O'REILLY®

Live!

Effective Modern C++

42 SPECIFIC WAYS TO IMPROVE YOUR USE OF C++11 AND C++14

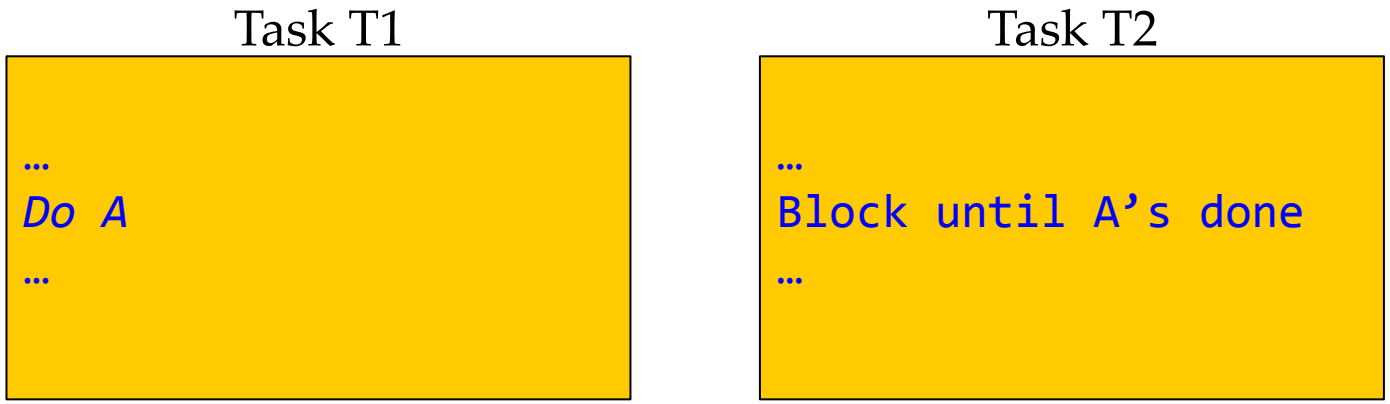
Scott Meyers

Last Revised: 3/30/15 11:03 PM

Consider `void` Futures for One-Shot Event Communication

Scenario:

- Concurrent tasks T1 and T2 begin.
- Among T1's work is some action A.
- At some point, T2 may continue only if A has occurred.



How have T1 tell T2 that A has taken place?

Poll (Multi-Choice Single Answer)

How would you approach this problem?

- Use a condition variable: T1 notifies T2 when A has occurred.
- Use a flag: T2 polls a flag that T1 sets when A has occurred.
- Use a condition variable and a flag: Avoids some drawbacks of each approach above.
- Use a future: T2 blocks on a future that T1 sets.
- None of the above.

Condition Variables

Basic idea:

- T2 `waits` on a condvar.
- T1 `notifys` condvar when A done.

```
std::condition_variable cv;
std::mutex m;
```

```
…
Do A
cv.notify_one();
…
```

Task T1

```
…
std::unique_lock<std::mutex> lk(m);
cv.wait(lk);
…
```

Task T2

Condition Variables

Three problems:

- Condvars require a mutex, but there's no state to protect.
  - ➡ "Feels wrong"
- Condvars may receive spurious notifications.
  - ➡ T2 must verify that A has occurred.
    - ◆ But that's our goal with the condvar!
- If T1 `notifys` before T2 `waits`, T2 blocks forever!
  - ➡ Must check to see if A has happened before `waiting`.
    - ◆ Again, our goal is to use the condvar for that.

```
…
Do A
cv.notify_one();
…
```

Task T1

```
…
std::unique_lock<std::mutex> lk(m);
cv.wait(lk);
…
```

Task T2

`std::atomic` Flags

Basic idea:

- T1 sets a flag when A occurs.
- T2 polls flag and proceeds only when flag set.

```
std::atomic<bool> flag(false);
```

```
…
Do A
flag = true;
…
```

Task T1

```
…
while (!flag);
…
```

Task T2

`std::atomic` Flags

Problem:

- Polling keeps T2 running, even though it's conceptually blocked.
  - ➡ Uses CPU (hence power).
  - ➡ Prevents other threads from using its HW thread.
  - ➡ Incurs context switch overhead each time it's scheduled.

