C++11 Concurrency

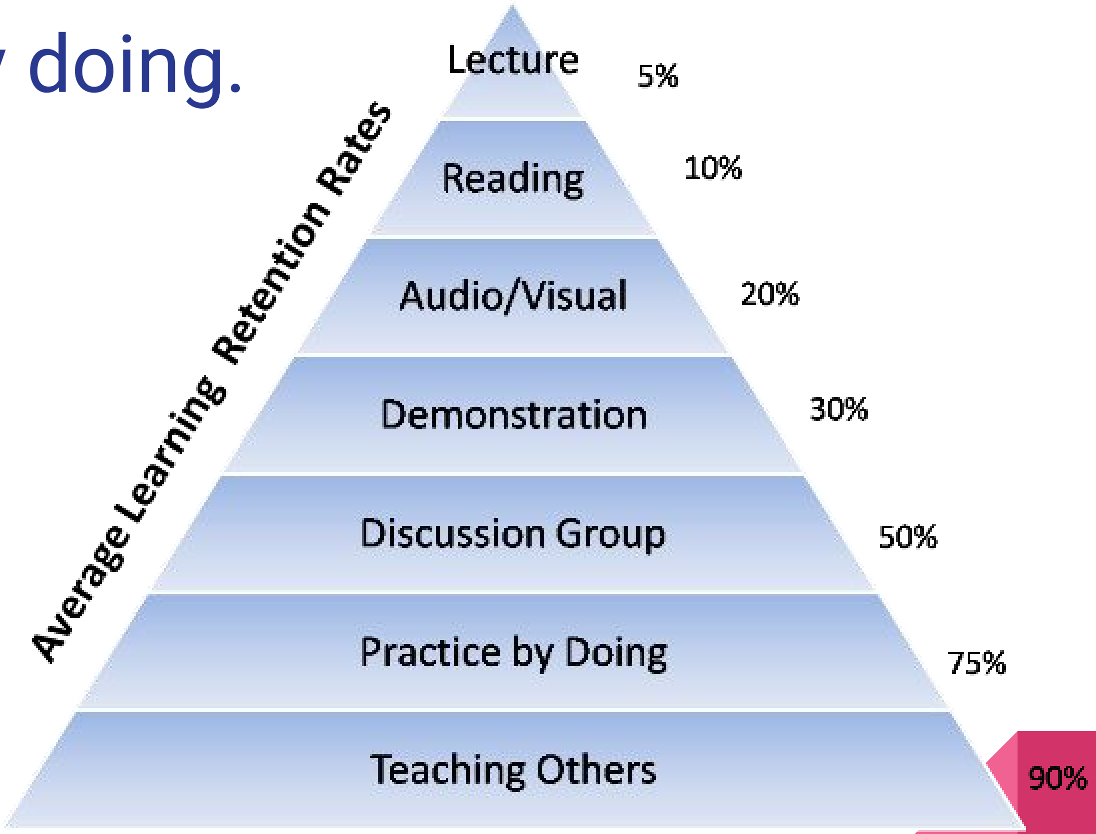
A step-by-step tutorial

<giorgio.zoppi@gmail.com>

How do we work today?

- Show C++11 Concurrency concepts: threads, futures, async, packaged_task, atomic
- Exercise those concepts with code. Learning by doing!
- We will do Tasks where you start coding.

Learning by doing.



Source: National Training Laboratories

Concurrent Hello World! <thread>

std::thread(func, arg1, arg2, …, argN);

Creates a thread that runs ‘func(arg1, arg2, …, argN)’ .

func is any Callable; lambdas and functor objects are fine.

