For Exam

38 Функции для работы с файловой системой. std::filesystem: filesystem::path, directory_iterator. Функции exists, is_regular_file, is_directory, is_symlink. Примеры работы с std::filesystem. -

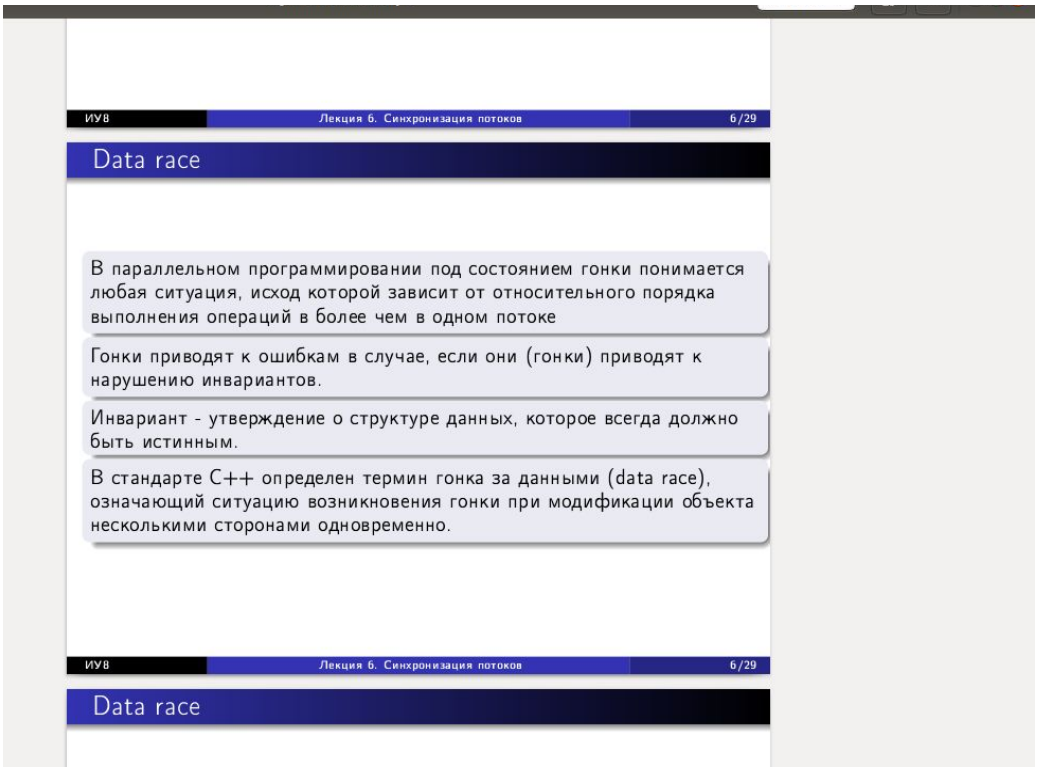
A: Появились в 17 стандарте, аналог boost

Пример работы

```
#include <fstream>
#include <iostream>
#include <filesystem>
namespace fs = std::filesystem;

int main()
{
   fs::create_directories("sandbox/a/b");
   std::ofstream("sandbox/file1.txt");
   std::ofstream("sandbox/file2.txt");
   for(auto& p: fs::directory_iterator("sandbox"))
      std::cout << p.path() << '\n';
   fs::remove_all("sandbox");
```

37 Синхронизация потоков. Состояние гонок. Шаблон producer-consumer.



Примеры.

Два способа Один поток может видеть и изменять промежуточные состояния(mutex)

2 - Последовательность неделимых изменений(атомарные переменные) - программирование без блокировок

https://github.com/bmstu-iu8-cpp/cpp-upper-intermediate/tree/master/cpp17/lectures/05_synchronization/sources

```
...
template<typename T> struct atomic<T*>; // частичная специализация
                                        // для указателей
```

Высокоуровневые операции над атомарными объектами перечислены в табл. 18.11. По мере возможности они непосредственно переводятся в соответствующие инструкции центрального процессора. Столбец ***triv*** обозначает операции, предусмотренные для типа `std::atomic<bool>` и других элементарных типов; столбец ***int type*** означает операции для класса `std::atomic<>`, если используется целочисленный тип, а столбец ***ptr type*** обозначает операции, предусмотренные для класса `std::atomic<>`, если используется тип указателей.

Сделаем несколько замечаний, касающихся этой таблицы.

- В основном операции возвращают копии, а не ссылки.
- Конструктор по умолчанию не полностью инициализирует переменную/объект. Единственной разрешенной операцией после выполнения конструктора по умолчанию является функция `atomic_init()`, инициализирующая объект (см. раздел 18.7.1).
- Конструктор для значения соответствующего типа не является атомарным.
- Все функции, за исключением конструкторов, перегружены для типов с модификатором `volatile` и без него.