Condition Variables + Flags

Basic idea:

- T2 `waits` on a condvar, checks a flag when `notifyd`.
- T1 sets a flag and `notifys` condvar when A occurs.

```
std::condition_variable cv;
std::mutex m;
bool flag;
```

```
…
Do A
{
  std::lock_guard<std::mutex> g(m);
  flag = true;
}
cv.notify_one();
…
```

Task T1

```
…
{
  std::unique_lock<std::mutex> lk(m);
  cv.wait(lk, []{ return flag; });
  …
}
…
```

Task T2

Condition Variables + Flags

Problem:

- Stilted communication protocol.
  - T1 sets flag ⇒ A has occurred, but condvar `notify` still necessary.
  - T2 wakes ⇒ A has probably occurred, but flag check still necessary.

`void` Futures

`void` is content-free. Its availability isn't.

- Unlike condvar:
  - ➡ No need for gratuitous mutex.
  - ➡ No need for "did event already/really occur?" mechanism.
- Unlike `std::atomic<bool>`, requires no polling.

Example: Starting Thread Suspended

Motivation:

- Separate thread creation from running something on it.
- Available on some platforms (e.g. Windows)

No direct support in C++11.

- Easy to implement via `void` future (at least in concept...).
- Function to run on thread still required at thread creation.
  ➡ "Function" ≡ "Callable object"

Starting Thread Suspended

```
std::promise<void> p;
```

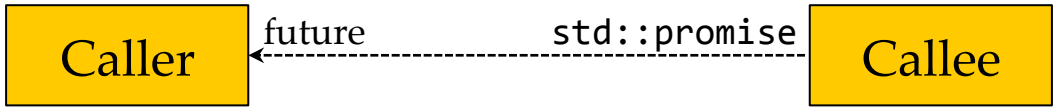
Poll (Multi-Choice Single Answer)

How familiar are you with C++11's `std::promises`?

- I've written code using them.
- I haven't used them, but I've read about them.
- I've heard of them, but that's about it.
- I've never heard of them.

Starting Thread Suspended

```
std::promise<void> p;
```




```
std::thread t2([&p]{ p.get_future().wait();   // start t2 and
                     funcToRun(); }           // suspend it
               );                             // (conceptually)
…                          // t2 is "suspended" waiting
                           // for p to be set
p.set_value();             // t2 may now continue
…
t2.join();
```

`std::thread` Destructor Behavior

A `std::thread` object may be *joinable*:

- Represent an underlying (i.e., OS) thread of execution (TOE).

Unjoinable `std::thread`s represent no underlying TOE.

- Default-constructed `std::thread`s.
- `std::thread`s that have been `detach`ed or moved from.
- `std::thread`s whose TOE has been `join`ed.
- Etc.

`std::thread` Destructor Behavior

`std::thread` dtor calls `std::terminate` if it's joinable:

```
{
  std::thread t(funcToRun);     // t is joinable
  …                             // assume t remains joinable
}                               // std::terminate called
```

`std::thread` Destructor Behavior

Implication: `std::thread`s must be unjoinable on all paths from block:

- Be unjoinable:
  - Be `joined` or
  - Be `detached` or
  - Be otherwise made unjoinable (e.g., moved from)
- All paths:
  - `continue`, `break`, `goto`, `return`
  - Flow off end
  - Exception

Unjoinable on *All Paths*

All paths ⇒ RAII classes.

- None for `std::thread` in standard library!
  - ➡ From the Standard:

    Either implicitly detaching or joining a `joinable()` thread in its destructor could result in difficult to debug correctness (for detach) or performance (for join) bugs encountered only when an exception is raised.
- Easy to write your own.

