этим.

Вся операция является атомарной (атомарная операция чтения-изменения-записи) — на значение не влияют другие потоки между моментом чтения его значения (которое должно быть возвращено) и моментом его изменения этой функцией.

val — значение для копирования в содержащийся объект.

T — это параметр шаблона атомарного объекта (тип содержащегося значения).

sync — режим синхронизации операции. Режим должен быть одним из следующих возможных значений перечислимого типа **memory_order**:

memory_order_relaxed — ослабленный;

memory_order_consume — режим использования;

memory_order_acquire — режим захвата;

memory_order_release — режим освобождения;

memory_order_acq_rel — режим захвата/освобождения

memory_order_seq_cst — последовательно согласованный режим

Пример atomic<bool> против atomic_flag

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic, std::atomic_flag, ATOMIC_FLAG_INIT
#include <thread>         // std::thread, std::this_thread::yield
#include <vector>         // std::vector

std::atomic<bool> ready(false);              // может проверяться без установки
std::atomic_flag winner = ATOMIC_FLAG_INIT; // всегда устанавливается при проверке

void count1m (int id) {
  while (!ready) { std::this_thread::yield(); }   // ждем сигнала готовности
  for (int i = 0; i < 1000000; ++i) {}            // go!, считаем до миллиона
  if (!winner.test_and_set()) {std::cout << "поток #" << id << " победил!\n";}
};

int main () {
  std::vector<std::thread> threads;
  std::cout << "Порождаем 10 потоков, считающих до одного миллиона...\n";
  for (int i = 1; i <= 10; ++i) threads.push_back(std::thread(count1m, i));
  ready = true;
  for (auto& th : threads) th.join();
}
```

Вывод

```
Порождаем 10 потоков, считающих до одного миллиона...
поток #4 победил
```

compare_exchange_weak — сравнить и обменять (слабая версия)

(1)

```
bool compare_exchange_weak (T& expected, T val,
           memory_order sync = memory_order_seq_cst) volatile noexcept;

bool compare_exchange_weak (T& expected, T val,
           memory_order sync = memory_order_seq_cst) noexcept;
```

(2)

```
bool compare_exchange_weak (T& expected, T val,
           memory_order success, memory_order failure) volatile noexcept;

bool compare_exchange_weak (T& expected, T val,
           memory_order success, memory_order failure) noexcept;
```

Сравнивает содержимое содержащегося значения атомарного объекта с ожидаемым:

- если **true**, он заменяет содержащееся значение на **val** (подобно **store**).
- если **false**, он заменяет ожидаемое на содержащееся значение.

Функция всегда обращается к содержащемуся значению чтобы его прочитать и, если сравнение верно, она его переписывает.

Вся операция атомарна — значение не может быть изменено другими потоками между моментом чтения его значения и моментом его замены.

Порядок доступа к памяти, используемый в (2), зависит от результата сравнения:

- если **`true`**, используется **success**;
- если **`false`**, используется **failure**.

Следует обратить внимание, что эта функция напрямую сравнивает физическое содержимое содержащегося значения с физическим содержимым ожидаемого. Это может привести к неудачным сравнениям значений, которые сравниваются на равенство с использованием оператора **==**, если базовый тип имеет биты заполнения, значения прерывания или альтернативные представления одного и того же значения.

В отличие от **`compare_exchange_strong`**, для этой слабой версии разрешается ложный сбой, который возвращает **`false`**, даже если **`expected`** действительно эквивалентно с содержащимся объектом. Это может быть приемлемым поведением для определенных алгоритмов на основе цикла и может привести к значительному повышению производительности на некоторых платформах. При этих ложных сбоях функция возвращает **`false`**, не изменяя **`expected`**.

Для алгоритмов без цикла обычно рекомендуется **`compare_exchange_strong`**.

compare_exchange_strong — сравнить и обменять (сильная версия)

(1)

```
bool compare_exchange_strong(T& expected, T val,
           memory_order sync = memory_order_seq_cst) volatile noexcept;
bool compare_exchange_strong(T& expected, T val,
           memory_order sync = memory_order_seq_cst) noexcept;
```

(2)

```
bool compare_exchange_strong (T& expected, T val,
           memory_order success, memory_order failure) volatile noexcept;
bool compare_exchange_strong (T& expected, T val,
           memory_order success, memory_order failure) noexcept;
```

Сравнивает содержимое содержащегося в объекте значения с ожидаемым (**expected**):
- если **true**, заменяет содержащееся значение на **val** (аналогично тому, как это делает **store**).
- если **false**, он заменяет **expected** на содержащееся в объекте значение.

Функция всегда обращается к содержащемуся значению, чтобы его прочитать и, если сравнение на эквивалентность верно, она его заменяет.

Вся операция атомарна — значение не может быть изменено другими потоками между моментом чтения его значения и моментом его замены.

Порядок памяти, используемый в (2), зависит от результата сравнения: если он **true**, используется **success**; если **false**, используется **failure**.

Эта функция напрямую сравнивает физическое содержимое содержащегося значения с содержимым **expected**. В отличие от **compare_exchange_weak**, эта сильная версия должна всегда возвращать **true**, если **expected** действительно равно содержащемуся объекту, не допуская ложных сбоев. Однако на определенных машинах и для определенных алгоритмов, которые проверяют это отношение эквивалентности в цикле, **compare_exchange_weak** может привести к значительно лучшей производительности.

Параметры

expected — ссылка на объект, значение которого сравнивается со значением, содержащемся в объекте, и которое в случае несоответствия перезаписывается этим самым содержащимся значением.

T — параметр шаблона атомарного объекта (тип содержащегося значения).

val — значение, которое будет скопировано в содержащийся объект в том случае, если **expected** совпадает с содержащимся значением.

sync — режим синхронизации для работы. Это может быть любое из возможных значений перечисляемого типа **memory_order**:

memory_order_relaxed — ослабленный;

memory_order_consume — режим использования;

memory_order_acquire — режим захвата;

memory_order_release — режим освобождения;

memory_order_acq_rel — режим захвата/освобождения

memory_order_seq_cst — последовательно согласованный режим

success — режим синхронизации для операции в случае, если **expected** соответствует содержащемуся значению.

failure — режим синхронизации для операции в случае, если **expected** не соответствует содержащемуся значению. Этот режим не должен быть более сильным режимом, чем в случае успеха, и не должен быть ни **memory_order_release**, ни **memory_order_acq_rel**.

Пример atomic::compare_exchange_weak

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic
#include <thread>         // std::thread
#include <vector>         // std::vector

// простой глобальный связный список
struct Node { int value; Node* next; };
std::atomic<Node*> list_head(nullptr);

void append(int val) {                          // append an element to the list
  Node* oldHead = list_head;
  Node* newNode = new Node {val, oldHead};

  // дальше следует эквивалент list_head = newNode,
  // но в безопасном в отношении гонок режиме:
  while (!list_head.compare_exchange_weak(oldHead, newNode))
    newNode->next = oldHead;
}
```

```
int main () {

  // spawn 10 threads to fill the linked list:
  std::vector<std::thread> threads;
  for (int i = 0; i < 10; ++i) threads.push_back(std::thread(append, i));
  for (auto& th : threads) th.join();

  // print contents:
  for (Node* it = list_head; it != nullptr; it = it->next) {
    std::cout << ' ' << it->value;
  }
  std::cout << '\n';

  // cleanup:
  Node* it; while (it=list_head) {list_head=it->next; delete it;}

  return 0;
}
```

Вывод

```
 9 8 7 6 5 4 3 2 1 0
```