Concurrency in modern C++

Nicola Bonelli

nicola@pfq.io

TOC

- Introduction to C++11/14
- RAII pattern
- Atomic operations
- Thread and namespace this_thread
- Mutex
- Mutex variants (timed_mutex, recursive_mutex…)
- Lock guard and unique lock
- Condition variable
- Promise/Future
- Async
- Compilers notes
- Exercises (monitor, working steal queue: producer/consumer)

Introduction to C++14

L-value vs. R-value expressions

Here are some general rules to distinguish between l-value and r-value expressions:

- An expression that can stand to the left of the assignment operator (=)
  - is an L-value
- An expression that **cannot** stand to the left of the assignment operator (=)
  - is an R-value

in addition...

- An expression that has a name is L-value
  - an expression with no name is R-value
- An expression that is possible to take the address of is L-value
  - otherwise R-value

L-value vs. R-value expressions (examples)

context	expression	kind	comment
	42	R-value	42 = 1; /* error! */
	“hello world”	R-value	“hello world” = “ciao”; /* error! */
int a = 10;	a	L-value	a = 42;
int a = 10;	(1+a)	R-value	(1+a) = 42; /* error! */
	std::string(“test”)	R-value	std::string(“test”) = “error!”;

L-value vs. R-value expressions (examples)

context	expression	kind	comment
int fun();	return value	R-value	fun() = 10; /* error! */
int & fun();	return value	L-value	fun() = 11;
int fun(std::string v)	v	L-value	v = “hello world”;
int fun(std::string &v)	v	L-value	v = “hello world”;
int fun(std::string &&v)	v	L-value	v = “hello world”;

Observations

- Unless (L-value) references, values returned from functions are R-values
- Regardless they are L- or R-value expressions, arguments passed to functions by value are L-values within the function body
    - because they have a name

and... (pre-C++11 rules):

- references can bind to L-value expressions
- references **cannot** bind to R-value expressions
- **const references** can bind to both L- and R-value expressions

R-value references

- Starting from C++11, references (&) are named L-value references
- C++11 **introduces** R-value references (&&) which can bind only to R-value expressions

Examples:

```
int fun(int &a) {...}
int fun(int &&b) {...}
```

```
int x = 10; fun(x);
```
-> call the red one

```
fun(10);
```
-> call the blue one

in addition...

```
fun(std::move(x))
```
-> call the blue one

std::move casts L-value expressions to R-value ones...

Move constructor (and move assign. operator)

- C++11 also introduces two additional special member functions:
  - Move constructor: `Object(Object && rhs)`
  - Move assignment operator: `Object& operator=(Object && rsh)`
- How a compiler decide to use move or copy constructor?
  - A ***temporary*** object is an R-value expression
  - A ***std::move(object)*** is an R-value expression
    - Call move constructor if available (or pick the right overloading), copy constructor otherwise
  - Returning objects from functions…
    - Try **RVO/NRVO** optimization…
    - Use move constructor, if available
- A moved object is left in a valid but unspecified state that is safe:
  - for being destroyed or re-assigned

Special Members

compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

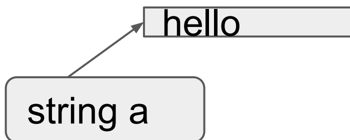
Move constructor (and move assign. operator)

- Classes from *std* are moveable when meaningful…
  - std::string is copyable, as well as moveable (assignable and move assignable)
  - all containers are moveable
  - std::unique_ptr<> is not copyable, but moveable (unique ownership)

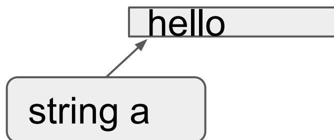
```
void fun(std::string name) {...}
```

```
std::string me = “Nicola”;
fun(me)                  -> copy constructor
fun(std::move(me))       -> move constructor
```

```
std::string a = “hello”;
std::string b(a);
```




```
std::string a = “hello”;
std::string b(std::move(a));
```



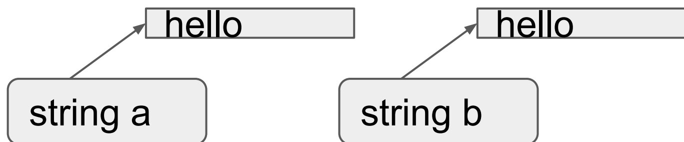
Move constructor (and move assign. operator)

- Classes from *std* are moveable when meaningful…
  - std::string is copyable, as well as moveable (assignable and move assignable)
  - all containers are moveable
  - std::unique_ptr<> is not copyable, but moveable (unique ownership)

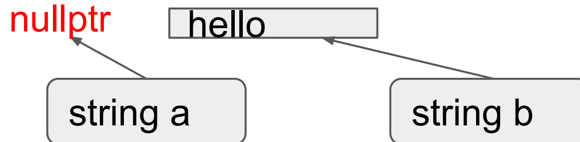
```
void fun(std::string name) {...}
```

```
std::string me = "Nicola";
fun(me)                 -> copy constructor
fun(std::move(me))      -> move constructor
```

```
std::string a = "hello";
std::string b(a);
```




```
std::string a = "hello";
std::string b(std::move(a));
```



Universal reference and perfect forwarding

- A special case is that of forwarding reference (or universal reference):
  - **forwarding reference == template r.value reference**

```
template <typename T>
void function(T && arg)          -> T && is unknown as Forwarding Reference
```

- It has a non-intuitive meaning because it obeys to:
  - type deduction of template arguments
  - collapsing rules
- A universal reference accepts both L-value and R-value expressions, but
  - Since the argument has a name (arg) within the body of the function it is always a L-value expression!!!!

Universal reference and perfect forwarding

- To fix this problem, a special function designed to restore the original L-value/R-value-ness of the expression is used: std::forward<T>

```
template <typename T>
void wrapper(T && arg)
{
	function( std::forward<T>(arg) );
}
```

- std::forward has the ability to cast the argument passed to:
  - L-value ref. if T was deduced as L-value ref.
    - if an L-value expression was passed to the function
  - R-value ref. otherwise
- perfect forwarding enables transparent wrappers
  - prior to C++11 transparent wrappers were possible only with macros

L-value ref, R-value ref, std::move and std::forward<T>

signature	std::move	std::forward<T>
void function(std::string arg)	OK	-
void function(std::string **&**arg)	BAD!	-
void function(std::string **const &**arg)	const &&?	-
void function(std::string **&&**arg)	OK	-
void function(**T &&** arg)	BAD!	OK

auto keyword and *for-loop* statement

- *auto* is a new keyword used to deduce the type of an expression in a declaration statement
  - it follows the type deduction rules of template functions
- *auto* is useful in several situations:
  - `auto n = 10;                      // deduced as int`
  - `auto result = function();         // deduced as the return type of function()`
  - `auto it = m.begin();              // std::map<std::string, int>::iterator it = …`
- another useful statement is for-loop: *for(element : container)*
  - similar to std::for_each, with the ability to break the iteration like a for

```
std::vector<std::string> vec; …
for (auto & elem : vec)
     std::cout << elem << std::endl;
```

Lambda expressions and callable types

- lambda expression is an unnamed function locally declared
  - often used in place of “functor”, as the implementation is defined in-place:

```
std::sort(std::begin(v), std::end(v),
          [](int l, int r) {
              return l > r;
          });
```

- lambda synopsis:
  - `[capture-list] (arguments...) -> ret_type { body... }`
- capture list includes the ‘local’ variables captured by lambda

[=]   => by default *capture all local variables in use by value*

[&]   => by default *capture all local variables in use by reference*

[this] => *capture `this` pointer by value*

[ ]   => *capture nothing*

Callable types

- A callable type is the type of an object that supports the call operator:
  - A pointer to function is a callable type:

```
void (*fun)(int x, std::string n) = &function;     -> fun(10, “hello world”);
```

  - A functor is a callable type:

```
struct functor {                                   -> functor fun;
     void operator()(int x, std::string n)            fun(10, “hello world”);
     {... }};
```

  - A lambda is a callable type:

```
auto fun = [] (int n, std::string n) { … };       -> fun(10, “hello world”);
```

  - …

- A function should always take a callback as a template argument:
  - `template <typename Callback> void high_order_function(Callback fun) { … };`

RAII: Resource Acquisition Is Initialization

- RAII is a well-known pattern that does not requires any special C++11-14 extension
- The idea is that of an object that performs some I/O operations in the constructor and (possibly) reverses the effects in the destructor:
    - An object file which opens the *fd* in the constructor and release (close) it in the destructor is a RAII object (std::fstream)
    - Smart pointers that get the ownership of a memory buffer in the constructor and release it in the destructor...
- RAII objects are typically declared in the stack of functions
    - => when the function returns or an exception is thrown the object in the stack are destroyed.
    - => therefore no “leak” is possible...
- RAII is used in multi-threading to safely perform mutex lock/unlock

```
#include <atomic>
```

Data race

- Data race: when more than one thread of execution access the same memory, with at least one writer.

```
int a = 0;
thread_1() { for(int i=0; i < 1000000000; ++i) {a++;} … }
thread_2() { for(int i=0; i < 1000000000; ++i) {a--;} … }
```

- Undefined behavior is expected!
    - What’s the value of **a** at the end? 0?
- To fix this one should use a mutex (LOCK/UNLOCK is pseudo code here):

```
mutex m;
thread_1() { for(int i=0; i < 1000000000; ++i) {LOCK(m); a++; UNLOCK(m);} … }
thread_2() { for(int i=0; i < 1000000000; ++i) {LOCK(m); a--; UNLOCK(m);} … }
```

Data race (notes)

- A data race is avoided with a mutex; but wait…
  - if a simple increment can be a data race, how can a mutex be implemented?
  - *It looks like a mutex is required in order to implement a mutex ?!?!?*
- Fortunately not!
  - It exists a set of elementary operations supported by the CPUs that are safe to be executed concurrently (and that operate on the same region of memory)
  - these operations are atomic, because they cannot be interrupted
- Before C++11…
  - atomic operations are implemented in assembly
    - no portability to different arch, #ifdef saga!
- Starting from C++11…
  - a subset common to all the existing CPUs is available!

Atomic operations

- What are the atomic operations supported by CPU?
    - load/store (that is read/write)
    - increment/decrement
    - add/sub
    - exchange
    - compare_exchange (a.k.a. CAS compare-and-swap)
    - and/or/xor
- If the architecture does not natively support a certain operation, a mutex is used

Atomic types

- Atomics are C++11 types that support atomic operations and correct memory alignment

```
std::atomic_char, std::atomic_schar, std::atomic_uchar
std::atomic_short, std::atomic_ushort
std::atomic_int, std::atomic_uint,
std::atomic_long, std::atomic_ulong,
std::atomic_llong, std::atomic_ullong,etc.
```

- In addition, a template version std::atomic<T> exists
  - std::atomic<int> is equivalent to std::atomic_int (etc.)
- A partial specialization for std::atomic<T *> also exists
  - std::atomic<int *> int_prt;

std::atomic<T>

- constructor: atomic(T value)
- copy constructor: atomic(atomic const &) = delete;
- …
- bool is_lock_free() const;
- void store(T value, std::memory_order order = std::memory_order_seq_cst);

```
std::atomic<int> a; a.store(1);
```

- T load(std::memory_order order = std::memory_order_seq_cst) const;

```
cout << a.load() << endl;
```

- *operator* T() const => equivalent to load()

```
cout << (1 + a) << endl;
```

- T operator++(), T operator++(int), T operator--(), T operator--(int) *(memory_order is not specifiable)*

```
++a; b--;
```

std::atomic<T>

- T operator +=(T arg) T operator-=(T arg) *(memory_order is not specifiable)*

```
atomic<int> a; a += 42;
```

- T operator&=(), T operator |=() and T operator ^=() *(memory_order is not specifiable)*

```
atomic<int> a(0xcafe); a &= 0xbeef;
```

- T fetch_add(T value, std::memory_order order = std::memory_order_seq_cst)
  T fetch_sub(T value, std::memory_order order = std::memory_order_seq_cst)
  T fetch_and(T value, std::memory_order order = std::memory_order_seq_cst)
  T fetch_or(T value, std::memory_order order = std::memory_order_seq_cst)
  T fetch_xor(T value, std::memory_order order = std::memory_order_seq_cst)

  perform the atomic operation and return the previous value, possibly with a specified memory order

std::atomic<T>

- T exchange(T new_value, std::memory_order order = std::memory_order_seq_cnt);

```
std::atomic<int> a(10);   cout << a.exchange(42) << endl;
```

- bool compare_exchange_*weak*(T & expected, T new_value, std::memory_order order = std::memory_order_seq_cnt); (also _*strong* version exists)

```
std::atomic<int> x (11);
int expected = 11;
x.compare_exchange_strong(expected, 42);
```

TRUE => x had value 11 and now has value 42
FALSE => x has a different value and it's stored in r

```
// pseudo code…
//
if (x == expected) {
        x = 42;
} else {
        expected = x;
}
```

- the difference between _weak and _strong version is:
  - _weak version under certain architecture may fail even if the value matches the expected one.
  - _strong version fails only in case of race-condition with another thread.

Memory models (hints)

- Memory models are policies that allow to control the way the architectures perform certain in-memory operations and related visibility
- They represent the building block for portable, lock-free data structures and algorithms
- In a nutshell, they allow to control the order of execution of certain operations
    - the compiler or CPUs may rearrange the order of instructions
    - other CPU may see the effects in a different order
- operations on <atomic> types have specifiable memory models
- the default memory_model_seq_cst (sequentially consistent)
    - guarantees a total order and no reordering is possible.

```
#include <thread>
```

std::thread

- std::thread class represents a thread of execution as an object
- std::thread is not copyable (copy ctor and assignment operator are explicitly deleted)
- std::thread is moveable (and move assignable)
- constructors:
  - `thread()` // default constructor, it does not represent a thread of execution
  - `thread( thread&& other )` // move constructor
  - `template< class Function, class... Args >`
    `explicit thread(Function&& f, Args&&... args)` // variadic template with perfect forwarding references (universal references)
- destructor:
  - `~thread()` // destructor: if the object is a joinable state std::terminate() is called

std::thread (methods)

- join: `void join()`
  - blocks the calling process/thread until the thread associated with the object finishes its execution
- detach: `void detach()`
  - detach the thread associated with the object, allowing the execution to continue independently
- joinable: `bool joinable() const`
  - return true if the thread is in a joinable state
    - not default constructed, not detached, not already joined
- get_id: `std::thread::id get_id() const`
  - return the std::thread::id of the thread
- ...

std::thread::id

- An instance of std::thread has an associated thread id (std::thread:id)
- A std::thread::id
  - is a unique identifier for a thread (like the pthread_t descriptor)
  - can be compared (with operator ==, !=, <, <=, >, >=),
  - has a stream operator (<<)
  - is hashable (a specialization of the functor class std::hash<> exists)
- std::thread::id can be used as Key in ordered associative containers (std::map) and unordered associative containers (like std::unorderd_map)
- A default constructed std::thread::id represents a non-thread of execution.

namespace std::this_thread

A collection of free functions is available to calling threads:

- std::this_thread::get_id
    - return the thread::id of the calling thread
- std::this_thread::sleep_for
    - sleep for a given amount of time, duration spec. in <chrono> (e.g. std::chrono::seconds(1))
- std::this_thread::sleep_until
    - sleep until a given time-point defined in <chrono> (e.g. std::chrono::system_clock::now())
- std::this_thread::yield
    - reschedule the calling thread, allowing other threads to run
- ...

std::thread (example)

```
void worker(std::string name, int n)
{
	for (int i = 0; i < n; ++i) {
	    std::cout << std::this_thread::id() << ‘:’ << n << std::endl;
	    std::this_thread::sleep_for(std::chrono::seconds(1));
	}
}
…
std::thread t (worker, “hello world”, 10);
bool test = t.joinable();        // true

t.join(); // block until t is terminated. (join or detach must be called before the object is destroyed)

// or… detach
t.detach();

// move the thread into a vector
std::vector<std::thread> workers;
workers.push_back(std::move(t));
```

std::thread (notes)

- All the arguments to be passed to the thread function must be passed to the std::thread constructor
  - they are forwarded to the the thread function
- The return value of the thread function is ignored!
  - To return a value from a thread you can either:
    - copy such a value into a global/shared object and (from the main thread) wait the value with join()
    - use a conditional_variable
    - use std::promise/std::future (new abstraction presented later)
- Exceptions thrown in the function thread terminate the program (if uncaught)
  - std::terminate is called
  - A good practice is to try { ... } catch { …} the whole thread function.

#include <mutex>

std::mutex

- A mutex is a synchronization primitive implemented as an non-copyable, non-moveable object.
- std::mutex is defined in <mutex>:
  - default constructible
  - destructible
  - non copyable
  - non moveable
- std::mutex has the following methods:
  - `void lock()      (block if the mutex is locked by another thread)`
  - `void unlock()`
  - `bool try_lock() (try to lock, return false in case the mutex is already locked).`

std::mutex (variants)

- Additional mutex types are:
  - std::recursive_mutex
  - std::timed_mutex
  - std::recursive_timed_mutex
  - std::shared_mutex (reader/writer mutex since C++17, today available from boost libraries)
- The recursive mutex permits the owner to take the lock multiple times
  - this prevents deadlock and is required under certain conditions (e.g. recursive functions)
- Timed mutex are equipped with two additional methods:
  - `template <typename …> bool try_lock_for(Duration dur);`
  - `template <typename …> bool try_lock_until(TimePoint tp);`

```
if (m.try_lock_for(std::chrono::milliseconds(10)) { … }
```

std::mutex

- std::mutex is not a RAII object
  - the effects (lock/unlock) are executed in methods and not in the constructor/destructor
  - having a mutex that locks in the ctor and unlocks in dtor is pointless, since the mutex is designed to be shared resource among threads of execution (e.g. declared global)

```
std::mutex m;
int a = 0;
....

void thread_1() {
      for(int i = 0; i < 1000000000; ++i)
      {
            m.lock(); a++; m.unlock();
      }
}
```

```
void thread_2() {
      for(int i = 0; i < 1000000000; ++i)
      {
            m.lock(); a--; m.unlock();
      }
}
```

std::lock_guard

- std::lock_guard is a generic (template) lock with RAII design
- it can be used with any kind of mutex equipped with lock/unlock functions
  - std::lock_guard<std::mutex> lock(m);
- std::lock_guard locks the passed mutex (by ref.) in the constructor and unlock it in the destructor…
- No explicit lock/unlock methods are exposed

```
std::mutex m;
int a = 0;
....

void thread() {
      for(int i = 0; i < 1000000000; ++i)
      {
            std::lock_guard<std::mutex> lock(m); // lock
            a++;
      }     // unlock
}
```

why std::lock_guard?

- std::lock_guard simplifies exception-safe code

  Example:

  An instance of std::vector<T> is shared among threads (protected by a mutex).
  Suppose that T is a copyable class whose copy ctor (as well as move ctor) may throw exceptions:

```
std::mutex m;
std::vector<Object> v;
....
void function(args) {
        Object data(args);
        try {
                m.lock();               // <- exception could be thrown here
                v.push_back(data);     // <- exception could be thrown here
                m.unlock();
        }
        catch(std::exception &e) {
                m.unlock();            // It is not safe to call if not locked!
                throw e;
        }
}
```

why std::lock_guard?

- std::lock_guard simplifies exception-safe code

  Example:

  An instance of std::vector<T> is shared among threads (protected by a mutex).
  Suppose that T is a copyable class whose copy ctor (as well as move ctor) may throw exceptions:

```
std::mutex m;
std::vector<Object> v;
....
void function(args) {
        Object data(args);
        try {
                m.lock();               // <- exception could be thrown here
                v.push_back(data);     // <- exception could be thrown here
                m.unlock();
        }
        catch(std::exception &e) {
                m.unlock();            // It is not safe to call if not locked!
                throw e;
        }
}
```

```
void function(args)
{
        O data(args);
        std::lock_guard<std:mutex> lock(m);
        v.push_back(data);
}
```

std::unique_lock (1/2)

- unique_lock<M> is a lock_guard<M> improved, with:
  - std::unique_lock(M &mutex); *(lock the mutex right now)*
  - std::unique_lock(M &mutex, std::defer_lock_t);

    don't lock the mutex right now (the lock can be taken later)…

    ```
    std::unique_lock<std::mutex> lock(m, std::defer_lock);
    ```
  - std::unique_lock(M &mutex, std::try_to_lock_t);

    try to lock, don't block in case it's already locked (by someone else)

    ```
    std::unique_lock<std::mutex> lock(m, std::try_to_lock);
    if (lock) { … } // explicit conversion to bool…
    ```
  - std::unique_lock(M &mutex, std::adopt_lock_t);

    the mutex is already locked (by me), don't lock it again!

std::unique_lock (2/2)

- destructor releases the lock (if owned)
- explicit locking methods:
  - void lock();
  - void unlock();
  - bool try_lock();
- additional methods:
  - template <typename…> bool try_lock_for(*Duration* dur);
  - template <typename…> bool try_lock_until(*TimePoint* tp);
  - M *mutex() const (*return a pointer to the resource mutex…)*
  - M *release() (*disassociate the mutex from the unique_lock, and return a pointer to the mutex or null is already released. No unlock takes place)*
  - bool own_lock() const;
  - explicit operator bool() const;
    - *return true if the unique_lock owns a locked mutex...*

std::unique_lock (example)

```
std::mutex m;
std::vector<int> vec;

void thread(int n) {
      std::vector<int> local;
      std::unique_lock<std::mutex> lock(m);
      vec.push_back(n);
      local = vec;
      lock.unlock();
      slow_IO(std::move(local));
      lock.lock();
      ...
      std::sort(std::begin(vec), std::end(vec));
}
```

- unique_lock allows to create holes within critical sections
- because IO operations are slow, unless necessary, they should take place outside critical sections
- because its features (the ability to explicitly lock/unlock the mutex) unique_lock is used with condition_variables.

std::lock_guard vs std::unique_lock

arch. 64-bits	**lock_guard (~no overhead)**	**unique_lock (internal state)**
size	8 bytes	16 bytes
moveable?	No	Yes
allows holes in critical sections?	No	Yes, with explicit unlock() and lock()
RAII?	Yes (pure)	Yes, different constructors
Allows a try_lock semantic?	No	Yes
work with condition_variable?	No	Yes
work with timed_mutex?	Yes, but no timeout is specifiable	Yes, try_lock_for, try_lock_until

```
#include <condition_variable>
```

std::condition_variable

- condition_variable (CV) is another synchronization primitive used to block threads until a certain condition is satisfied
- since the CV is used in critical sections, a mutex is required
  - in particular a lock (and not directly a mutex) is used
  - because CV requires to unlock/lock the mutex, a unique_lock<M> is used (lock_guard is not suitable)
- constructors:
  - default constructor
  - copy constructor *deleted (CV is not copyable, move constructor not defined)*
- notify:
  - `void notify_one();`
  - `void notify_all();`

std::condition_variable

- `wait:`
  - `void wait(std::unique_lock<std::mutex> &lock);`
  - `template <typename Pred> void wait(std::unique_lock<std::mutex> &lock, Pred predicate);`
- `wait_for/wait_until:`
  - `template <typename ...>  std::cv_status wait_for(std::unique_lock<std::mutex> &lock, Duration dur);`
  - `template <typename Pred> std::cv_status wait_for(std::unique_lock<std::mutex> &lock, Duration dur, Pred predicate);`
  - `template <typename ...>  std::cv_status wait_until(std::unique_lock<std::mutex> &lock, TimePoint tp);`
  - `template <typename Pred> std::cv_status wait_until(std::unique_lock<std::mutex> &lock, TimePoint tp, Pred predicate);`
- std::cv_status:

```
enum class cv_status { no_timeout, timeout };
```

std::condition_variable

Differences between various *wait* methods:

- `wait` blocks the calling thread until it is notified (_one or _all)
- `wait` with predicate blocks the calling thread until the predicate is satisfied
  - this is useful to deal with "*spurious wakeup*"
  - when notified if the condition is not satisfied the calling thread blocks again
  - it is basically equivalent to:

```
while (!predicate())
{  condvar.wait(lock}; }
```

- `wait` with timeout (for a duration, or until a time-point)
  - blocks the calling thread until the thread is notified (or the predicate is satisfied) or the timeout is expired
  - return `cv_status::no_timeout` if notified (or predicate satisfied), `cv_status::timeout` otherwise

std::condition_variable (basic with spurious wakeup)

```
std::condition_variable condvar;
std::mutex m;

...

    // thread ...
    {
      std::unique_lock<std::mutex> lock(m);

      condvar.wait(lock);

      // 1) while the thread is waiting the mutex is unlocked!
      // 2) this thread might be woken up spuriously!!!


      std::cout << "this thread just woke up!" << endl;

    }
```

```
// thread ...

condvar.notify_one();      // wake-up one thread

condvar.notify_all();      // wake-up all threads
                              (order is unspecified)
```

std::condition_variable (spurious wakeup handled)

```
std::condition_variable condvar;
std::mutex m;

bool signal = false;

 // thread …
    {
      std::unique_lock<std::mutex> lock(m);

      condvar.wait(lock, []() { return signal; });
      signal = false;

      // while the thread is waiting the mutex is unlocked!

      std::cout << "this thread just woke up!" << endl;

    }
```

```
// thread …
{
        std::lock_guard<std::mutex> lock(m);
        signal = true;
}

condvar.notify_one();     // wake-up one thread
```

#include <future>

std::promise<T>/std::future<T>

- std::promise and std::future provide a thread-safe mean to pass a value across threads
- the promise is used to store a value (or an exception) and the future is used to access such a value (or rethrow an exception) asynchronously
    - it's not a queue (rather a single message thread-safe mechinery)
- std::shared_future is used to access the value from multiple threads of execution
- a special `std::promise<void>` specialization is also available...

std::promise<T>

- constructor:
  - default constructor
  - move constructor
  - copy constructor deleted
  - destructor
- methods:
  - std::future<T> get_future()
  - ```
    void set_value(T const &)
    void set_value(T &)
    void set_value(T &&)
    void set_value()                          // for promise<void>
    ```
  - ```
    void set_exception(std::exception_ptr e)  // see: std::current_exception()
                                                      std::make_exception_pointer(e);
    ```
  - ...

std::future<T>

- constructor:
  - default constructor
  - move constructor
  - copy constructor deleted
- methods:
  - std::shared_future<T> share() // obtain a shared_future
  - bool valid() const // tells whether the value is in a valid state (default ctor or moved futures are **not** valid)
  - void wait() const // wait for value to be ready (or the promise associated released)

    template <typename ...> std::future_status wait_for(*Duration* dur) const

    template <typename ...> std::future_status wait_until(*TimePoint* tp) const
  - T get() // block until the result is ready and return it

    T& get()

    void get() // special. for future<void>

std::promise<T> & std::future<T> (notes)

- std::promise and std::future have an internal state which takes into account:
  - if the state is valid or not
  - if the value is available
  - if an exception is stored
- the value (or the exception) can be set into the promise only once
  - an exception is thrown otherwise
- the value (or the exception) can be get from the future only once
  - the behavior is undefined if the future is not in a valid state
- the future destructor:
  - blocks if the future was created with std::async and the value is not ready yet
  - does not block otherwise

promise + future example

```
#include <iostream>
#include <future>
#include <vector>
#include <thread>

...

      std::promise<std::vector<int>> pro;
      auto fut = pro.get_future();

      std::thread t(thread_fun, std::move(pro));
      t.detach();
      ...
      auto res = fut.get();
      for (auto elem : res)
            std::cout << elem << std::endl;
```

promise + future example

```cpp
#include <iostream>
#include <future>
#include <vector>
#include <thread>

...

    std::promise<std::vector<int>> pro;
    auto fut = pro.get_future();

    std::thread t(thread_fun, std::move(pro));
    t.detach();
    ...
    auto res = fut.get();
    for (auto elem : res)
        std::cout << elem << std::endl;
```

```cpp
void thread_fun(std::promise<std::vector<int>> pro)
{
    try
    {
        std::vector<int> ret;

        for(int i = 0; i < 10; ++i)
            ret.push_back(i);

        pro.set_value(std::move(ret));
    }
    catch(...)
    {
        pro.set_exception(std::current_exception());
    }
}
```

std::async

- To ease the use of *promise* and *future*, std::async packages everything in a function
  - std::thread, std::promise, std::future, exception handling
- std::async deduces the type for promise/future (as the return type of the callable function) and launches a computation (possibly) asynchronously
  - an optional policy specifies whether the computation is deferred (lazy) or asynchronous
- Synopsis:
  - **`auto fut = std::async(callable_function, args...);`**

```
int sum_v(std::vector<int> const &v)
{
     int ret = 0; for(auto elem : v) ret += elem; return ret;
}
auto sum = std::async(sum_v, std::vector<int>{1,2,3});
std::cout << sum.get() << std::endl;
```

std::async

- ```
  template <typename Function, typename … Args>
  std::future<...> std::async(Function fun, Args && ...args)
  ```
  - the computation may be evaluated in a new thread, or it is executed deferred (lazy)

- ```
  template <typename Function, typename … Args>
  std::future<...> std::async(std::launch policy, Function fun, Args && ...args)
  ```

- ```
  namespace std { enum class launch {
       async,         // launch the computation in a new thread
       deferred       // make a lazy evaluation
                         the computation is evaluated in the calling thread...
  }; }
  ```
  ```

std::async example

```
#include <iostream>
#include <future>
#include <vector>
#include <thread>
...

      auto fut = std::async(std::launch::async,
                            async_fun);
      ...
      auto res = fut.get();
      for (auto elem : res)
            std::cout << elem << std::endl;
```

```
std::vector<int> async_fun()
{
      std::vector<int> ret;
      for(int i = 0; i < 10; ++i)
            ret.push_back(i);
      return ret;
}
```

std::async + lambda example

```
#include <iostream>
#include <future>
#include <vector>
#include <thread>
...

      auto fut = std::async(std::launch::async,
                            []() {
                              std::vector<int> ret;
                              for(int i = 0; i < 10; ++i)
                                    ret.push_back(i);
                              return ret;
                            });
      ...
      auto res = fut.get();
      for (auto elem : res)
            std::cout << elem << std::endl;
```

std::async (defects)

- The ~future<T> is blocking when:
    - the future is created with std::async and the value is not ready
- This makes difficult to pass a std::future to generic code
    - behavior is different
- Therefore the following code (on the left) does not parallelize the *action*
    - e.g. an IO action that returns an integer read from a file

```
for(int i = 0; i < 10; ++i)
{
    std::async(std::policy::async, action);

}
```

```
std::vector<std::future<int>> vec;
for(int i = 0; i < 10; ++i)
{
    vec.push_back(std::async(std::policy::async,
                             action));
}
for (auto & fut : vec)
      std::cout << fut.get() << std::endl;
```

Compilers notes

- g++-4.9 (GNU compiler)
- clang-3.5 (apple/google open-source)
- options for different standards:
  - C++11: `-std=c++11`
  - C++14: `-std=c++1y` (not fully available)
  - draft C++17: `-std=c++1z`
- multithreading: -pthread
- optimizations: `-O0, -O1, -O2, -O3, -Os`
- debug options: `-g` (usually used with -O0)
- command line example:
  - `g++ test.cpp -o test -std=c++1y`
- preferred build-system:
  - cmake (www.cmake.org)

References

- Effective Modern C++: Scott Meyers
  - introduction to C++11/14
- C++ Concurrency in Action: practical multithreading - A. Williams
- http://en.cppreference.com/: C++11/14/~17 online documentation
- http://www.italiancpp.org/: italian C++ community

Exercises

1. Immaginare uno scenario in cui N thread incrementano una variabile condivisa e poi si sospendono per un tempo variabile tra i 3 e i 5 secondi, prima di proseguire. Implementare una barriera che impone a tutti i thread, una volta risvegliati, di attendere il risveglio degli altri prima di proseguire

2. Immaginare un buffer condiviso in cui scrivono e leggono due tipi di thread: lettori e scrittori. Ogni valore nel buffer è inserito da un solo scrittore, ma deve essere letto da tutti i thread. Le letture possono avvenire in parallelo. Implementare lo scenario con i thread C++14 e le strutture viste a lezione.