КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 24.5 — Мьютексы II

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

Функции-не члены

`lock()` – блокировка нескольких мьютексов
`try_lock()` – попытка заблокировать несколько мьютексов
`call_once()` – однократный вызов функции

std::lock — заблокировать несколько мьютексов

```
template <class Mutex1, class Mutex2, class... Mutexes>
  void lock (Mutex1& a, Mutex2& b, Mutexes&... cde);
```

Блокирует все объекты, переданные в качестве аргументов, при необходимости блокируя вызывающий поток.

Функция блокирует объекты, используя неопределенную последовательность вызовов их членов **lock()**, **try_lock()** и **unlock()**, что гарантирует блокировку всех аргументов при возврате (без создания каких-либо взаимоблокировок).

Если функция не может заблокировать все объекты (например, из-за того, что один из ее внутренних вызовов вызвал исключение), функция сначала разблокирует все объекты, которые она успешно заблокировала (если таковые имеются), прежде чем завершится сбоем.

a, b, **cde** — объекты, подлежащие блокировке

Mutex1, **Mutex2** и **Mutexes** — блокируемые типы

Пример

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex, std::lock

std::mutex foo, bar;

void task_a () {
  // foo.lock(); bar.lock(); // replaced by:
  std::lock (foo, bar);
  std::cout << "task a\n";
  foo.unlock();
  bar.unlock();
}

void task_b () {
  // bar.lock(); foo.lock(); // replaced by:
  std::lock (bar, foo);
  std::cout << "task b\n";
  bar.unlock();
  foo.unlock();
}
```

```
int main () {
  std::thread th1(task_a);
  std::thread th2(task_b);

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

  return 0;
}
```

std::try_lock — попытка заблокировать несколько мьютексов

```
template <class Mutex1, class Mutex2, class... Mutexes>
int try_lock(Mutex1& a, Mutex2& b, Mutexes&... cde);
```

Пытается заблокировать все объекты, переданные в качестве аргументов, используя их неблокирующие функции-члены **`try_lock()`**.

Функция вызывает функцию-член **`try_lock()`** для каждого аргумента (сначала **`a`**, затем **`b`** и, наконец, остальные в **`cde`**, в том же порядке), пока либо все вызовы не будут успешными, либо как только один из вызовов завершится неудачно (либо путем возврата **`false`** или выбросив исключение).

Если функция завершается из-за сбоя вызова, для всех объектов, для которых вызов **`try_lock`** был успешным, вызывается разблокировка и функция возвращает порядковый номер аргумента объекта, блокировка которого не удалась.

Для остальных объектов в списке аргументов дальнейшие вызовы не выполняются.

Блокировки

Блокировки — это объекты, которые управляют мьютексом, связывая его доступ со своим собственным временем жизни.

В **<mutex>** предоставляется два класса:

lock_guard — удерживает мьютекс заблокированным.

unique_lock — управляет мьютексом в обоих состояниях — заблокирован и разблокирован.

std::lock_guard — защита блокировки

Защита блокировки — это объект, который управляет объектом мьютекса, постоянно удерживая его заблокированным.

```
template <class Mutex> class lock_guard;
```

При построении объект мьютекса блокируется вызывающим потоком, а при уничтожении мьютекс разблокируется.

Это простейшая блокировка, которая особенно полезна в качестве объекта с автоматической продолжительностью, которая длится до конца его существования. Таким образом, она гарантирует, что объект мьютекса будет должным образом разблокирован в случае возникновения исключения.

Однако, слудует заметить, что объект типа **`lock_guard`** никоим образом не управляет временем жизни мьютекса — продолжительность существования объекта мьютекса должна простираться, по крайней мере, до разрушения **`lock_guard`**, который его блокирует.

Параметры шаблона

`Mutex` — мьютекс-подобный тип.

Это должен быть базовый блокируемый тип, такой как мьютекс (см. BasicLockable).

Функции-члены

(constructor) — construct lock_guard (public member function)

(destructor) — destroy lock_guard (unlocking mutex) (public member function)

Конструктор

(1) создание и блокировка при инициализации

```
explicit lock_guard (mutex_type& m);
```

Созданный объект управляет мьютексом **m** и блокирует его (вызывая **m.lock()**).

m — объект мьютекса, которым будет управлять созданный объект типа **lock_guard**.

mutex_type — это параметр шаблона **lock_guard**.

Аргумент **m** доступен как по чтению, так и по записи (как атомарная операция, не вызывающая гонок данных). Объект хранит ссылку на **m**, которая изменяется при уничтожении.

(2) создание с принятием существующей блокировки