RAII Class for `std::thread`

```
class ThreadRAII {
public:
  enum class DtorAction { join, detach };

  ThreadRAII(std::thread&& t, DtorAction a)   // in dtor, take
  : action(a), t(std::move(t)) {}             // action a on t

  ~ThreadRAII()
  {
    if (t.joinable()) {
      if (action == DtorAction::join) t.join();
      else t.detach();
    }
  }

  std::thread& get() { return t; }

private:
  DtorAction action;
  std::thread t;
};
```

RAII Class for `std::thread`

```
ThreadRAII t1(std::thread(doThisWork),           // join on
              ThreadRAII::DtorAction::join);     // destruction
ThreadRAII t2(std::thread(doThatWork),           // detach on
              ThreadRAII::DtorAction::detach);   // destruction
```

RAII Class for `std::thread`

Dtor prevents generation of move ops, but they make sense, so:

```
class ThreadRAII {
public:
  enum class DtorAction { join, detach };          // as before

  ThreadRAII(std::thread&& t, DtorAction a)        // as before
  : action(a), t(std::move(t)) {}

  ~ThreadRAII()
  {
    … // as before
  }

  ThreadRAII(ThreadRAII&&) = default;              // support
  ThreadRAII& operator=(ThreadRAII&&) = default;   // moving

  std::thread& get() { return t; }                 // as before

private:                                           // as before
  DtorAction action;
  std::thread t;
};
```

Starting Thread Suspended

Use of `ThreadRAII` ensures `join` is always performed:

```
std::promise<void> p;                                  // as before
std::thread t2([&p]{ p.get_future().wait();  // as before
                     funcToRun(); }
               );
ThreadRAII tr(std::move(t2), ThreadRAII::DtorAction::join);
…                                                      // as before

p.set_value();                                         // as before

…
t2.join();                                             // no longer needed
```

Poll (Multi-Choice Multi-Answer)

Which of these consequences arise from using `ThreadRAII`?

- The `ThreadRAII` object will occupy stack space.
- `ThreadRAII`'s ctor/dtor will reduce the speed of the code.
- The code is exception safe.

Starting Thread Suspended

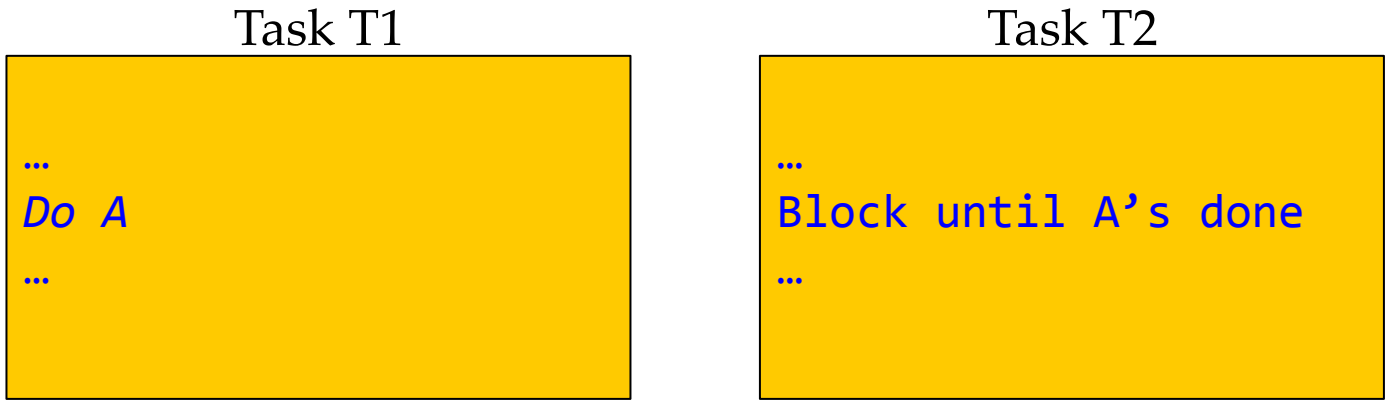
Use of `ThreadRAII` ensures `join` is always performed:

```
std::promise<void> p;                                  // as before
std::thread t2([&p]{ p.get_future().wait();   // as before
                     funcToRun(); }
               );
ThreadRAII tr(std::move(t2), ThreadRAII::DtorAction::join);
…
p.set_value();

…
t2.join();                                  // handled by ThreadRAII dtor
```

When A Doesn't Occur

What if T1 needs to tell T2 that A didn't occur (e.g., due to an exception)?



- **Condvar approach**: No `notify` ⇒ T2 blocks forever (modulo spurious wakes).
- **Flag approach**: Flag never set ⇒ T2 polls forever.
- **Condvar + Flag approach**: No `notify` + flag never set ⇒ T2 blocks forever (maybe with occasional spurious wakes).

