Оглавление

[Потоки thread 2](#_Toc142399418)

[th.detach 3](#_Toc142399419)

[th.join 4](#_Toc142399420)

[Потоки с параметрами 6](#_Toc142399421)

[Возврат результата из потока (по ссылке) 7](#_Toc142399422)

[Передача в поток методов класса 8](#_Toc142399423)

[Mutex (синхронизация) 9](#_Toc142399424)

[Команды,функции, методы 11](#_Toc142399425)

# Потоки thread

При запуске программы создается процесс, который является абстрактной оболочкой, которая разграничивает ресурсы программы от других программ.

В процессе должен быть хотя бы 1 поток.

В потоке происходит выполнение логики.

Поток определяет последовательность исполнения кода.

С помощью создания доп.потоков и передачи в них части логики нашей программы можно обеспечить парралельное выполнение программы (т.н. ассинхронное).

Важно учесть, чтоб были подходящие аппаратные ресурсы. На одноядерном процессоре мы лишь потеряем производительность.

Если задача простая и легкая, то не имеет смысла выделять под неё отдельный поток

Библиотека **thread** используется для работы с потоками

Процессы находятся в пространстве имен this\_thread

Библиотека **chrono** используется для работы со временем

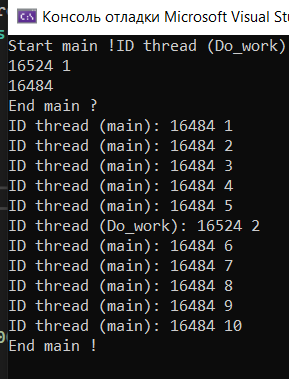
Время находится в пространстве имен chrono

Чтобы запустить задачу параллельно надо передать в конструктор потока имя функции (не вызвать, а передать имя, т.к. имя – это указатель на функцию), которая будет выполняться параллельно.



Однако они при выходе из зоны видимости (в нашем случае из фунции main), то вызовется дестуктор и программа попытается закрыться из другого потока, из-за чего вызовется исключение. Когда мы создаем потом, то обязаны определить его поведение. Это делается следующим образом:

th.detach**()** – разрывает связь между объектом и тем потоком, который мы начали выполнять. С помощью метода detach мы сообщаем, потоку, что он может выполняться самостоятельно, но, если у нас закончил работу основной проект, то вне зависимости от степени завершенности нашего потока, он будет принудительно завершен

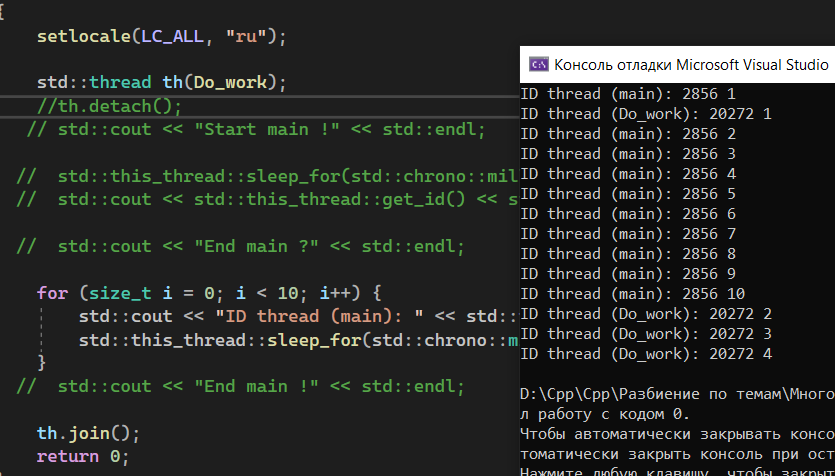


Поток Do\_work должен был вывестись 4 раза, но вывелся всего 2. Это произошло из-за того, что время выполнения процесса Do\_work 1.5 секунды на итерацию, а время выполнения мейн-потока 0.2 секунды на итерацию. Оставшиеся операции потока Do\_work пропали по завершению мейн-потока.