```
lock_guard(mutex_type& m, adopt_lock_t tag);
```

Созданный объект управляет мьютексом **m**, который в настоящее время заблокирован потоком, создающим болкировку.

tag — этот аргумент используется просто для выбора определенного конструктора (значения этих типов не имеют состояния).

Принимает одно из следующих значений:

(тега нет) — блокировка при создании путем автоматического вызова члена **lock()**.

adopt_lock — принять текущую блокировку (предположим, что она уже заблокирована потоком).

adopt_lock_t — это тип объекта **adopt_lock**.

(3) Конструктор копирования [deleted]

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

Удален (объекты **lock_guard** нельзя ни копировать, ни перемещать.

Созданный объект сохраняет мьютекс **m** заблокированным и поддерживает на него ссылку, которая используется для его разблокировки при уничтожении (путем вызова для его члена **unlock()**).

std::lock_guard::~lock_guard — деструктор

```
~lock_guard();
```

Перед уничтожением деструктор вызывает член **unlock()** для мьютекса, которым он управляет. Пр этом управляемый объект мьютекса не уничтожается.

Пример lock_guard

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex, std::lock_guard
#include <stdexcept>      // std::logic_error

std::mutex mtx;           // мьютекс под управление блокировкой

void print_even(int x) {
  if (x%2 == 0)
      std::cout << x << " is even\n";
  else
      throw (std::logic_error("нечет"));
}

// функция потока
void print_thread_id(int id) {
  try {
    // использование lock_guard для блокировки mtx гарантирует разблокировку
    // как при уничтожении, так и в случае возникновения исключения
    std::lock_guard<std::mutex> lck(mtx);
    print_even(id);
  } catch (std::logic_error&) {
    std::cout << "[перехват исключения]\n";
  }
}
```

```
int main () {

  std::thread threads[10];
  for (int i=0; i<10; ++i)                          // порождаем 10 потоков:
      threads[i] = std::thread(print_thread_id,i+1);
  for (auto& th : threads) th.join();               // ждем
  return 0;
}
```

Вывод

```
[перехват исключения]
2 is even
[перехват исключения]
4 is even
[перехват исключения]
6 is even
[перехват исключения]
8 is even
[перехват исключения]
10 is even
```

Пример создания `lock_guard` с тегом `adopt_lock`

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex, std::lock_guard, std::adopt_lock

std::mutex mtx;           // мьютекс для критической секции доступа к std::cout

void print_thread_id(int id) {
  mtx.lock();             // блокируем мьютекс
  std::lock_guard<std::mutex> lck(mtx, std::adopt_lock); // принимаем управлен.
  std::cout << "поток #" << id << '\n';                  // выводим
}

int main () {
  std::thread threads[10];
  for (int i=0; i<10; ++i)                              // порождаем 10 потоков
      threads[i] = std::thread(print_thread_id,i+1);
  for (auto& th : threads) th.join();                   // ждем их завершения
  return 0;
}
```

Возможный вывод (порядок строк может быть разным, но они никогда не пересекаются)

```
поток #1
поток #2
...
thread #10
```

std::adopt_lock_t — тип adopt_lock

```
struct adopt_lock_t {};
```

Это пустой класс, используемый как тип **adopt_lock**.

Передача **adopt_lock** конструктору **unique_lock** или **lock_guard** заставляет объект не блокировать объект мьютекса и вместо этого предполагает, что он уже заблокирован текущим потоком.

std::try_to_lock_t — тип try_to_lock

```
struct try_to_lock_t {};
```

Это пустой класс, используемый как тип **try_to_lock**.

Передача **try_to_lock** конструктору **unique_lock** заставляет его попытаться заблокировать объект мьютекса, вызывая член **try_lock()** вместо **lock()**.

std::defer_lock_t — тип defer_lock

```
struct defer_lock_t {};
```

Это пустой класс, используемый как тип **defer_lock**.

При передаче **defer_lock** конструктору **unique_lock** объект мьютекса при создании автоматически не блокируется; объект при этом инициализируется как не владеющий блокировкой.

std::adopt_lock — принять блокировку

```
constexpr adopt_lock_t adopt_lock {};
```

Значение, используемое как возможный аргумент конструктора **unique_lock** или **lock_guard**.

Объекты **unique_lock**, созданные с помощью **adopt_lock**, не блокируют объект мьютекса при создании, предполагая вместо этого, что он уже заблокирован текущим потоком.

std::try_to_lock — попытка захватить мьютекс

```
constexpr try_to_lock_t try_to_lock {};
```

Значение, используемое как возможный аргумент конструктора **unique_lock**.

Объекты **unique_lock**, созданные с помощью **try_to_lock**, пытаются заблокировать объект мьютекса, вызывая его член **try_lock** вместо его члена блокировки.

std::defer_lock — отложить блокировку

```
constexpr defer_lock_t defer_lock {};
```

Значение, используемое как возможный аргумент конструктора **unique_lock**.

Объекты **unique_lock**, созданные с помощью **defer_lock**, при создании не блокируют объект мьютекса автоматически, инициализируя их как не владеющие блокировкой.

Значения представляют собой константу времени компиляции, которая не несет состояния и используется просто для устранения неоднозначности между сигнатурами конструктора.

std::unique_lock — уникальная блокировка

```
template <class Mutex> class unique_lock;
```

Уникальная блокировка — это объект, который управляет мьютексом в обоих состояниях — какк заблокированом, так и в разблокированном.

При построении (или путем присваивания перемещением) объект типа **unique_lock** получает мьютекс и становятся ответственными за операции его блокировки и разблокировки.

Объект типа **unique_lock** поддерживает оба состояния — заблокирован и разблокирован.

Этот класс гарантирует разблокированное состояние мьютекса при уничтожении (даже если не вызывается явно). Поэтому он особенно полезен как объект с автоматической продолжительностью существования, поскольку он гарантирует, что мьютекс будет правильно разблокирован в случае возникновения исключения.

Однако, слудует обратить внимание, что объект типа **lock_guard** никоим образом не управляет временем жизни мьютекса — продолжительность существования объекта мьютекса должна простираться, по крайней мере, до разрушения объекта **lock_guard**, который его блокирует.

Параметры шаблона

Mutex — мьютекс-подобный тип.

Это должен быть базовый блокируемый тип, такой как мьютекс (см. **BasicLockable**).

Функции-члены

(constructor) — construct lock_guard (public member function)

(destructor) — destroy lock_guard (unlocking mutex) (public member function)

std::unique_lock::unique_lock — конструктор

(1) По умолчанию

```
unique_lock( ) noexcept;
```

Созданный объект не управляет объектом мьютекса.

(2) Блокирующий

```
explicit unique_lock(mutex_type& m);
```

Созданный объект управляет мьютексом **m** и блокирует его при создании, вызывая **m.lock( )**.

m — объект мьютекса, которым должен управлять объект типа **unique_lock**.

mutex_type — это параметр шаблона **unique_lock** (тип объекта мьютекса, которым предполагается управлять).

(3) С попыткой заблокировать

```
unique_lock(mutex_type& m, try_to_lock_t tag);
```

Созданный объект управляет мьютексом **m** и пытается заблокировать его без ожидания, вызывая **m.try_lock( )**.

tag — данный агрумент просто для того, чтобы выбрать конкретный конструктор (значения данного типа не имеют состояни — это пустые классы).

tag принимает одно из следующих значений:

(тега нет) — блокировка при создании путем автоматического вызова члена **lock()**.

adopt_lock — принять текущую блокировку (в предположении, что мьютекс уже заблокирован потоком).

adopt_lock_t — это тип объекта **adopt_lock**.

defer_lock — не блокировать и предполагать, что мьютекс еще не заблокировал текущим потоком.

try_to_lock_t, **defer_lock_t** и **adopt_lock_t** — типы объектов **try_to_lock**, **defer_lock** и **adopt_lock**, соответственно.

(4) С отложенной блокировкой

```
unique_lock(mutex_type& m, defer_lock_t tag) noexcept;
```

Созданный объект управляет мьютексом **m**, не блокируя его.

m должен быть мьютексным объектом, который в настоящее время не заблокирован создающим потоком.

(5) Принимающий блокировку

```
unique_lock(mutex_type& m, adopt_lock_t tag);
```

Созданный объект управляет мьютексом **m**, который является мьютексом, который в настоящее время заблокирован создающим потоком (объект в этом случае получает право собственности на блокировку).

(6) С ожиданием в течение некоторого времени

