

# Параллелизм в C++

future<RetType> ( ) M M=

**RetType | RetType& | void get()**  
возвращает результат типа ResType, блокируется пока результат не доступен

**bool valid()**  
true если *get* не был вызван

**shared\_future<RetType> share()**  
конвертирует *future* в *shared\_future*

**void wait()**  
блокируется, пока результат не будет доступен

**future\_status wait\_for(const duration&)**  
ожидает результат в течение какого-то времени

**future\_status wait\_until(const time\_point&)**  
ожидает результат до определённого времени

shared\_future<RetType>

( ) C C= M M=

**shared\_future(future<RetType>&&)**  
move-создание из *future*

**RetType | RetType& | void get()**  
возвращает результат типа ResType, блокируется пока результат не доступен

**bool valid()**  
true если *get* не был вызван

**shared\_future<RetType> share()**  
конвертирует *future* в *shared\_future*

**void wait()**  
блокируется, пока результат не будет доступен

**future\_status wait\_for(const duration&)**  
ожидает результат в течение какого-то времени

**future\_status wait\_until(const time\_point&)**  
ожидает результат до определённого времени

Legend

( )Конструктор по-умолчанию

CКонструктор копирования

MКонструктор перемещения

C=Оператор присваивания

M=Оператор перемещения

↔Метод swap

thread ( ) M M= ↔

**thread<F, Args...>(F&&, Args&&...)**  
конструирование из F и Args

**bool joinable()**  
true если поток не был отделен (detached)

**void join()**  
блокируется пока поток не завершится

**void detach()**  
отказаться от контроля над потоком

**id get\_id()**  
возвращает ID потока

**native\_handle\_type native\_handle()**  
возвращает платформенно-зависимый handle потока

**static unsigned hardware\_concurrency()**  
возвращает количество аппаратных потоков

this\_thread namespace

**thread::id get\_id()**  
возвращает ID вызывающего потока

**void yield()**  
уступает процессорное время другим потокам

**void sleep\_until(const time\_point&)**  
блокирует вызывающий поток до определённого времени

**void sleep\_for(const duration&)**  
блокирует вызывающий поток на определённое время

Free functions

**future<RetTypeOff> async([launch], F&&, Args&&...)**  
Возвращает *future* и вызывает *F* с *Args* в соответствии с политикой запуска (если задана) или с *launch::async* / *launch::deferred* иначе

**void lock<L1, L2, L3...>(L1&, L2&, L3&...)**  
захватывает все аргументы без дедлоков, в случае неудачи освобождает все аргументы и завершает работу

**int try\_lock<L1, L2, L3...>(L1&, L2&, L3&...)**  
вызывает *try\_lock* для каждого аргумента если аргумент не может быть захвачен – освобождает все предыдущие аргументы и возвращает индекс незахваченного

**void call\_once(once\_flag&, F&&, Args&&...)**  
вызывает *F* с *Args* только однажды

lock\_guard<Mutex>

**lock\_guard(Mutex&, [adopt\_lock\_t])**  
Захватывает мьютекс при создании и освобождает при удалении

packaged\_task<RetType, ArgTypes...> ( ) M M= ↔

**packaged\_task<F>(F&&)**  
**packaged\_task<F, Alloc>(allocator\_arg\_t, const Alloc&, F&&)**

конструирует из *F*, использует *Alloc* для внутренних данных (если задано)  
**future<RetType> get\_future()**  
возвращает future для данной задачи

**void operator()(ArgTypes...)**  
выполняет задачу и даёт об этом знать *future*

**bool valid()**  
*true* если задача имеет разделяемое состояние

**void make\_ready\_at\_thread\_exit(ArgTypes...)**  
выполняет задачу и даёт об этом знать *future* по выходе из потока

**void reset()**  
создаёт новое разделяемое состояние, удаляет старое

promise<RetType> ( ) M M= ↔

**promise<Alloc>(allocator\_arg\_t, const Alloc&)**  
конструирует, используя Alloc для разделяемого состояния

**future<RetType> get\_future()**  
возвращает *future* для данного *promise*

**void set\_value(const RetType&)**  
**void set\_value(RetType&& | RetType& | void)**  
устанавливает результат и дёргает *future*

**void set\_exception(exception\_ptr)**  
устанавливает исключение и дёргает *future*

**void set\_value\_at\_thread\_exit(const RetType&)**  
устанавливает результат и дёргает *future* по выходе из потока

**void set\_exception\_at\_thread\_exit(exception\_ptr)**

устанавливает исключение и дёргает *future* по выходу из потока

unique\_lock<Mutex> ( ) M M= ↔

**unique\_lock(Mutex&, [defer\_lock\_t | try\_to\_lock\_t | adopt\_lock\_t])**  
по возможности захватывает мьютекс при создании

**mutex\_type\* release()**  
разблокирует и возвращает указатель на мьютекс

**bool owns\_lock()**  
*true* если мьютекс заблокирован

**mutex\_type\* mutex()**  
возвращает указатель на мьютекс

Также имеет все методы *timed\_mutex* (кроме *native\_handle*)

condition\_variable ( )

**void notify\_one()**  
разблокирует один из ожидающих потоков

**void notify\_all()**  
разблокирует все ожидающие потоки

**void wait(unique\_lock<mutex>&, [Predicate])**  
освобождает мьютекс и блокирует поток, пока условная переменная не изменится; использует *Predicate* для проверки ложных пробуждений

**cv\_status | bool wait\_until(unique\_lock<mutex>&, const time\_point&, [Predicate])**

как *wait*, но ожидает только до определённого времени; возвращает *cv\_status* или, если *Predicate* задан, значение *Predicate*

**cv\_status | bool wait\_for(unique\_lock<mutex>&, const duration&, [Predicate])**

как *wait*, но ожидает только определённое время; возвращает *cv\_status* или, если *Predicate* задан, значение *Predicate*

**native\_handle\_type native\_handle()**  
возвращает платформенно-зависимый handle

condition\_variable\_any ( )

Тоже что *condition\_variable*, но *wait*\*-методы позволяют задать определённый lock-класс вместо *unique\_lock*. Метод *native\_handle* не доступен

mutex/recursive\_mutex ( )

**void lock()**  
*recursive\_mutex* позволяет множественные вызовы из вложенных функций

**bool try\_lock()**  
немедленно возвращает *false* если захват невозможен

**void unlock()**

**native\_handle\_type native\_handle()**  
возвращает платформенно-зависимый handle

timed\_mutex/  
recursive\_timed\_mutex ( )

То же самое, что и *mutex/recursive\_mutex*, с двумя дополнительными методами:

**bool try\_lock\_for(const duration&)**  
Попытка захватить мьютекс (за какое-то время)

**bool try\_lock\_until(const time\_point&)**  
Попытка захватить мьютекс (до какого-то времени)