- join();
- detach();
- std::thread::id get_id();

Please craft your thread!

```
#include <thread>

#include <iostream>

int main() {

std::thread hello([] { std::cout <<
"Hello, world!\n"; });

hello.join();

}
```

Thread management

1. **bool joinable() const noexcept**. Checks if the thread object identifies an active thread of execution. After created a thread is not joinable until it is not executed.
2. **native_handle_type native_handle()**..Threads own his own structure dependent on the OS.
3. If the thread goes out scope without joining or detaching, the program terminates (std::terminate). Why?
4. Threads are movable objects, please use std::move for passing ownership. In C++11 we think many things like an handle.
5. **std::terminate** is also called when you move-assign to a joinable std::thread. Semantically equivalent to point 3.
6. **std::thread::hardware_concurrency()** give us the number of cores in the box.
7. If a threads throws an exception, you will get std::terminate. Why?
8. Better use **std::async**, higher level interface.

Thread management. Where is the danger here?

Bad example of Parallel Fibonacci :

```
int fib(int n) {
    if (n <= 1) return n;
    int fib1, fib2;

    std::thread t([=, &fib1]{fib1 = fib(n-1);});
    fib2 = fib(n-2);
    if (fib2 < 0) throw std::runtime_error("Negative Value");
    t.join();
    return fib1 + fib2;
}
```

Again: whenever you can use std::async.

Task 1. Partial Cosine and Partial Sinus functions.

```
void psin(double angle)
 {
   double sum = 0;
   for (int i = 0; i< 100; ++i)
   {
     double result =
std::sin(angle*PI/180);
     sum+=result;
     std::cout << "partial sin(" <<
angle << "):" <<  sum  << "\n";
   }
 }
```

```
void pcos(double angle)
 {
   double sum = 0;
   for (int i = 0; i< 100; ++i)
   {
     double result =
std::cos(angle*PI/180);
     sum+=result;
     std::cout << "partial sin("
<< angle << "):" <<  sum  <<
"\n";
   }
 }
```

Task 1. Partial Cosine and Partial Sinus functions.

Given two function partial cosine and partial sinus:

1. create two threads thread_one and thread_two with std::thread
2. assign the partial_sin function at thread_one
3. move thread_one to thread_two
4. assign the partial_cos to thread_one
5. join all threads
6. launch the execution.
7. Remove the join...what happens and why?
8. Create a new std::thread thread_thread, move thread_one to thread_two and thread_two to thread_one, what happens and why?

std::this::thread and std::call_once

- std::this::thread
  - yield()
  - std::thread::id get_id()
  - sleep_for(const std::chrono::duration &)
  - sleep_until(const std::chrono::time_point &)
- std::call_once

We play bingo! :std::call_once.

```
int bingoWinner = -1;
void set_winner (int x) { bingoWinner = x; }

std::once_flag winner_flag;
int main()
{
  std::thread players[MAX_PLAYERS];
  std::srand (std::time(NULL));

  for (int i = 0; i < MAX_PLAYERS; ++i)
    {
      players[i] = std::thread([=,&i] {

          for (int i = 0; i < MAX_TRIAL; ++i)
            {
              int num = rand() % (BINGO_NUMBER + 1);
              std::cout << "Generated by thread " << i+1 << " Number: " << num << std::endl;
              if (num == BINGO_NUMBER)
                {
                  std::call_once(winner_flag, set_winner, i+1);
                  std::cout << "I have done bingo " << i+1 << std::endl;
                }
            }
        });
    }
  for (int i = 0; i < MAX_PLAYERS; ++i)
    {
      players[i].join();
    }
  std::cout << "This winner  is " << bingoWinner << std::endl;
}
```

Acquire and Release semantic.

A release store makes its prior accesses visible to a thread performing an acquire load that sees (pairs with) that store.

Thread 1.	Thread 2.
ACQ.LOCK L LOAD A, #10FF INC A STORE.RELEASE L	ACQ.LOCK L LOAD A, #10FF SUM A, 20 STORE.RELEASE L

We can get this semantic in C++11 with:

1. **mutex**
2. **atomics.**
3. **fences (we will not see fences here).**

<mutex>

- std::mutex a_mutex;
    - lock(), unlock() – “Lockable”
- std::lock_guard<std::mutex> a_lock;
    - RAII object representing scope-based lock
    - Simplifies reasoning about locks
- std::unique_lock<std::mutex>
    - RAII, but more flexible
    - Locks mutex in ctor, unlocks in dtor (if owned)
    - lock(), unlock()
    - bool try_lock()
    - bool try_lock_for(const std::chrono::duration &)
    - bool try_lock_until(const std::chrono::time_point &)

Task 2. Create a thread-safe singleton pattern.

1. You may use mutex
2. A better solution is to use call_once.
3. A better solution is to know about C+11 Memory model.

<condition_variable>

Event to wait for, periodically.

- std::condition_variable()
- notify_one(), notify_all()
- wait(std::unique_lock<std::mutex> & lk)
- NB: unique_lock!
- Releases lock, blocks thread until notification
- Upon notification: reacquires lock
- wait(std::unique_lock<std::mutex> & lk, pred)
- E.g.: cv.wait(lk, []{ return !input.empty(); });
- while (!pred()) { wait(lk); }
- wait_for(lk, duration, pred)
- wait_until(lk, time_point, pred)

Issues.

You never should assume that the predicate is true on wakeup. Why?

- Intercepted wakeups. What if some other thread acquires the mutex first?
- Loose predicates. You may want to do a condition.notify_all() based on the loose approximation of the state.
- Spurious wakeup. Multi Core and memory model.

Task 3. Concurrent Queue.

Write a thread safe blocking queue using condition variables and locks where is necessary.

```cpp
#include "concurrent_queue.hpp"
#include <exception>
#include <random>

void producer(concurrent::ConcurrentQueue<int>& q)
{
    std::default_random_engine dre(0X111);
    std::uniform_int_distribution<int> id(10,1000);
    for (int i =0; i < 10; ++i)
    {
        int n = id(dre);
        std::cout << " Generated number " << n << "." << std::endl;
        q.send(n);
    }
}
void consumer(concurrent::ConcurrentQueue<int>& q)
{
    for (int i = 0; i < 10; ++i)
    {
        int n = q.receive();
        std::cout << " Received number " << n << "." << std::endl;
    }

}
int main()
{
    concurrent::ConcurrentQueue<int> c_queue;
    try
    {
        auto prod = std::async(std::launch::async, producer, c_queue);
        auto consumer = std::async(std::launch::async,consumer, c_queue);
        prod.wait()
        consumer.wait();
    }
    catch (std::exception& e)
    {
        std::cout << e.what() << std::endl;
    }
}
```

Herb’s Quiz: What is the difference?

Mutex Locks.

```
class log
{
std::fstream log;
std::mutex mutex;
public:
void println(const std::string& s)
{
std::lock_guard<mutex> m;
 log << s <<  std::endl;
};
}
```

Message Queue.

```
class log
{
std::fstream log;
worker_thread thread;
public:
void println(const std::string& s)
{
        w.send([=] {
                log << s <<  std::endl;
        };);
}
};
```

Herb’s Quiz: What is the difference?

So the result of the quiz is **Avoid blocking like hell!**

C++11 provides us:

1. std::future / std::shared_future. **Great usability!**
2. std::async.

Very important for composability! We will see.

std::async and std::future.

std::future

- wait(). I want to wait for side effects.
- get(). I want to get the value.

A single thread is allowed to use the value. After a get() another get() can cause undefined behaviour.

std::shared_future (MCSP = Multiple consumer, single producer case).

Multiple threads are allowed using the result a shared state.

Future is great for composability. You may want to do:

```
auto r1 = std::async(std::launch::async, f1, 10);
auto r2 = std::async(std::launch::async,f2, 20);
auto r3 = std::async(std::launch::async,f3, r1.get(), r2.get());
auto v = r3.get();
```

Herb’s Quiz 2. Is this code parallel?

```
int f() { return 1;};
int g() { return 2;};
int main(int argc, char** argv)
{
std::async(std::launch::async, [] { f(); });
std::async(std::launch::async, [] { g(); });
}
```

Task 4. Parallel Addition

Using std::async and std::future provide an algorithm for parallel sum for a std::vector containing 10000 integers.

```
int main()

{

std::vector<int> v(10000, 1);

std::cout << "The sum is " << parallel_sum(v.begin(), v.end()) << '\n';

}
```

Task 4. Parallel Addition

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <future>

template <typename RAIter>
int parallel_sum(RAIter beg, RAIter end)
{
    typename RAIter::difference_type len = end-beg;
    if(len < 1000)
        return std::accumulate(beg, end, 0);

    RAIter mid = beg + len/2;
    auto handle = std::async(std::launch::async,
                             parallel_sum<RAIter>, mid, end);
    int sum = parallel_sum(beg, mid);
    return sum + handle.get();
}

int main()
{
    std::vector<int> v(10000, 1);
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end()) << '\n';
}
```

Task 5: std::promise and std::packaged_task

std::promise. *Provides a facility to store a value or an exception that is later acquired asynchronously via a std::future object created by the std::promise object:*

1. You should create a thread, pass a promise
2. In the main functor you set the value.
3. In the exception handler you set the exception in the promise.
4. In the main you get the future and see if there is an exception.

Do a simple class that tests the std::promise concept with two cases:

1. A sum of 100 random integers in the range (0,20).
2. Set an exception if the sum is more than 1000.
3. Get the value if the sum is less then 1000.
4. Launch the thread that is computing the sum.

std::packaged_task is useful for doing a workpool.

```
std::packaged_task<int(int,int)> task([](int a, int b) {
        return std::pow(a, b);
        });
        std::future<int> result = task.get_future();

        task(2, 9);
```

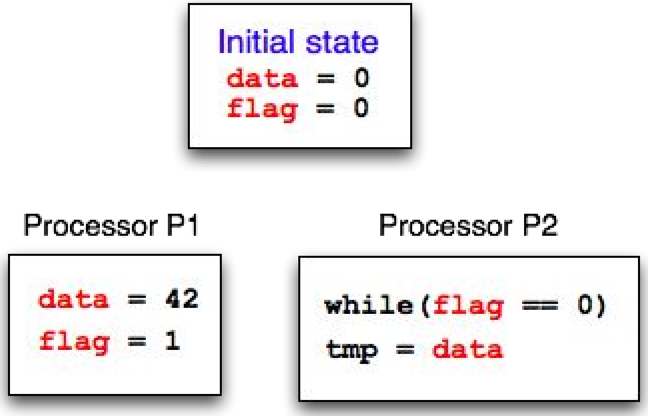
C++11 Memory Model.

Fact of life in C++: All data in a C++ program is made of objects. An object is region of storage and has a type. An object is stored in one or more memory location. **The CPU cores doesn’t execute the same program you wrote.** Why?

Fact of life in Programming Languages: All modern languages C++11, Java, Go support the Sequential Consistency model for data race free programs. In C++11 is the default and allows other relaxed models.

We are not saying a Friki-C+11 buzzwords...

C++11 Memory Model.



Sequential Consistency: ***the result of any execution is the same as if the reads and writes occurred in some order and the operations of each individual processor appear in this sequence in the order specified by its program.***

Processor memory model is acquire/release for data free race program.

C++11 default model is sequential consistency, so the runtime try to simulate as sequencial consistent program.

- sequenced-before (sb). If `A` is **sequenced before** `B`, then the execution of `A` *shall precede* the execution of `B`.
- synchronizes-with (sw). Acquire/Release semantic.
- happens-before (hb). A happens before an evaluation B if A is sequenced before B, or we have an interthread happens before.
- Memory locations & objects
- Reordering

Task 6. Synchronize With.

Create a lock-free sender and receiver that uses an atomic boolean for synchronize, using synchronize with semantic. The atomic is a guard to a message, the message is placed in memory (1-position Buffer). We have two functions:

1. void Send(const Message& m)
2. bool TryReceive(Message& m). This gets called in loop until the message is not received.

The message has:

1. an integer type
2. char value[1024];

It was hard ....but this...is ...

THE END!