```
template <class Rep, class Period>
unique_lock (mutex_type& m,
             const chrono::duration<Rep, Period>& rel_time);
```

Созданный объект управляет мьютексом **m** и пытается заблокировать его в течение времени **rel_time**, вызывая **m.try_lock_for(rel_time)**.

rel_time — максимальный промежуток времени, в течение которого поток может блокироваться, ожидая получения блокировки мьютекса.

Если он исчерпан, объект инициализируется без блокировки.

duration — это объект, который представляет конкретное относительное время.

(7) **С ожиданием до определенного момента времени**

```
template <class Clock, class Duration>
unique_lock(mutex_type& m,
            const chrono::time_point<Clock, Duration>& abs_time);
```

Созданный объект управляет мьютексом **m** и пытается заблокировать его до момента времени **abs_time**, вызывая **m.try_lock_until(abs_time)**.

abs_time — момент времени, когда поток перестанет блокироваться в ожидании получения блокировки. Если момент времени достигнут без захвата мьютекса, объект инициализируется без блокировки.

time_point — это объект, представляющий конкретное абсолютное время.

(8) Копирования [deleted]

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

Объекты **unique_lock** не могут копироваться (конструктор копирования удален).

(9) Перемещения

```
unique_lock(unique_lock&& x);
```

Созданный объект получает мьютекс, управляемый unique_lock-объектом **x**, включая его текущее состояние владения.

unique_lock-объект **x** остается в том же состоянии, как если бы он был сконструирован по умолчанию (объект мьютекса после перемещения отсутствует).

x — другой объект типа **unique_lock object**.

Созданный объект типа **unique_lock** управляет состоянием полученного мьютекса, сохраняя на него ссылку и информацию о том, владеет ли он его блокировкой.

Объекты, построенные с помощью блокирующего конструктора (2) и конструктора, принимающего блокировку (5), всегда владеют блокировкой мьютекса.

Те объекты, которые созданы с помощью конструктора по умолчанию (1) и конструктора с отложенной блокировкой (4), никогда блокировкой не владеют.

Объекты, построенные с помощью конструкторов с попыткой заблокировать (3) и с ожиданием (6) и (7), владеют мьютексом, если попытка его захвата завершилась успешно.

Пример unique_lock

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex, std::unique_lock

std::mutex mtx;           // mutex for critical section

void print_block(int n, char c) {
  // критическая секция (доступ к std::cout завершается с временем жизни lck):
  std::unique_lock<std::mutex> lck(mtx);      // блокирующий конструктор
  for (int i = 0; i < n; ++i) { std::cout << c; }
  std::cout << '\n';
}

int main () {
  std::thread th1(print_block, 50,'*');
  std::thread th2(print_block, 50,'$');

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

  return 0;
}
```

Вывод (порядок линий может быть любым, но перемешивания сиволов не будет)

```
**************************************************
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

Пример создания объектов `unique_lock`

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex, std::lock, std::unique_lock
                          // std::adopt_lock, std::defer_lock
std::mutex foo, bar;

// функция потока
void task_a() {
  std::lock(foo, bar);         // одновременная блокировка (предотвращ. тупик)
  std::unique_lock<std::mutex> lck1(foo, std::adopt_lock); // с принятием
  std::unique_lock<std::mutex> lck2(bar, std::adopt_lock); // блокировки
  std::cout << " задача a\n";
  // foo и bar разблокируются автоматически при разрушении lck1 и lck2
}

// функция потока
void task_b () {
  // foo.lock(); bar.lock(); // заменяется на:
  std::unique_lock<std::mutex> lck1, lck2;
  lck1 = std::unique_lock<std::mutex>(bar, std::defer_lock); // move
  lck2 = std::unique_lock<std::mutex>(foo, std::defer_lock); //
  std::lock(lck1, lck2);     // одновременная блокировка (предотвращ. тупик)
  std::cout << "задача b\n";
  // foo и bar разблокируются автоматически при разрушении lck1 и lck2
}
```

```
int main () {

  std::thread th1(task_a);
  std::thread th2(task_b);

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

  return 0;
}
```

Возможный вывод

```
задача a
задача b
```

Открытые функции-члены блокировки/разблокировки

`lock()` – заблокировать мьютекс
`unlock` – разблокировать мьютекс
`try_lock()` – заблокировать мьютекс если не заблокирован
`try_lock_for()` – пытаться заблокировать в течение некоторого времени
`try_lock_until()` – пытаться заблокировать до определенного момента времени

`owns_lock()` – владеет ли объект блокировкой
`operator bool()` – возвращает , владеет ли он блокировкой
`mutex()` – получить мьютекс

std::unique_lock::operator bool — возвращает, владеет ли он блокировкой

```
explicit operator bool() const noexcept;
```

Возвращает, владеет ли объект блокировкой.

true, если управляемый мьютекс был заблокирован (или принят) объектом **unique_lock** и с тех пор не был разблокирован или освобожден.

false во всех остальных случаях.