Must use out-of-band mechanism to communicate "A didn't occur".

When A Doesn't Occur

Observations:

- Futures can hold exceptions.
- `std::promise` dtor puts an exception into an unset promise.

Ergo:

- Use local `std::promise` in T1.
  - ➡ It's set ⇒ T2 "receives" `void`.
  - ➡ It's destroyed w/o being set ⇒ T2 receives an exception.
- In T2, use `get` instead of `wait`.
  - ➡ Allows T2 to determine whether A occurred.
    - ◆ Non-exception: it did.
    - ◆ Exception: it didn't.

Starting Thread Suspended

```
{                                                    // make p local
  std::promise<void> p;
  std::thread t2([&p]
                 {
                   try {
                     p.get_future().get();
                     funcToRun();                    // A occurred
                   }
                   catch(...) {
                     …                               // A didn't occur
                   }
                 );
  ThreadRAII tr(std::move(t2), ThreadRAII::DtorAction::join);
  …
  p.set_value();
  …
}                                                    // exception written to p
                                                     // if it hasn't been set
```

Poll (Multi-Choice Single Answer)

Does this work, i.e., ensure that T2 gets an exception if A doesn't occur in T1?

- Yes.
- No.
- Leave me alone—it's nearly 5AM in Sydney, and it's already tomorrow!

Starting Thread Suspended

```
{
  std::promise<void> p;                              // created first,
                                                     // destroyed last

  std::thread t2([&p]
                 {
                   try {
                     p.get_future().get();
                     funcToRun();
                   }
                   catch(...) { … }
                 );

  ThreadRAII tr(std::move(t2),                        // created after
                ThreadRAII::DtorAction::join);        // p, destroyed
  …                                                   // before p

  p.set_value();
  …
}                                          // if p hasn't been set,tr's
                                           // dtor hangs on a join
```

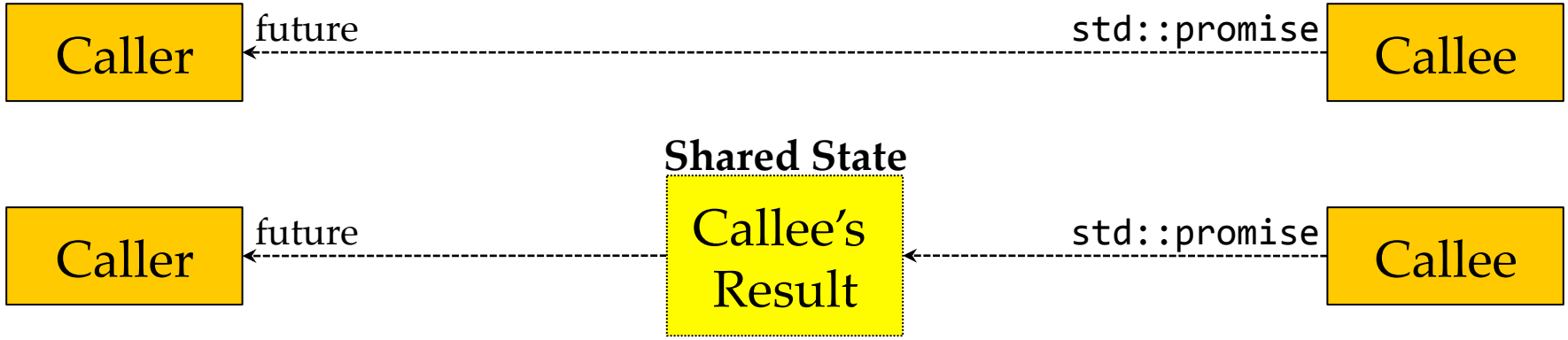
A Clean, Simple, Straightforward, Non-Error-Prone Solution

I don't know of one.

- I'm sorry, too.