Для выполнение задач «на фоне»

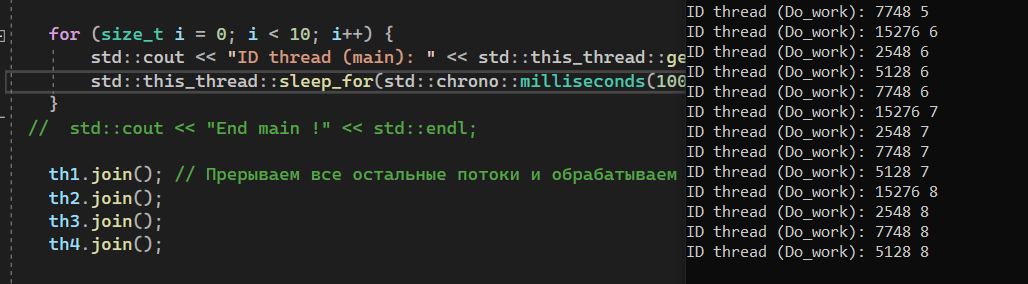
th.join**()** – позволяет дождаться выполнения той задачи, которую мы загнали в поток.

Нужно вызывать в том месте кода, где мы хотим дождаться работы выполнения потока. Если вызвать его сразу после создания потока, то это не будет ничем отличаться от последовательно-работающей программы.



Вызвав в конце фунцию join(thread) мы не допустим того, чтобы наш поток «потерялся», как в случае с detach, а дождемся конца его работы.

Используется, когда необходимы определенные данные.



В случае с двумя и более join (на 2 и более потоков соответственно), то Join`ы будут выполняться также параллельно. На скриншоте видно, что 4 джойна одновременно выполняются.

**Примеры некоторых задач**

* **Получение** **id** текущего потока (нужны сиауты)

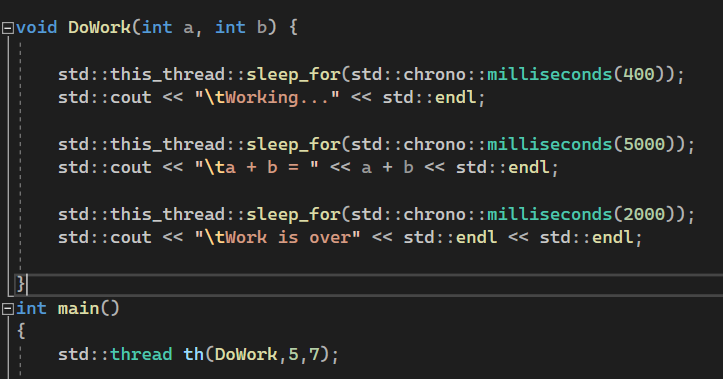


* Эмуляция задержки в 1 сек



# Потоки с параметрами

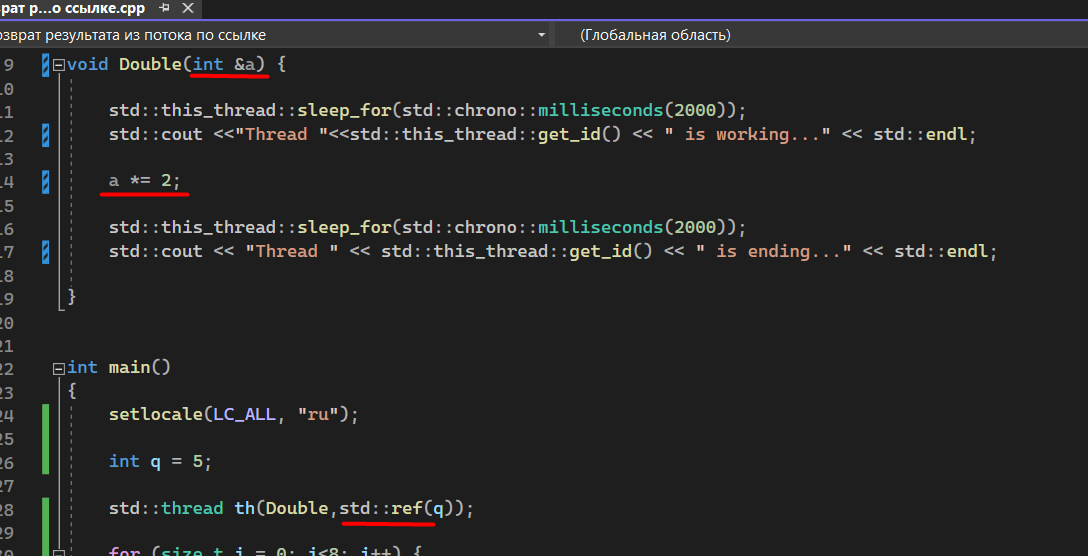
Параметры в поток передаются через запятую



Количество и тип данных параметров в потоке должны соответствовать количеству и типу данных в функции (логично)

# Возврат результата из потока (по ссылке)

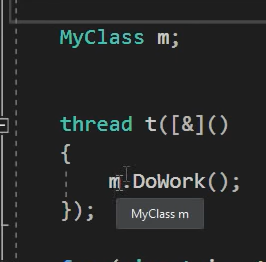
* Функция должна принимать значение по ссылке
* При создании потока надо переменную, которую будем возвращать, обернуть в std::ref(var)
* Надо дожидаться результатов, т.е., если нам нужны данные, возвращаемые из потока, то надо сперва дождаться завершения потока th.join(), а уже потом работать с ними. Иначе поток может не успеть и данные будут неверными или их вообще может не быть



# Передача в поток методов класса

* **Первый вариант**

Через лямбда-функции:



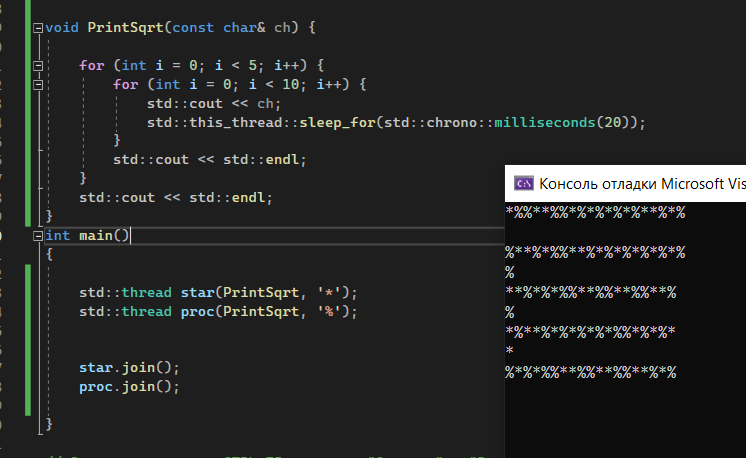
* **Второй вариант**

Через явное указание

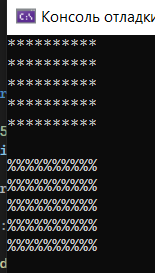
# Mutex (синхронизация)

Mutex – не разрешает другим потокам обращаться к ресурсу и работать с ним, если этот ресурс уже занят другим потоком.

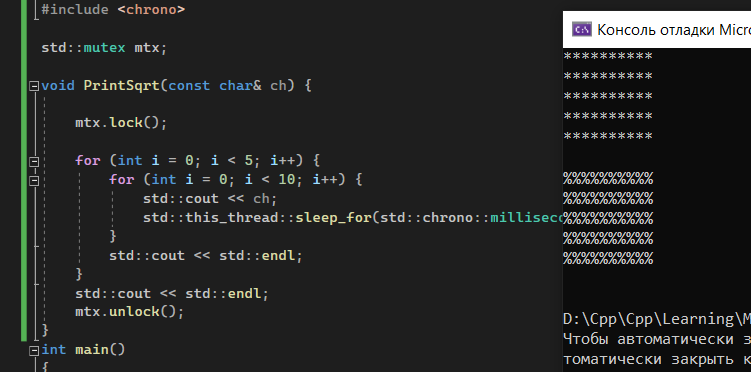
Пример ошибки:



Что должно было вывести



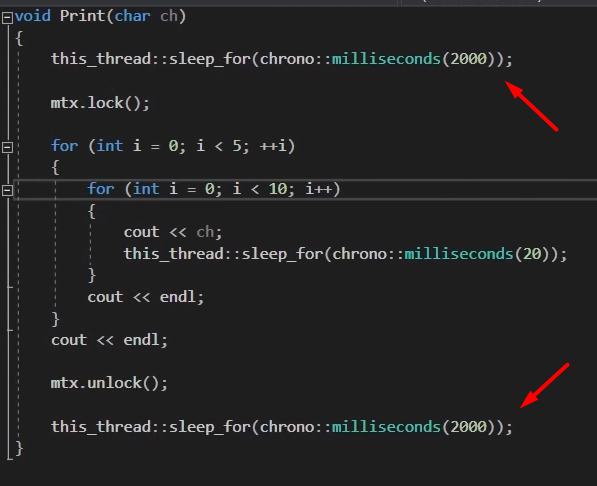
Пример использования мутеха



Если сравнивать время «неправильного» выполнения программы, когда все вперемешку было, и правильного – когда последовательно, то в первом случае время в 2 раза меньше: 1 секунда против 2

Делаем выводы, mutex – это «узкое горлышко бутылки», которое надо использовать с осторожностью, которым надо «мутить» только опасные места.

Следующий код показывает преимущство многопотока в отличии от обычных функций последовательных и использование мутеха



Тут мы мутим только тот момент, где выводятся данные, оставляя остальную часть кода незамученной, т.е. два слипа могут выполняться одновременно.

Последовательный код выполнит это за 10 секунд, а многопоток за 6. Если же просто все смешать и не мутить ничего, то выйдет 5 секунд.

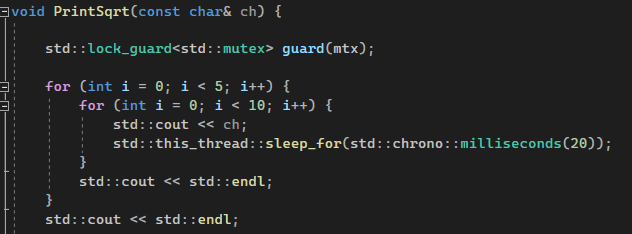
Команды

* std::mutex mtx – создание экземляра класса mutex
* mtx.lock() – лочит поток
* mtx.unlock() – анлочит поток

# Lock\_guard Mutex

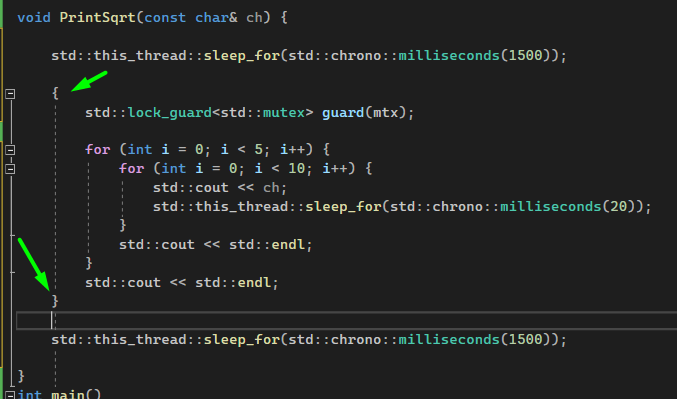
Захватить mutex в конструкторе и освободить mutex в деструкторе

Пример использования



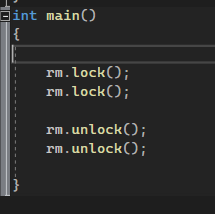
Однако, если нам надо только часть функции выполнять синхроно, то стоит прибегнуть к области видимо, а именно к фигурным скобкам.

Пример:



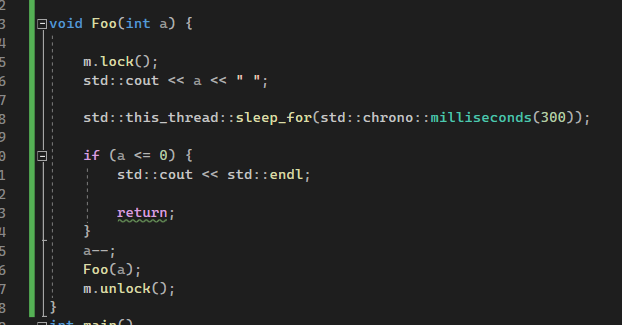
# Рекурсивный mutex

В отличии от обычного mutex, позволяет захватывать поток (th.lock()) сколько угодно раз.

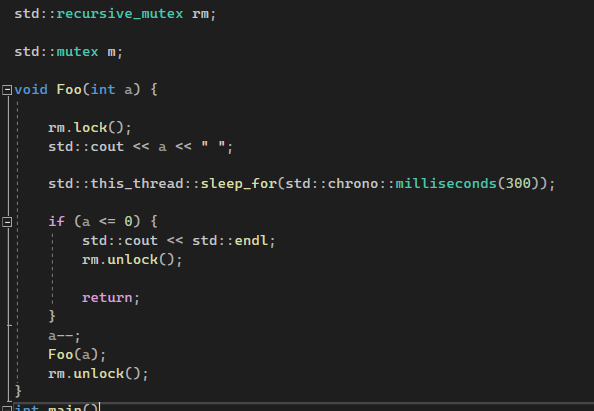


Как и где использовать? В рекурсивных функциях.

Здесь мы рекурсивно выводим числа от 10 до 0, но сталкиваемся с проблемой, что мы еще не освободив поток, пытаемся его захватить



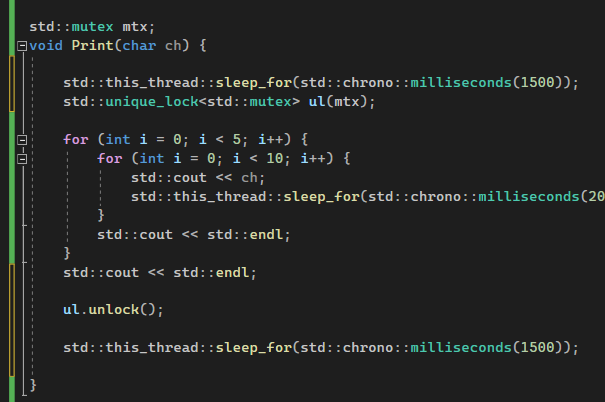
Пример правильной реализации



# Unique\_lock Mutex

Работает по аналогии с lock\_guard, но имеет дополнительные методы

Пример реализации (вручную указан анлок в конце)

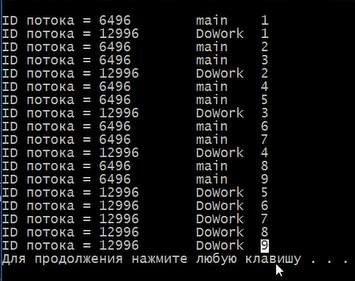


# Команды,функции, методы

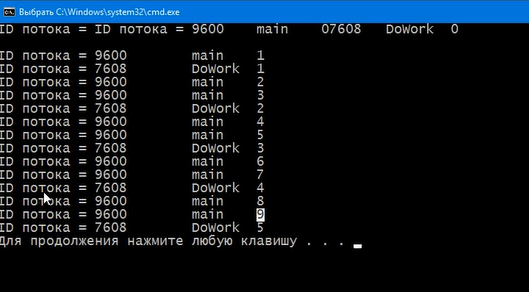
* Заголовочный файл - #include<thread>
* Запуск потока – std::thread t\_name (вызываемый объект, аргументы)
  + аргументы передаются через std::ref(arg)



* Жизненный цикл потока (обязательно определять)
  + t.join() – блокирует всё и ждет завершения потока (ставить в конце)



* + t.detach() – когда завершается родительский поток, тогда останавливается и наш (обе итерации по 10 элементов, но вывелось 5 в DoWork)



* std::this\_thread::sleep\_for(time) – поток спит определенное время
* std::this\_thread::get\_id() – получение ID потока