Это псевдоним **unique_lock::owns_lock**.

std::unique_lock::owns_lock — владеет ли объект блокировкой

```
bool owns_lock() const noexcept;
```

Возвращает, владеет ли объект блокировкой.

true, если управляемый мьютекс был заблокирован (или принят) объектом **unique_lock** и с тех пор не был разблокирован или освобожден.

false во всех остальных случаях.

Это псевдоним **unique_lock::operator bool**.

Пример

```
#include <iostream>       // std::cout
#include <vector>         // std::vector
#include <thread>         // std::thread
#include <mutex>          // std::mutex, std::unique_lock, std::try_to_lock

std::mutex mtx;           // mutex for critical section

void print_star() { // потоковая функция печати символа
  std::unique_lock<std::mutex> lck(mtx, std::try_to_lock);
  // печатает '*' если удалось заблокировать mtx, 'x' в противном случае
- if (lck.owns_lock())  // или функция
- if (lck)              // или оператор bool()
    std::cout << '*';
  else
    std::cout << 'x';
}

int main () {
  std::vector<std::thread> threads;
  for (int i = 0; i < 500; ++i)
      threads.emplace_back(print_star);
  for (auto& x: threads) x.join();

}
```

Вывод (что-то типа такого)

```
**********************************x*****x*******************x*******************
*****************x**x*x*x***x***x************************x***************x******
**************************************************************************
**************************************************************************
**************************************************************x***********
********x*****************************************************************
************x***********x**
```

std::unique_lock::mutex — получить мьютекс

```
mutex_type* mutex( ) const noexcept;
```

Возвращает указатель на управляемый объект мьютекса.

Обратите внимание, что **`unique_lock`** не снимает владение управляемым мьютексом — если объект владеет блокировкой мьютекса, он по-прежнему отвечает за его разблокировку в какой-то момент (например, когда он уничтожается).

Пример unique_lock::mutex()

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex, std::unique_lock, std::defer_lock

class MyMutex : public std::mutex {
  int _id;
public:
  MyMutex(int id) : _id(id) {}
  int id() {return _id;}
};

MyMutex mtx(101);

void print_ids(int id) {
  std::unique_lock<MyMutex> lck(mtx);
  std::cout << "thread #" << id << " locked mutex " << lck.mutex()->id();
  std::cout <<  '\n';
}

int main () {
  std::thread threads[10];
  for (int i=0; i<10; ++i)                        // spawn 10 threads
    threads[i] = std::thread(print_ids,i+1);
  for (auto& th : threads) th.join();
  return 0;
}
```

Вывод

```
thread #1 locked mutex 101
thread #2 locked mutex 101
thread #4 locked mutex 101
thread #3 locked mutex 101
thread #6 locked mutex 101
thread #5 locked mutex 101
thread #7 locked mutex 101
thread #8 locked mutex 101
thread #10 locked mutex 101
thread #9 locked mutex 101
```

std::unique_lock::lock — заблокировать мьютекс

```
void lock();
```

Вызывает функцию-член **lock()** для управляемого мьютекса.

Вызов **lock()** для мьютекса, который уже был заблокирован другими потоками, заставляет текущий поток блокироваться (ждать), пока он не сможет завладеть для него блокировкой.

Когда функция возвращается, объект владеет блокировкой мьютекса.

Если вызов блокировки завершается неудачно, генерируется исключение **system_error**.

std::unique_lock::unlock — разблокировать мьютекс

```
void unlock();
```

Вызывает член **unlock()** для управляемого мьютекса и устанавливает для состояния владения значение **false**.

Если до вызова состояние владения было **false**, функция выдает исключение **system_error** с **operation_not_permitted** в качестве условия ошибки.

Пример lock/unlock

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex, std::unique_lock, std::defer_lock

std::mutex mtx;           // mutex for critical section

void print_thread_id (int id) {
  std::unique_lock<std::mutex> lck(mtx, std::defer_lock);
  // critical section (exclusive access to std::cout signaled by locking lck):
  lck.lock();
  std::cout << "thread #" << id << '\n';
  lck.unlock();
}

int main () {

  std::thread threads[10];
  // spawn 10 threads:
  for (int i=0; i<10; ++i)
      threads[i] = std::thread(print_thread_id,i+1);
  for (auto& th : threads) th.join();

  return 0;
}
```

Вывод (порядок может быть любой)

```
thread #1
thread #2
...
thread #10
```

std::unique_lock::try_lock — заблокировать мьютекс если не заблокирован

```
bool try_lock();
```

Вызывает функцию-член **try_lock()** для управляемого мьютекса и использует возвращаемое значение для установки состояния владения.

Если состояние владения уже истинно до вызова или если объект в настоящее время не управляет мьютексом, функция выдает исключение **system_error**.

Пример unique_lock::try_lock()

```
#include <iostream>       // std::cout
#include <vector>         // std::vector
#include <thread>         // std::thread
#include <mutex>          // std::mutex, std::unique_lock, std::defer_lock

std::mutex mtx;           // mutex for critical section

void print_star() {
  std::unique_lock<std::mutex> lck(mtx, std::defer_lock);
  // print '*' if successfully locked, 'x' otherwise:
  if (lck.try_lock())
    std::cout << '*';
  else
    std::cout << 'x';
}
```

```
int main () {

  std::vector<std::thread> threads;
  for (int i = 0; i < 500; ++i)
    threads.emplace_back(print_star);
  for (auto& x: threads) x.join();

  return 0;
}
```

Вывод типа такого

```
**********************x**********************x*x***x***x*x*x**x*
*x*********x*****************************************************
**************x*x*x*x*****************************************************
**********x********x*********x*****************************************************
**************************************x*x*x**x*x*x*x*x*x****************
*******x*****************************************************************
**********************x****
```

std::unique_lock::try_lock_for — пытаться заблокировать в течение промежутка времени

```
template <class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
```

Вызывает функцию-член **try_lock_for()** для управляемого мьютекса и использует возвращаемое значение для установки состояния владения.

Если состояние владения уже было истинно до вызова или если объект в настоящее время не управляет мьютексом, функция выдает исключение **system_error**.

rel_time — максимальный промежуток времени, в течение которого поток может блокироваться, ожидая получения блокировки мьютекса.

duration — это объект, который представляет конкретное относительное время.

Возвращает

true, если функция успешно блокирует управляемый мьютекс.

false в противном случае

Пример unique_lock::try_lock_for()

```
#include <iostream>       // std::cout
#include <chrono>         // std::chrono::milliseconds
#include <thread>         // std::thread
#include <mutex>          // std::timed_mutex, std::unique_lock,
std::defer_lock

std::timed_mutex mtx;

void fireworks () {
  std::unique_lock<std::timed_mutex> lck(mtx,std::defer_lock);
  while (!lck.try_lock_for(std::chrono::milliseconds(200))) {
    std::cout << "-";
  }
  // got a lock! - wait for 1s, then this thread prints "*"
  std::this_thread::sleep_for(std::chrono::milliseconds(1000));
  std::cout << "*\n";
}

int main () {
  std::thread threads[10];
  for (int i=0; i<10; ++i)   // spawn 10 threads
    threads[i] = std::thread(fireworks);
  for (auto& th : threads) th.join();
  return 0;
}
```

Вывод похож на следующий

```
_____________________________________*
_____________________________________*
_____________________________________*
_____________________________*
_________________________*
_____________________*
_______________*
__________*
_____*
*
```

std::unique_lock::try_lock_until — пытаться заблокировать до определенного момента времени

```
template <class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Вызывает функцию-член **try_lock_until()** для управляемого мьютекса и использует возвращаемое значение для установки состояния владения.

Если состояние владения уже истинно до вызова или если объект в настоящее время не управляет объектом мьютекса, функция выдает исключение **system_error**.

abs_time — момент времени, когда поток перестанет блокироваться, отказавшись от получения блокировки.

time_point — это объект, представляющий конкретное абсолютное время.

Возвращает

true, если функция успешно блокирует управляемый мьютекс.

false в противном случае

Деструктор

```
~unique_lock();
```

Уничтожает объект **`unique_lock`**.

Если объект в настоящее время владеет блокировкой управляемого мьютекса, перед уничтожением объекта вызывается его функция-член разблокировки.

Сам объект управляемого мьютекса не уничтожается.

std::unique_lock::operator= — оператор перемещения управляемого мьютекса

(1) Оператор присваивания перемещением