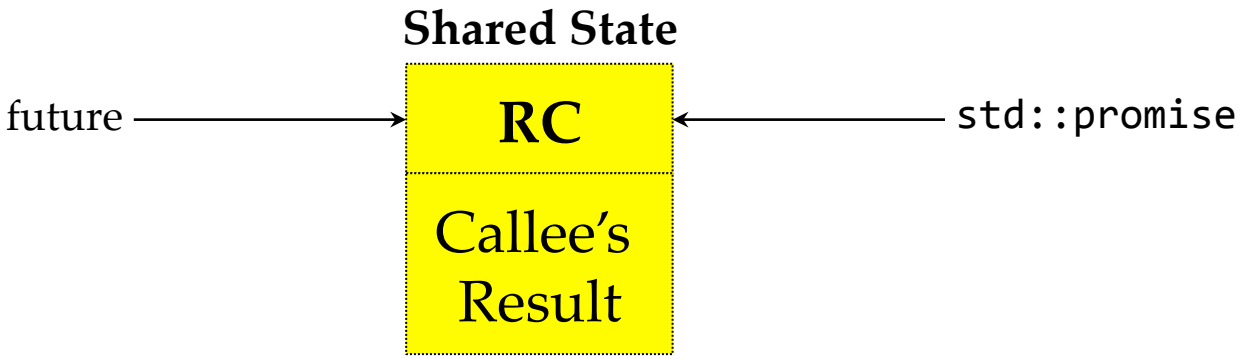
Image: "Depressed puma?,"(c) Tambako The Jaguar @ Flickr. License: Creative Commons Attribution-NoDerivs 2.0 Generic.

Costs for `void` Futures



Shared state:

- Dynamically allocated ⇒ use of heap.
- Reference-counted ⇒ atomic increments/decrements.



One-Shot Event Communication

Shared state may be written at most *once*.

- Not suitable for recurring communication.

Alternatives avoid this restriction:

- Condition variables.
  - ➡ May be `notify`-ed repeatedly.
- `std::atomics`.
  - ➡ May be assigned repeatedly.