Таблица 18.11. Высокоуровневые операции над атомарными типами

Операция	*triv*	*int type*	*ptr type*	Описание
atomic `a=`*val*	Да	Да	Да	Инициализирует `a` значением *val* (не атомарная операция)
atomic `a;` `atomic_init(&a,`*val*`)`	Да	Да	Да	То же самое (без `atomic_init()`, объект `a` не инициализируется)

Окончание табл. 18.11

Операция	*triv*	*int type*	*ptr type*	Описание
`a.is_lock_free()`	Да	Да	Да	`true`, если тип не использует блокировки внутри
`a.store(`*val*`)`	Да	Да	Да	Присваивает *val* (возвращает `void`)
`a.load()`	Да	Да	Да	Возвращает копию значения `a`
`a.exchange(`*val*`)`	Да	Да	Да	Присваивает *val* и возвращает копию старого значения `a`
`a.compare_exchange_strong(`*exp*`,` *des*`)`	Да	Да	Да	Операция CAS (сравнение с обменом — compare and swap)

34 Атомарные операции. Классы std::atomic, std::atomic_flag. Примеры работы с std::atomic

Рассмотрим полный пример с атомарными типами:

```
// concurrency/atomics1.cpp

#include <atomic> // для атомарных типов
#include <future> // для async() и фьючерсов
#include <thread> // для потока this_thread
#include <chrono> // для интервалов времени
#include <iostream>

long data;
std::atomic<bool> readyFlag(false);

void provider ()
{
   // после чтения символа
```



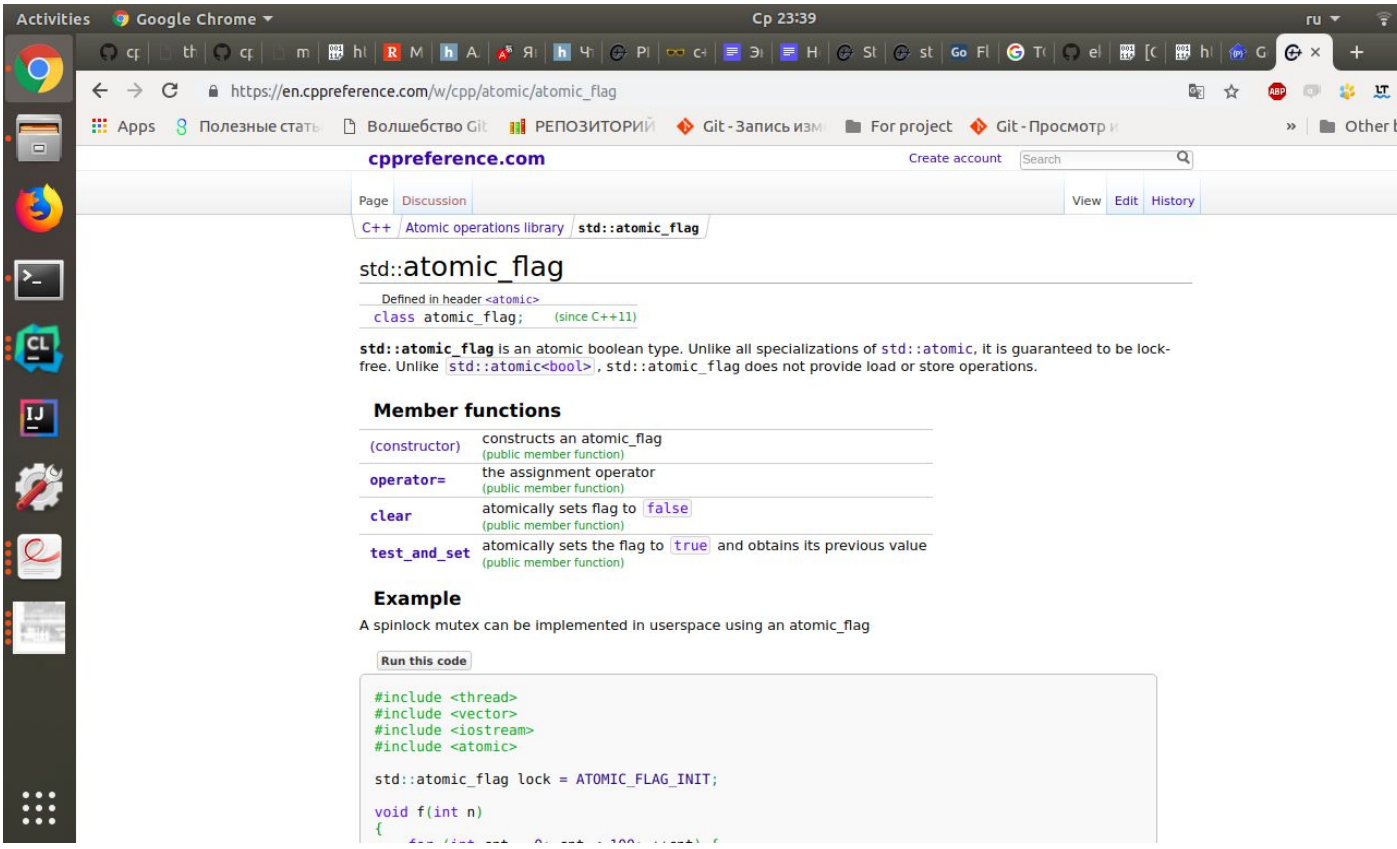
```
   std::cout << "<return>" << std::endl;
   std::cin.get();

   // подготавливаем данные
   data = 42;

   // и сигнал готовности
   readyFlag.store(true);
}

void consumer ()
{
   // ожидаем сигнала готовности и выполняем другие операции
   while (!readyFlag.load()) {
      std::cout.put('.').flush();
      std::this_thread::sleep_for(std::chrono::milliseconds(500));
   }
   // обрабатываем поступившие данные
   std::cout << "\nvalue : " << data << std::endl;
}

int main()
{
   // запускаем потоки provider и consumer
   auto p = std::async(std::launch::async,provider);
   auto c = std::async(std::launch::async,consumer);
}
```



35 - Перегрузка new и delete. Аллокаторы, системы управления памятью. TCMalloc

- смотри лекции
- Пример Аллокатора

```
// alloc/myalloc11.hpp

#include <cstddef>        // для типа size_t

template <typename T>
class MyAlloc {
   public:
     // определения типов
     typedef T value_type;

     // конструкторы
     // - ничего не делают, потому что распределитель памяти не имеет состояния
     MyAlloc () noexcept {
     }
     template <typename U>
     MyAlloc (const MyAlloc<U>&) noexcept {
         // нет состояния для копирования
     }

     // выделяем память для num элементов типа T, но не инициализируем ее
     T* allocate (std::size_t num) {
         // выделяем память с помощью глобальной операции new
         return static_cast<T*>(::operator new(num*sizeof(T)));
     }

     // освобождаем память, на которую ссылается указатель p,
     // от удаленных элементов
     void deallocate (T* p, std::size_t num) {

     // освобождаем память с помощью глобальной операции delete
     ::operator delete(p);
   }
};

// возвращаем признак того, что все специализации данного
// распределителя памяти являются эквивалентными
template <typename T1, typename T2>
bool operator== (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) noexcept {
   return true;
}
template <typename T1, typename T2>
bool operator!= (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) noexcept {
   return false;
}
```

33

Управление потоками. Состояния гонок. Класс std::future, функция std::async. Пример работы с std::future и std::async.

Смотри пример /home/netta/Downloads/AL_EXAM/Code/async3 (copy).cpp

Окончание табл. 18.1

Операция	Описание
f.~***future***()	Разрушает состояние и объект `*this`
f = *rv*	Перемещающее присваивание; разрушает старое состояние объекта *f*, присваивает состояние объекта *rv* и делает его некорректным
f.`valid()`	Возвращает `true`, если объект находится в корректном состоянии, так что можно вызвать следующие функции-члены
f.`get()`	Блокирует поток, пока выполняется фоновая операция (инициируя синхронный запуск отложенной ассоциированной функциональной сущности), возвращает результат (если он существует) или генерирует любое возникающее исключение и делает состояние некорректным
f.`wait()`	Блокирует поток, пока выполняется фоновая операция (вынуждая синхронный запуск отложенной ассоциированной функциональной сущности)
f.`wait_for(`*dur*`)`	Блокирует поток на период, заданный аргументом *dur*, или на период выполнения фоновой операции (запуск отложенного потока не вынуждается)
f.`wait_until(`*tp*`)`	Блокирует поток до момента *tp* или на период выполнения фоновой операции (запуск отложенного потока не вынуждается)
f.`share()`	Возвращает объект класса `shared_future` с текущим состоянием и делает некорректным состояние объекта *f*

31Синхронизация потоков. Пул потоков. Класс std::condition_variable. Примеры работы с std::condition_variable.

См лекцию и /home/netta/Downloads/AL_EXAM/Code/condition_varible

30 Синхронизация потоков. Состояние гонок. Потокобезопасные структуры данных с блокировками. - /home/netta/Downloads/AL_EXAM/Code/thread_queue

29 Синхронизация потоков. Состояние гонок. Классы std::recursive_mutex, boost::shared_mutex, std::unique_lock. Функция std::lock. Пример использования std::unique_lock.

См /home/netta/Downloads/AL_EXAM/Code/condition_varible

std::lock(m1, ..mn) - блокирует все mutex

recursive_mutex

снова заблокировать dbMutex, пока его блокировка не станет доступной, а это никогда не произойдет, так как функция `createTableAndInsertData()` блокирует поток до тех пор, пока не будет вызвана функция `createTable()`.

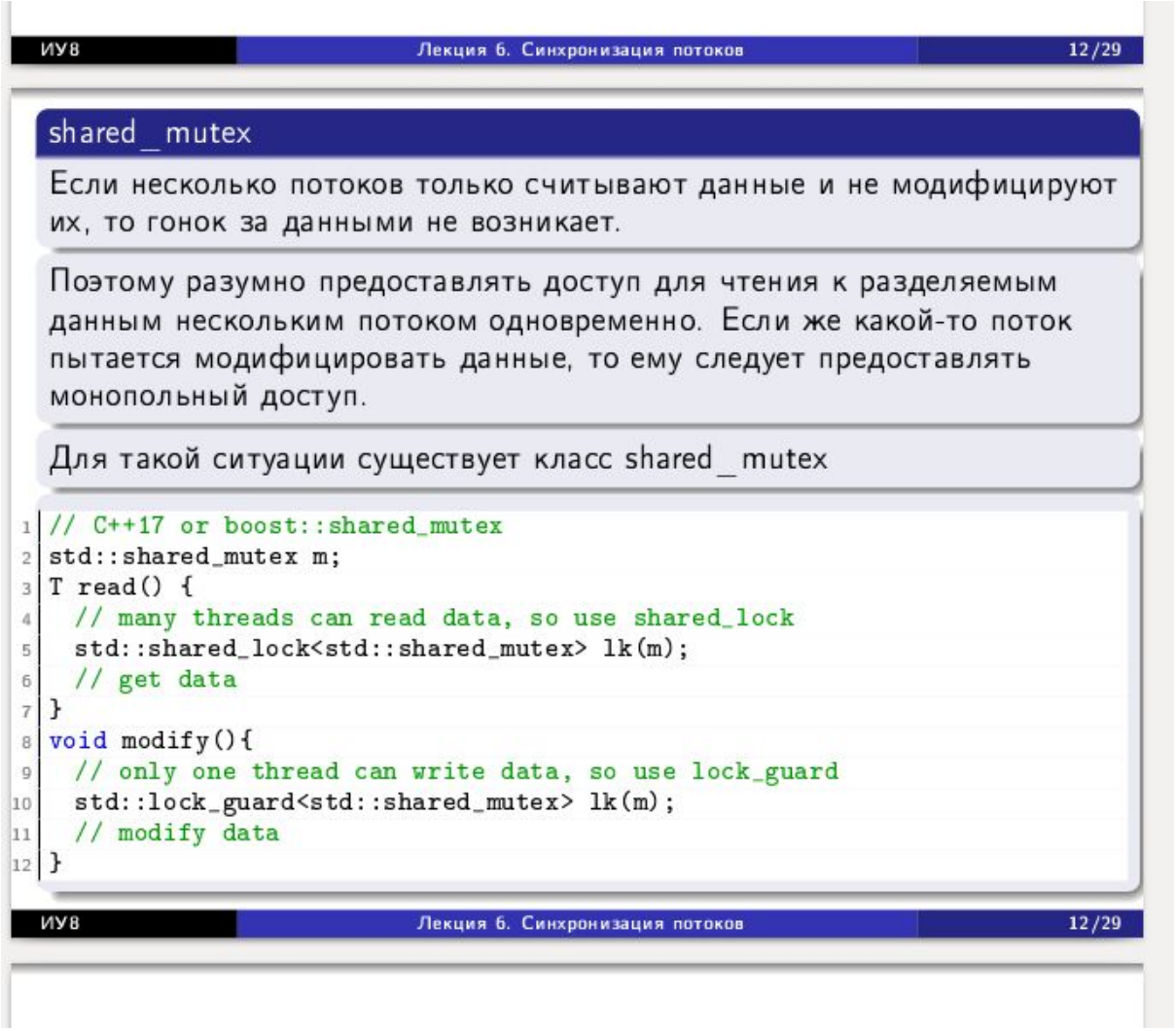
Стандартная библиотека C++ допускает вторую попытку генерирования исключения `std::system_error` (см. раздел 4.3.1) с кодом ошибки `resource_deadlock_would_occur` (см. раздел 4.3.2), если платформа распознает такую взаимную блокировку. Однако это не обязательное требование, и обычно оно не выполняется.

Использование мьютекса `recursive_mutex` решает описанную проблему. Этот мьютекс допускает многократные блокировки одного и того же потока и освобождает блокировку, когда происходит последний вызов функции `unlock()`.


```
class DatabaseAccess
{
   private:
      std::recursive_mutex dbMutex;
      ... // состояние доступа к базе данных
   public:
      void insertData (...)
      {
         std::lock_guard<std::recursive_mutex> lg(dbMutex);
         ...
      }
      void insertData (...)
      {
         std::lock_guard<std::recursive_mutex> lg(dbMutex);
         ...
      }
      void createTableAndinsertData (...)
      {
         std::lock_guard<std::recursive_mutex> lg(dbMutex);
         ...
         createTable(...); // OK: взаимной блокировки нет
      }
     ...
};
```

Попытки блокировки и временные блокировки

Иногда программа хочет установить блокировку, но не хочет блокировать мьютекс (на-



28Синхронизация потоков. Состояние гонок. Классы std::mutex, std::lock_guard, std::unique_lock. Функция std::lock. Примеры использования мьютексов.

См /home/netta/Downloads/AL_EXAM/Code/condition_varible

26-27Переключение контекста потоков. Класс std::thread. Ключевое слово thread_local. Примеры использования std::thread.

Ex:

```
#include <iostream>
#include <thread>
#include <chrono>

void foo()
{
   // simulate expensive operation
   std::this_thread::sleep_for(std::chrono::seconds(1));
}

void bar()
{
   // simulate expensive operation
```

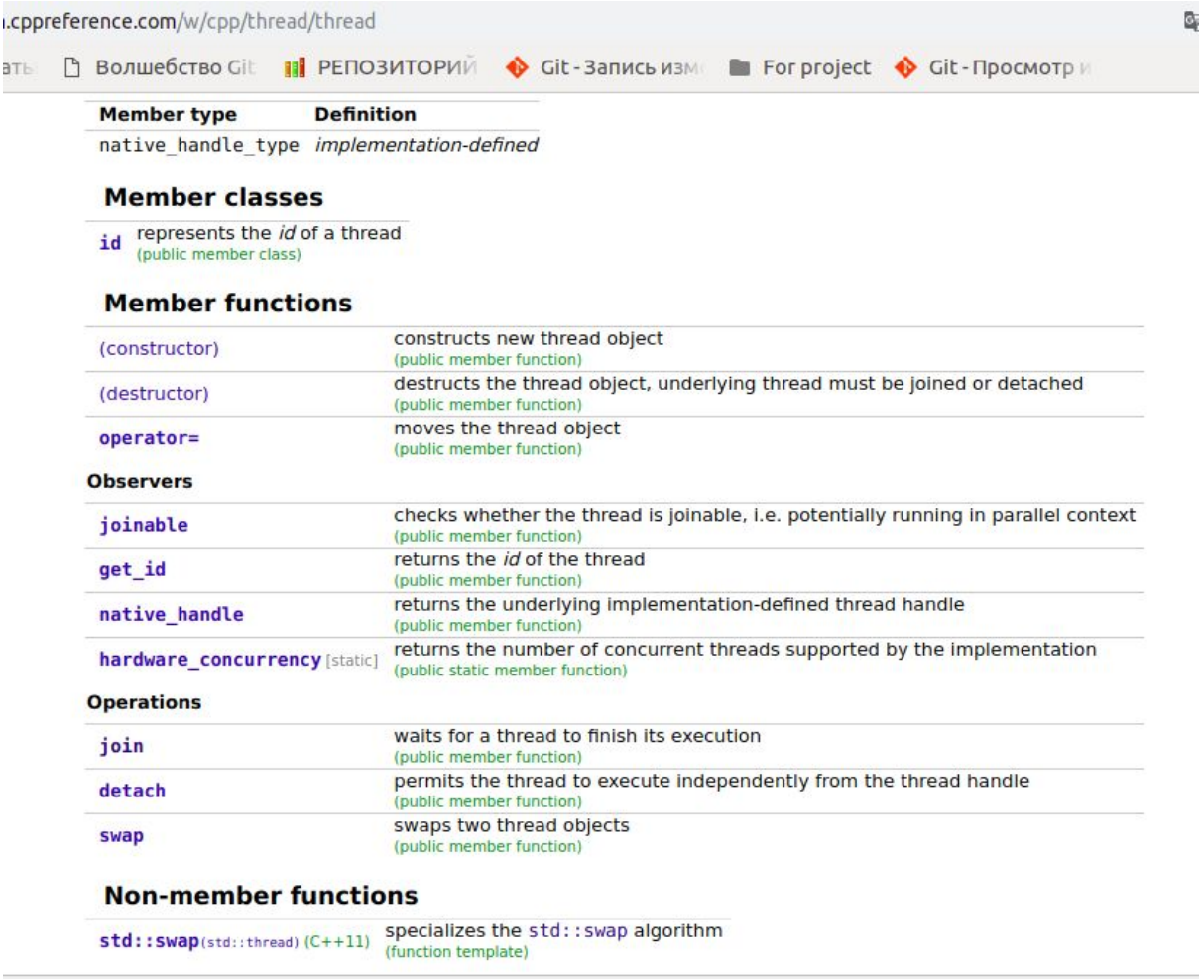
```
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main()
{
    std::cout << "starting first helper...\n";
    std::thread helper1(foo);

    std::cout << "starting second helper...\n";
    std::thread helper2(bar);

    std::cout << "waiting for helpers to finish..." << std::endl;
    helper1.join();
    helper2.join();

    std::cout << "done!\n";
}
```



.cppreference.com/w/cpp/thread/thread

Волшебство Git | РЕПОЗИТОРИЙ | Git - Запись изм | For project | Git - Просмотр и

Member type	Definition
native_handle_type	*implementation-defined*

Member classes

id	represents the *id* of a thread (public member class)

Member functions

(constructor)	constructs new thread object (public member function)
(destructor)	destructs the thread object, underlying thread must be joined or detached (public member function)
operator=	moves the thread object (public member function)
Observers	
joinable	checks whether the thread is joinable, i.e. potentially running in parallel context (public member function)
get_id	returns the *id* of the thread (public member function)
native_handle	returns the underlying implementation-defined thread handle (public member function)
hardware_concurrency [static]	returns the number of concurrent threads supported by the implementation (public static member function)
Operations	
join	waits for a thread to finish its execution (public member function)
detach	permits the thread to execute independently from the thread handle (public member function)
swap	swaps two thread objects (public member function)

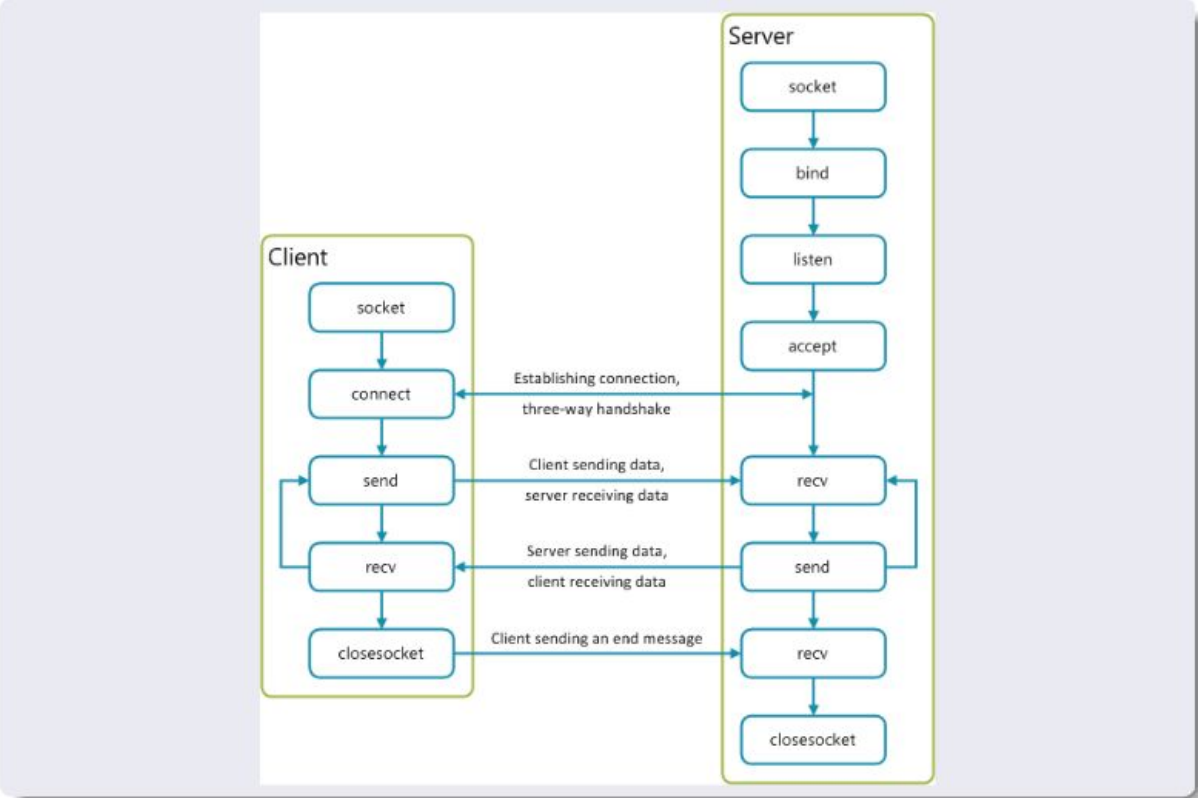
Non-member functions

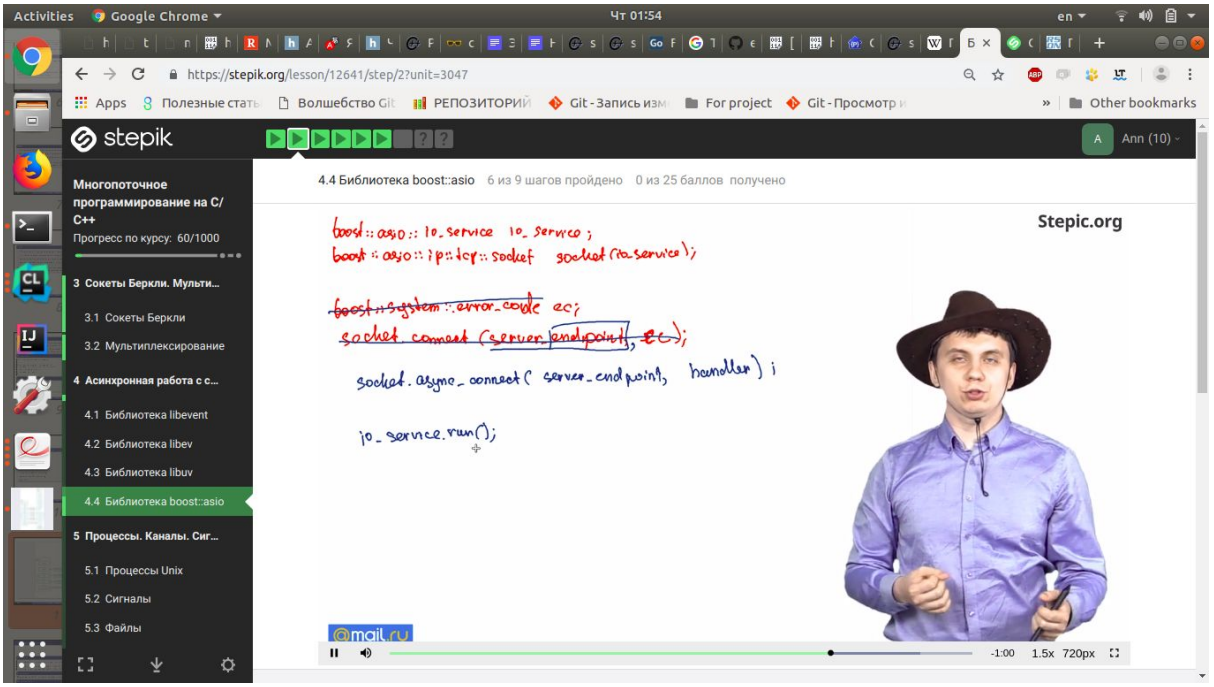
std::swap(std::thread) (C++11)	specializes the std::swap algorithm (function template)

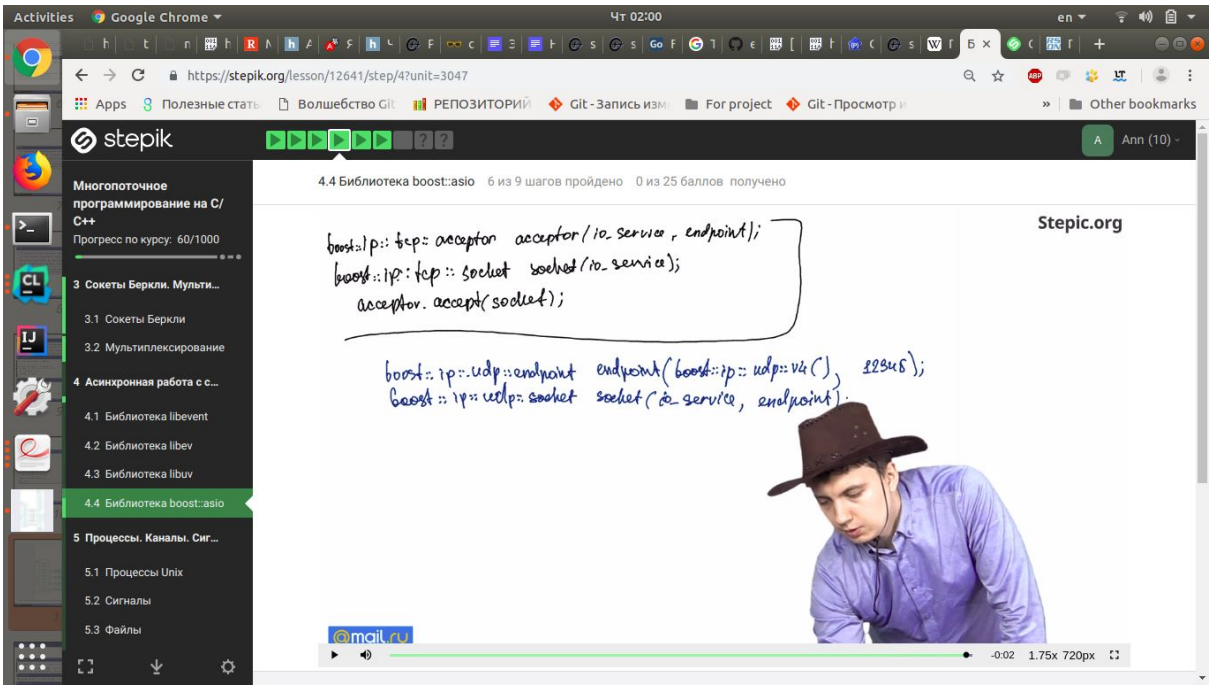
25 Управление потоками. Состояния гонок в интерфейсе структур данных. Класс std::future, функция std::async. - см 33

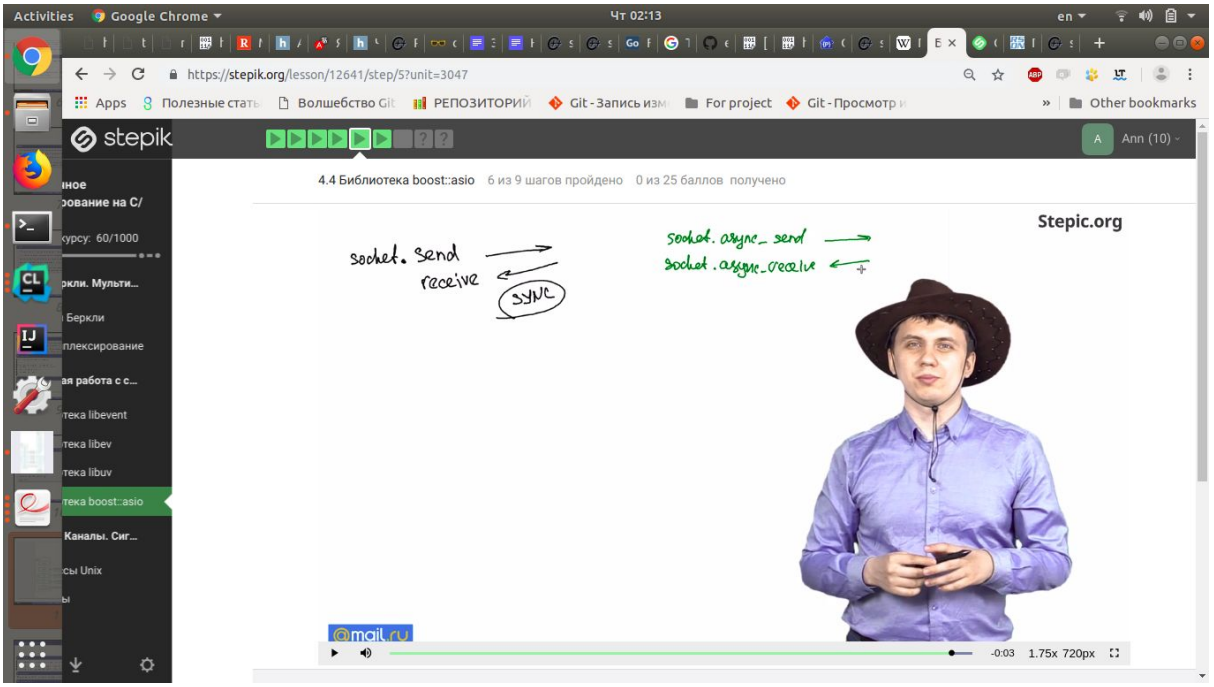
24Сетевое взаимодействие. Сокеты. Библиотека boost::asio. io_service, endpoint, ip::address. Функции асинхронного и синхронного чтения и записи.

https://stepik.org/lesson/12576/step/13?unit=3004 - про сокеты Беркли




Activities
Google Chrome
Чт 01:54
en
https://stepik.org/lesson/12641/step/2?unit=3047
Apps
Полезные стать
Волшебство Git
РЕПОЗИТОРИЙ
Git - Запись изм
For project
Git - Просмотр и
Other bookmarks
stepik
Ann (10)
Многопоточное программирование на C/C++
Прогресс по курсу: 60/1000
3 Сокеты Беркли. Мульти...
3.1 Сокеты Беркли
3.2 Мультиплексирование
4 Асинхронная работа с с...
4.1 Библиотека libevent
4.2 Библиотека libev
4.3 Библиотека libuv
4.4 Библиотека boost::asio
5 Процессы. Каналы. Сиг...
5.1 Процессы Unix
5.2 Сигналы
5.3 Файлы
4.4 Библиотека boost::asio 6 из 9 шагов пройдено 0 из 25 баллов получено
Stepic.org
boost::asio::io_service io_service;
boost::asio::ip::tcp::socket socket(io_service);
boost::system::error_code ec;
socket.connect(server_endpoint, ec);
socket.async_connect(server_endpoint, handler);
io_service.run();
@mail.ru
-1:00
1.5x
720px


Activities
Google Chrome
Чт 02:00
en
https://stepik.org/lesson/12641/step/4?unit=3047
Apps
Полезные стать
Волшебство Git
РЕПОЗИТОРИЙ
Git - Запись изм
For project
Git - Просмотр и
Other bookmarks
stepik
Ann (10)
Многопоточное программирование на C/C++
Прогресс по курсу: 60/1000
3 Сокеты Беркли. Мульти...
3.1 Сокеты Беркли
3.2 Мультиплексирование
4 Асинхронная работа с с...
4.1 Библиотека libevent
4.2 Библиотека libev
4.3 Библиотека libuv
4.4 Библиотека boost::asio
5 Процессы. Каналы. Сиг...
5.1 Процессы Unix
5.2 Сигналы
5.3 Файлы
4.4 Библиотека boost::asio 6 из 9 шагов пройдено 0 из 25 баллов получено
Stepic.org
boost::ip::tcp::acceptor acceptor(io_service, endpoint);
boost::ip::tcp::socket socket(io_service);
acceptor.accept(socket);
boost::ip::udp::endpoint endpoint(boost::ip::udp::v4(), 12345);
boost::ip::udp::socket socket(io_service, endpoint);
@mail.ru
-0:02
1.75x
720px



17 Обработка ошибок с использованием механизма обработки исключений. RAII. Примеры классов, использующих RAII. Ключевое слово noexcept.

- **https://habr.com/post/253749/**