```
unique_lock& operator= (unique_lock&& x) noexcept;
```

Заменяет управляемый объект мьютекса на мьютекс из **x**, включая его состояние владения.

Если объект владел блокировкой своего управляемого мьютекса до вызова, перед заменой вызывается его функция-член **unlock()**.

x остается в том же состоянии, как если бы он был сконструирован по умолчанию (без мьютекса).

(2) Оператор присваивания копированием [deleted]

```
unique_lock& operator= (const unique_lock&) = delete;
```

Объекты **unique_lock** не могут копироваться.

Пример `unique_lock::operator=`

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
#include <mutex>          // std::mutex, std::unique_lock

std::mutex mtx;           // mutex for critical section

void print_fifty(char c) {
  std::unique_lock<std::mutex> lck;         // по умолчанию
  lck = std::unique_lock<std::mutex>(mtx);  // перемещение
  for (int i = 0; i < 50; ++i) { std::cout << c; }
  std::cout << '\n';
}

int main () {

  std::thread th1(print_fifty,'*');
  std::thread th2(print_fifty,'$');

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

  return 0;
}
```

std::unique_lock::release — освободить мьютекс

```
mutex_type* release( ) noexcept;
```

Возвращает указатель на управляемый объект мьютекса, освобождая владение им текущего объекта **unique_lock**.

После вызова объект **unique_lock** больше не управляет каким-либо объектом мьютекса (т.е. он остается в том же состоянии, что и объект, созданный по умолчанию).

Следует обратить внимание, что эта функция не блокирует и не разблокирует возвращаемый объект мьютекса.

Пример unique_lock::release

```
#include <iostream>       // std::cout
#include <vector>         // std::vector
#include <thread>         // std::thread
#include <mutex>          // std::mutex, std::unique_lock

std::mutex mtx;
int count = 0;

void print_count_and_unlock(std::mutex* p_mtx) {
  std::cout << "count: " << count << '\n';
  p_mtx->unlock( );
}
```

```
void task() {
  std::unique_lock<std::mutex> lck(mtx);
  ++count;
  print_count_and_unlock(lck.release());
}

int main () {

  std::vector<std::thread> threads;
  for (int i = 0; i < 10; ++i) {
    threads.emplace_back(task);
  }
  for (auto& x: threads) x.join();

  return 0;
}
```

Вывод

```
count: 1
count: 2
...
count: 10
```

std::call_once — вызов функции один раз

```
template <class Fn, class... Args>
void call_once(once_flag& flag, Fn&& fn, Args&&... args);
```

В мультипоточных приложениях бывает, что некоторая задача должна быть выполнена только один раз. В таком случае помогает функция **`std::call_once()`**.

Вызывает функцию **`fn`**, передавая в качестве аргументов **`args`**, но только если другой поток уже не выполнил (или в настоящее время не выполняет) вызов **`call_once()`** с тем же флагом.

Если другой поток уже активно выполняет вызов **`call_once()`** с тем же флагом, он вызывает пассивное выполнение — пассивные выполнения не вызывают **`fn`**, но не возвращаются до тех пор, пока не вернется активное выполнение, и все видимые побочные эффекты не будут в этой точке синхронизированы для всех одновременных вызовов этой функции с одним и тем же флагом.

Если активный вызов **`call_once()`** завершается выдачей исключения (которое передается в вызывающий поток) и существуют пассивные исполнения, выбирается одно из этих пассивных выполнений и вызывается как новый активный вызов вместо рухнувшего.

Следует обратить внимание, что после возврата активного выполнения также возвращаются, не становясь активными выполнениями, и все текущие пассивные выполнения и будущие вызовы **`call_once()`** (с тем же флагом).

Активное выполнение использует «распадающиеся» копии ссылок lvalue или rvalue для **`fn`** и **`args`**. Значение, возвращаемое **`fn`**, игнорируется.

flag — объект, используемый функцией для отслеживания состояния вызовов.

Использование одного и того же объекта для вызовов в разных потоках приводит к тому, что в случае одновременных вызовов из разных потоков реализован будет только один — активный вызов.

C++11

Если флаг имеет недопустимое состояние, функция генерирует исключение **system_error** с ошибкой **invalid_argument**.

C++14

Если флаг имеет недопустимое состояние, вызов вызывает неопределенное поведение.

fn — указатель на функцию, указатель на функцию-член или любой тип функционального объекта, который можно перемещать (т.е. объект, класс которого определяет перегрузкой **operator ()**).

args... — аргументы, передаваемые **fn()**. Их типы должны быть перемещаемыми.

Если **fn** является указателем на функцию-член, первым аргументом должен быть объект, для которого этот член определен (ссылка, или указатель на него).

Пример call_once()

```
#include <iostream>       // std::cout
#include <thread>         // std::thread, std::this_thread::sleep_for
#include <chrono>         // std::chrono::milliseconds
#include <mutex>          // std::call_once, std::once_flag

int winner;
void set_winner(int x) { winner = x; }
std::once_flag winner_flag;

void wait_1000ms (int id) {
  for (int i = 0; i < 1000; ++i) // считаем до 1000, ожидая 1 мс между инкрем.
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
  std::call_once (winner_flag,set_winner, id); // претендуем на победителя
}

int main () {
  std::thread threads[10];
  for (int i = 0; i < 10; ++i)                  // порождаем 10 потоков
    threads[i] = std::thread(wait_1000ms, i + 1);

  std::cout << "ждем покак первая нить из 10 не досчитает до 1000...\n";
  for (auto& th : threads) th.join();
  std::cout << "победившая нить: " << winner << '\n';

  return 0;
}
```

Вывод

```
ждем покак первая нить из 10 не досчитает до 1000...
победившая нить: 7
```

BasicLockable — базовый блокируемый тип

BasicLockable <-- Lockable <-- TimedLockable

Базовый блокируемый тип — это тип, который поддерживает **`lock()`** и **`unlock()`**.
Стандартная библиотека определяет следующие типы **`BasicLockable`**:

```
mutex
recursive_mutex
timed_mutex
recursive_timed_mutex
unique_lock
```

Требования

Объект **`m`** относится к типу **`BasicLockable`**, если следующие выражения имеют правильное поведение:

`m.lock()`

Действие — блокировка выполнения (ожидание) агента выполнения до тех пор, пока не будет получена блокировка **`m`**. Если генерируется исключение, то **`m`** не должен блокироваться.

`m.unlock()`

Требует — текущий агент выполнения должен удерживать **`m`**.
Действие — снимает блокировку **`m`**, удерживаемую текущим агентом выполнения.

Исключения — нет

Lockable — блокируемый тип (тип мьютекса)

BasicLockable <-- **Lockable** <-- TimedLockable

Блокируемый тип, известный как тип мьютекса, котороый поддерживает **`try_lock()`**.

Помимо поддержки **`m.lock()`** и **`m.unlock()`** согласно требованиям к BasicLockable поддерживается

`m.try_lock()`

Действие — пытается получить блокировку для текущего агента выполнения без ожидания. Если генерируется исключение, то m не должен блокироваться.

Тип возврата — **`bool`**

Возвращает — **`true`**, если блокировка была получена, иначе **`false`**.

TimedLockable — Тип с ограниченным ожиданием блокировки

BasicLockable <-- **Lockable** <-- TimedLockable

Это Lockable тип, который поддерживает **`try_lock_for()`** и **`try_lock_until()`**.
К данному типу принадлежат мьютексы:
timed_mutex
recursive_timed_mutex

```
m.try_lock_for(rel_time)
```

Действие — пытается получить блокировку для текущего агента выполнения в течение относительного тайм-аута, указанного **`rel_time`**.

Функция не должна возвращаться в течение тайм-аута, указанного в **`rel_time`**, если только она не получила блокировку **`m`** для текущего агента выполнения.

Если генерируется исключение, то для текущего агента выполнения блокировка не должна быть получена.

Тип возврата — **`bool`**

Возвращает — **`true`**, если блокировка **`m`** была получена, иначе **`false`**.

```
m.try_lock_until(abs_time)
```

Действие — пытается получить блокировку для текущего агента выполнения до абсолютного тайм-аута, указанного в **`abs_time`**.

Функция не должна возвращаться до истечения времени ожидания, указанного в **`abs_time`**, если только она не получила блокировку **`m`** для текущего агента выполнения.

Если генерируется исключение, то для текущего агента выполнения блокировка **`m`** не должна быть получена.

Тип возврата — **`bool`**

Возвращает — **`true`**, если блокировка **`m`** была получена, иначе **`false`**.

`m` — объект типа **`TimedLockable`**
`rel_time` — значение экземпляра шаблона **`duration`**
`abs_time` — значение экземпляра шаблона **`time_point`**

Функции для одновременной блокировки нескольких мьютексов:
try_lock — шаблон функции для попытки заблокировать несколько мьютексов
lock — шаблон функции для блокировки несколько мьютексов
Функции для предотвращения одновременного выполнения определенной функции:
call_once — шаблон функции, которая не будет вызывана одновременно несколькими вызовами.