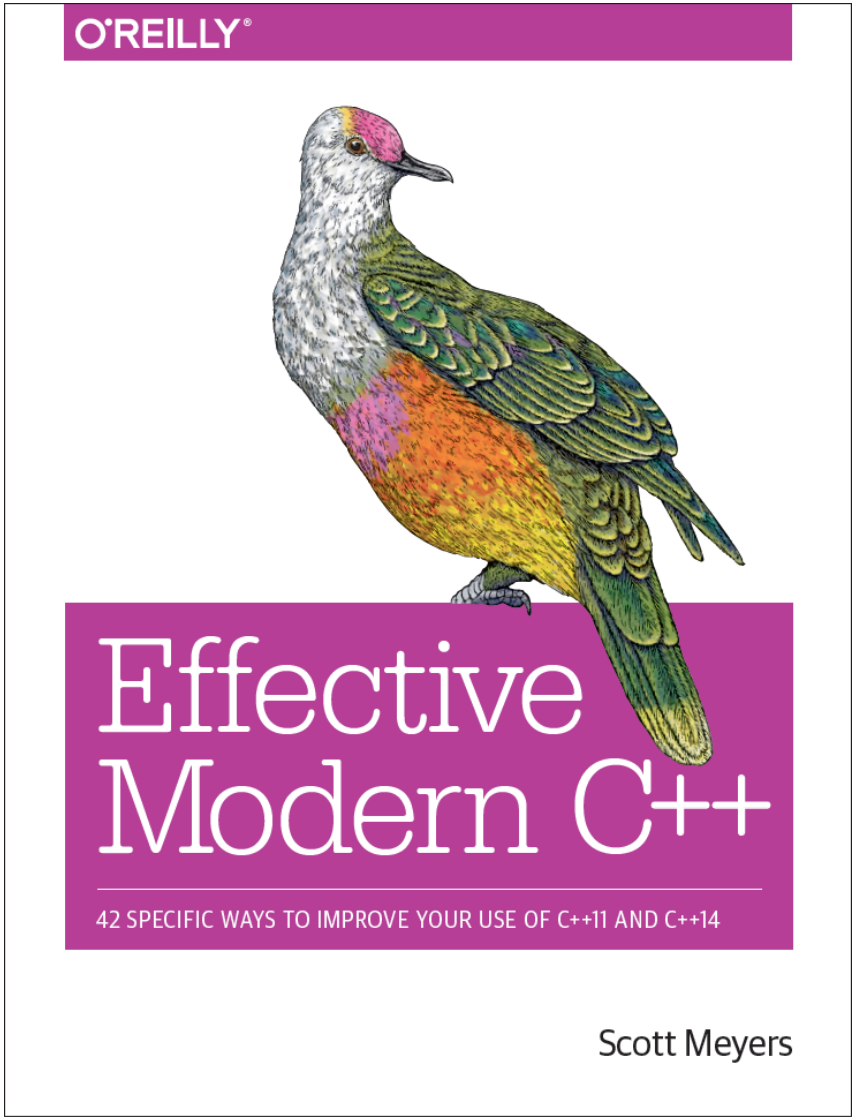
Guideline

Consider `void` futures for one-shot event communication.

Further Information

Source for this talk:

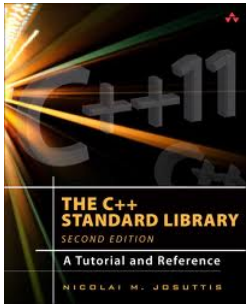
- *Effective Modern C++*, Scott Meyers, O'Reilly, 2015.
  - Item 17: Special member function generation.
  - Item 37: `ThreadRAII`.
  - Item 38: shared state.
  - Item 39: `void` futures.
- Email from Tomasz Kamiński.



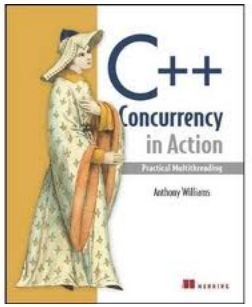
Further Information

The C++11 Concurrency API:

- *The C++ Standard Library, Second Edition*, Nicolai M. Josuttis, Addison-Wesley, 2012.



- *C++ Concurrency in Action*, Anthony Williams, Manning, 2012.



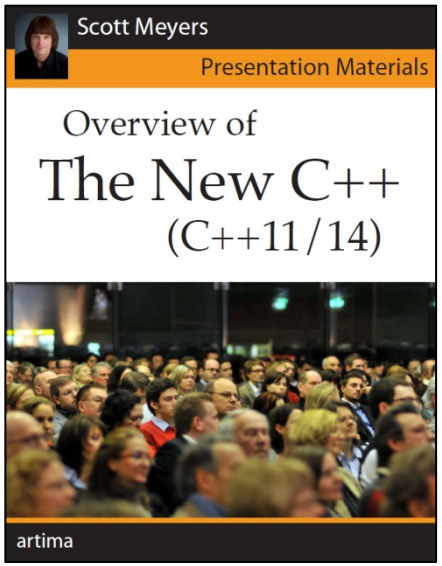
- "Why would I want to start a thread 'suspended'?," *Stack Overflow*, asked 1 July 2010.
- "ThreadRAII + Thread Suspension = Trouble?," Scott Meyers, *The View from Aristeia*, 24 December 2013.
- "std::futures from std::async aren't special!," Scott Meyers, *The View from Aristeia*, 20 March 2013.

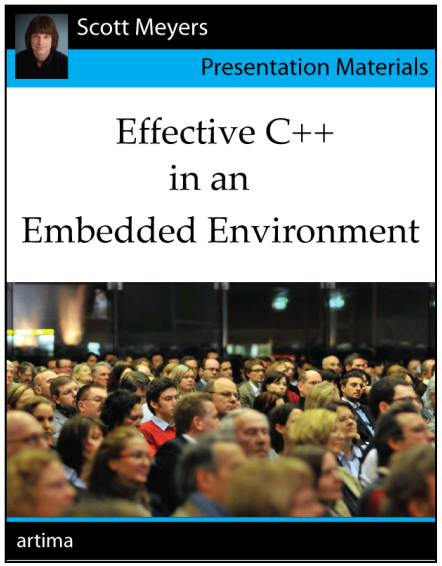
Licensing Information

Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- Commercial use: http://aristeia.com/Licensing/licensing.html
- Personal use: http://aristeia.com/Licensing/personalUse.html

Courses currently available for personal use include:





About Scott Meyers

Scott offers training and consulting services on the design and implementation of C++ software systems. His web site,

http://aristeia.com/

provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog