# C++ Concurrency

# future<RetType>







## RetType | RetType& | void get()

get the result - blocks until result is ready; return type determined by *RetType* template parameter

#### bool valid()

true if get hasn't been called

## shared\_future<RetType> share()

convert future to shared future

### void wait()

block until result is available

#### future status wait for(const duration&)

wait for the result for a specified period; unblock when result is available or after duration elapsed

## future\_status wait\_until(const time\_point&)

wait for the result until the specfied point in time; unblock when result is available or when time point passed

# shared future < RetType >













## shared future(future<RetType>&&)

move-construct from a future

### RetType | RetType& | void get()

get the result - blocks until result is ready; return type determined by *RetType* template parameter

#### bool valid()

true if get hasn't been called

# shared future<RetType> share()

convert future to shared future

#### void wait()

block until result is available

## future\_status wait\_for(const duration&)

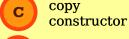
wait for the result for a specified period; unblock when result is available or after duration

## future status wait until(const time point&) wait for the result until the specfied point in time

# Legend









constructor



move assignment operator

copy assignment



swap method

operator

## cpprocks.com

@cpp\_rocks



## thread ()







# thread<F, Args...>(F&&, Args&&...)

construct from F and Args

#### bool joinable()

true if the thread hasn't been detached

#### void join()

block until thread completes

#### void detach()

give up control of the thread

#### id get id()

get thread ID

#### native\_handle\_type native\_handle() get platform specific thread handle

#### static unsigned hardware\_concurrency()

return an estimate of hardware thread contexts

# this thread namespace

#### thread::id get\_id()

return the unique ID of the calling thread

#### void yield()

offer the implementation a chance to reschedule

#### void sleep until(const time point&)

block the calling thread until specified time

#### void sleep\_for(const duration&)

block the calling thread for specified period

### Free functions

#### future < RetTypeOfF > async([launch], F&&, Args&&...)

return a future and execute F with Args according to launch policy if provided, or with *launch::async* | launch::deferred otherwise

#### void lock<L1, L2, L3...>(L1&, L2&, L3&...)

lock all arguments using a deadlock avoidance algorithm; in case of failure, unlock all previously locked arguments and return

# int try\_lock<L1, L2, L3...>(L1&, L2&, L3&...)

call *try lock* on each argument in order & return -1; if an argument can't be locked, unlock all previous arguments & return its index

# void call\_once(once\_flag&, F&&, Args&&...)

execute *F* with *Args* only once in a multi-threaded context

# lock\_guard<Mutex>

# lock\_guard(Mutex&, [adopt\_lock\_t])

lock the mutex on construction and release on destruction

# packaged\_task<RetType,

# ArgTypes...> ()

Alloc&, F&&)

(if provided)

bool valid()

void reset()

packaged\_task<F>(F&&)

future<RetType> get\_future()

void operator()(ArgTypes...)

true if the task has shared state

promise<RetType>

construct using Alloc for shared state

void set\_value(const RetType&)

set the result and signal the future

void set exception(exception ptr)

set an exception and signal the future

future<RetType> get\_future()

return a future for this promise

RetType& | void)

(exception ptr)

execute the task and signal the future

return a future for this task



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

construct from F, using Alloc for internal data structures

void make\_ready\_at\_thread\_exit(ArgTypes...)

execute the task and signal the future at thread exit

promise<Alloc>(allocator\_arg\_t, const Alloc&)

void set value(RetType&& | RetType& | void)

void set\_value\_at\_thread\_exit(const RetType&)

void set value at thread exit(RetType&& |

set exception and signal the future at thread exit

**Also** has the same methods as *timed mutex* (except

set result and signal the future at thread exit

unique\_lock(Mutex&, [defer\_lock\_t |

possibly acquire mutex on construction

unlock and return a pointer to mutex

void set exception at thread exit

unique lock<Mutex>

try\_to\_lock\_t | adopt\_lock\_t])

mutex type\* release()

true if the mutex is locked

mutex\_type\* mutex()

return a pointer to mutex

bool owns lock()

*native* handle)

construct new shared state, abandon old state







# unblock one of the waiting threads void notify\_all()

unblock all of the waiting threads

condition variable

void notify one()

# void wait(unique\_lock<mutex>&, [Predicate])

unlock the mutex and block the thread until the condition variable is signalled; use *Predicate* to check for spurious wakeups

#### cv\_status | bool wait\_until

## (unique lock<mutex>&, const time point&, [Predicate])

like wait, but only wait until specified time point; return cv status or, if Predicate is supplied, the value of *Predicate* 

### cv status | bool wait for

## (unique\_lock<mutex>&, const duration&, [Predicate])

like wait, but only wait for the specified duration; return cv status or, if Predicate is supplied, the value of Predicate

#### native handle type native handle() get platform specific handle

# condition variable any

Same interface as *condition variable*, but *wait*\* methods allow a custom lock class in place of unique lock, and native handle method isn't available

# mutex/recursive\_mutex



#### void lock()

recursive mutex allows multiple calls to lock with increasing levels of ownership

## bool try lock()

immediately return false if unable to lock

# void unlock()

#### native\_handle\_type native\_handle() get platform specific handle

# timed mutex/ recursive timed mutex



Same as *mutex/recursive mutex*, with two extra methods:

### bool try lock for(const duration&)

try to lock for the specified duration

#### bool try\_lock\_until(const time\_point&)

try to lock until the specified time point