3.1 Threads

Summary and Goals

The goal of the exercises in this section is to get basic hands-on experience with multi-threading in C++11. It can be seen as a transition from the Boost Thread library.
The following code example is useful because you customise it and use when testing code's processing time:

```
void TestChrono()
{ // Exemplar code for system performance

    std::chrono::time_point
          <std::chrono::system_clock> start, end;
    start = std::chrono::system_clock::now();

    // Do something heavy here
    // Introduce delay to simulate a heavy algorithm
    boost::this_thread::sleep
          (boost::posix_time::millisec(10000));

    end = std::chrono::system_clock::now();

    std::chrono::duration<double>
          elapsed_seconds = end - start;
std::time_t
          end_time = std::chrono::system_clock::to_time_t(end);

    std::cout << "Finished computation at " << "elapsed time: "
          << elapsed_seconds.count() << "s\n";

}
```

1. (Thread Class in C++11 101)

The objective of this exercise is to write a multithreaded program using `std::thread`. Specifically, we define a number of threads that print textual information on the system console (it is similar to the initial Boost thread example in section 3.1 of the course material). We wish to use the following features:

- Creating and launching instances of `std::thread`.
- Using various kinds of *callable objects* as a thread's start function.
- Waiting on threads to complete; has a thread finished executing code, but has not yet been joined? (use `joinable()`).
- Creating threads that run in the background (*detached/daemon* threads).
- Passing arguments to a thread function.

The threads that you create all call a function that prints a string a number of times on the console. A typical interface is `Iprint(const std::string& s,int count)`.

Answer the following questions:

a) Create as many callable objects as you can (free functions, lambda function and stored lambda function, function object, binded member function, static member function) that all call the interface `Iprint` above.
b) Create a thread for each of the callable objects in part a). One of the threads should be a detached thread. Each thread's function has its own unique ID.
c) Introduce code to check if threads have completed and then join each one if this is the case.
d) We need to introduce exception handling in the case that a thread may not be joined. In these cases we need to introduce `join()` inside the `catch` blocks.
e) Print the thread ID of each thread in each implementation of `Iprint`.
f) Compute the running time of the program based on the example code in the summary based on `std::chrono`.

Examine the output; what do you see? What is happening?

2. (Controlling Thread Execution: Synchronising Access to Shared Resources)

Moving on from exercise 1 we now synchronise access to the shared resource that is the console. In this exercise we use synchronising mechanisms to make the code *thread-safe* and hence avoid *race conditions*. In short, we use *locking mechanisms* to ensure that only one thread can access the console at one time.

Answer the following questions:

a) Use the `std::mutex` synchronisation to control access to the shared resource. Use `lock()` and `unlock()`. Measure processing time.
b) Now use `try_lock()` and measure the number of times each thread has failed to acquire the lock. Again, measure processing time.
c) Simulate *deadlock* (for example, a thread that never unlocks a mutex) by commenting out `unlock()` in your code. What happens?

3. (*Safe* Locking)

We continue with the code in exercise 2 by using two special lock mechanisms:

- The class `std::lock_guard`: a *mutex wrapper* that provides a convenient RAII-style mechanism for owning a mutex or several mutexes (since C++17) for the duration of a scoped block.
- The class `std::unique_lock`: a general-purpose mutex ownership wrapper allowing deferred locking, time-constrained attempts at locking, recursive locking, transfer of lock ownership, and use with condition variables.

In other words, `std::unique_lock` provides more flexibility than `std::lock_guard` by relaxing invariants; the former does not necessarily own the mutex that it is associated with.

Answer the following questions:

a) Modify the code in exercise 2 to let it work with `std::lock_guard`. Create a function that increments a counter by multiple threads. Check the output.
b) We now test the functionality of `std::unique_lock`. In particular, you should investigate its interface (for example, www.cppreference.com) in order to answer the questions in the rest of this exercise.
c) Lock an associated mutex without locking using `std::unique_lock::try_lock`. Take care of exceptions such as i) no associated mutex and ii) mutex is already locked. You need to use `std::system::error_code` and `std::errc` as discussed in section 2.7 of the course.
d) Apply `std::unique_lock::try_lock_for` that tries to lock a mutex that blocks until a specific *timeout duration* has elapsed or the lock has been acquired. As in part c) you need to catch system errors.
e) Apply `std::unique_lock::unlock` that unlocks the associated mutex and releases ownership. As in parts c) and d) you need to catch system errors.
f) Apply `std::unique_lock::try_lock_until` that tries to lock a mutex that blocks until a *specific time* has been reached or the lock has been acquired. As in parts c), d) and e) you need to catch system errors.
g) Show how to release an associated mutex without unlocking it.

4. (Synchronising Queue)

We implement a synchronising (thread-safe) FIFO queue in C++11. In this exercise we create our own generic synchronising queue having an embedded STL queue.

Answer the following questions:

a) Implement this template class, in particular its template parameters and functions for queuing and dequeing.
b) Add locking mechanisms to functions for synchronisation effects.
c) Add notification mechanisms to allow thread notification (use condition variables for notification).
d) Test the queue by creating producer and consumer threads.

5. (Atomics and Atomic Operations 101)

The objective of this exercise is to become acquainted with the mechanics of using atomic data types and atomic operations in C++11.

Answer the following questions:

a) Create atomic objects of type `long long`, `bool` and `int`.
b) Determine which atomic types are lock-free.
c) Atomically replace the value of one atomic object by another atomic object.
d) Atomically load an atomic object and return its current value.

3.2/3.3 C++ Concurrency

Summary and Goals

The objective of the exercises in this section is to learn some more C++11 syntax and features and to apply them to several simple parallel design patterns. See “Patterns for Parallel Programming” by Timothy G. Mattson et al 2005.

1. (Atomic Operations on Shared Pointers)

In general, *shared pointers* are not thread-safe. However, reading the count while another thread modifies a shared pointer does not introduce a race condition but the value may not be up-to-date.
The syntax in this context is a bit tricky. In particular, you have to take *memory ordering* into account. To keep things focused, we shall work with instances (all shared pointers) of the class:

```
struct X
{
    double val;

    X() : val(0.0) {}
    void operator ()()
    {
           std::cout << "An X " << val << std::endl;
    }
};

// C++11
template <typename T>
    using GenericPointerType = std::shared_ptr<T>;
using PointerType = GenericPointerType<X>;
```

Answer the following questions:

a) Create an instance `x` of class `PointerType` and give it a value.
b) Create a second instance `x2` and atomically store it value in `x`.
c) Create an instance `x3` and exchange it with both `x` and `x2`.
d) Keep tabs on the *use count* of these instances.
e) Why is `atomic<std::shared_ptr>` not possible?
f) Are shared pointers lock-free?

2. (“Smart Races with Smart Pointers”)

The objective of this exercise is to show how we can simulate (undesirable) non-deterministic race conditions when multiple threads access functions that use shared pointers. This is good to know if you are considering porting single-threaded code that makes (possibly extensive) use of smart pointers and we wish to avoid race conditions and non-deterministic values. We use the same class `PointerType` as in exercise 1.

Answer the following questions:

a) Create a function that accepts a smart pointer and a new value for its state:

```
void Modify(PointerType& p, double newVal)
{

    // Wait a while, long enough to trigger the race
    // TBD

           p->val = newVal;

    // TBD
}
```

The body should actually update the state and we should introduce enough latency (delay) in order to trigger a race condition; for example, you generate a uniform random number using C++ random library and then sleep for that amount of time.

b) Create an array of 100 threads, fire each one up (each one has thread function `Modify()` and a given value that will be the new value of the instance of `PointerType`).
c) Run the program a number of times; examine the output and explain what is happening.

3. (Active Objects 101)

An *active object* is one that executes in its own thread of control. One advantage of creating/using active objects is that it results in a one-to-one correspondence between an object and a thread. C++ does not directly support active objects. It only supports *passive objects* that are acted on by other objects.
In this chapter we create an *adapter class* called `ActiveObject` that is composed of a thread. The requirements are:

- `ActiveObject` has a reference to an instance of an `std::thread` instance.
- It has an `explicit` constructor accepting a thread with associated thread function. The active object starts computing as soon as it is instantiated (in principle).
- In the destructor of `ActiveObject` we check if the thread is joinable and if it is we then join it.
- Check that the thread has an associated thread function; use try and catch blocks in client code.

In module 5 we shall also discuss active objects in the context of the *Command Processor* and *Command* patterns. More generally, the *Active Object design pattern* decouples method execution from method invocation to enhance concurrency and simplify synchronised access to objects that reside in their own thread of control (as seen in the ACE library, for example).

Answer the following questions taking the above requirements into account:

a) Create a thread and a function that will be its thread function.
b) Create an active object based on a thread with no associated thread function.
c) Implement thread functions using function objects and lambda functions.
d) Test the code.

4. (My First *Master-Worker* Pattern in C++11)

The *Master-Worker pattern* can be summarised as follows: “A common problem in parallel programming is how to balance the computational load among a set of processing elements within a parallel computer. For task parallel programs with no communication between tasks (or infrequent but well-structured, anonymous communication) an effective solution with “automatic dynamic load balancing” is to define a single master to manage the collection of tasks and collect results. Then a set of workers grab a task, do the work, send the results back to the master, and then grab the next task. This continues until all the tasks have been computed.”

In this exercise we create a simple application consisting of one master and one worker. The worker is responsible for the updating of a shared resource (in this case, a string) on behalf of the master. Some hints on the design of this exercise are:

- The master initiates the computation and sets up the problem.
- Then the master creates a *bag of tasks* (in our case just one worker).
- The master waits until the job is done, consumes the results and then shuts down.

Answer the following questions:

a) We start with the following global data that is needed in order to implement this pattern:
   1: The resource that the worker updates.
   2: A mutex to synchronise access to the resource.
   3: A condition variable for thread notification.
   4: Two atomic Booleans to model the status of the master and worker.

```
// Types and data needed
std::string data;// Shared data between master and worker
std::mutex m;     // Ensures no race condition

// Synchronisation between master and worker
std::condition_variable cv;

// Initial state of worker and master
std::atomic<bool> workerReady = false;
std::atomic<bool> masterReady = false;
```

b) Create the thread function: it updates the resource by acquiring the lock, waiting on the worker ready flag to be set, updating the resource and then finally notifying the master.
c) Create the code that implements the master's duties. Test the code.
d) We now replace atomic Booleans by class `std::atomic_flag` which is a bare-bones *lock-free* data type. It does not provide load nor store operations. It has functions to set and clear. Modify the code accordingly and test again.

5. (Barbershop Problem)

Consider the following problem:
"A barbershop consists of a waiting room with n chairs. There is one barber chair in the room. The barber goes to sleep if there are no customers in the room. If a customer enters the barber shop and all chairs are occupied then the customer leaves the shop. If the barber is busy and there are chairs available then the customer occupies one of these free chairs. If the barber is asleep he is woken by a customer."

The main events in this example are:

- Arrival of a client.
- Starting of a service for a client.
- Client leaves barber shop after having received service.
- Client leaves the barber shop if it is full.
- The barber waits (sleeps or does something else) if there are no clients waiting for service.

The objective of this exercise is to write a program to coordinate the barber and the customers using the *Producer-Consumer pattern*.

Answer the following questions:

a) Create producer and consumer classes. Producers arrive every 10 seconds. There is one customer.
b) Reuse the class for synchronising queues.
c) Test the code.

3.4 C++11 Tasks

Summary and Goals

In this section we concentrate on tasking in C++11 and creating code that offers more functionality and is less prone to subtle errors than when using threads. It is the so-called *high-level interface*.

1. (Tasks 101: Running Functions asynchronously)

For this exercise, you need to understand C++11 *futures*, `std::async` and thread/task launching policies. This exercise involves running synchronous and asynchronous functions based on various launch policies. The functions should have both `void` and non-`void` return types to show how clients can use the results of computations. The objective is not in creating complex functions but in showing how the tasks execute and exchange information.

Answer the following questions:

a) Create two functions having the following signatures:

```
void func1()
{

}

double func2(double a, double b)
{

}
```

You may place any code into the body of these functions for the purpose of this exercise.

b) Use `std::async` (default settings) to launch `func1` and `func2`. Get the results of the computations and print them when applicable. Check the validity of the associated future before and after getting the result.

c) What happens if you try to get the result of a `std::future` more than once?

d) Now test the same code using the launch parameter `std::launch::async`. Do you notice any differences?

e) We now wish to asynchronously call a function at some time later in the client code (*deferred/lazy evaluation*). Get the result of the function and check that it is the same as before.

2. (Shared Futures 101)

We wish to process the outcome of a concurrent computation more than once, especially when multiple threads are running and to this end we use `std::shared_future` so that we can make multiple calls to `get()`.

Answer the following questions:

a) Create the following shared future by calling the appropriate constructor:
- Default instance.
- As a shared future that shares the same state as another shared state.
- Transfer shared state from a 'normal' future to a shared future.
- Transfer shared state from a shared future to a shared future.

b) Check the member functions from `std::future` that they are also applicable to `std::shared_future`.

c) Test what happens when you call `get()` twice on a shared future.

d) Create a shared future that waits for an infinite loop to finish (which it never does). To this end, use `wait_for` and `wait_until` to trigger a time out.

3. (C++11 Promises 101)

A *promise* is the counterpart of a future object. Both are able to temporarily hold a shared state. Thus, a promise is a general mechanism to allow values and exceptions to be passed out of threads. A promise is the "push" end of the promise-future communication channel.

Answer the following questions:

a) Create a default promise, a promise with an empty shared state and a promise based on the move constructor.
b) Create a promise with `double` as stored value. Then create a future that is associated with the promise.
c) Start a thread with the new future from part b). Create a thread function that uses the value of the shared data.
d) Use the promise to set the value of the shared data.

4. (Packaged Tasks)

The added value of using a *packaged task* is that we can create a background task without starting it immediately. In particular, the task is typically started in a separate thread.

Answer the following questions:

a) Consider the thread function:

```
// Thread function
double compute(double x, double y)
{
    // Wait a while
    std::default_random_engine dre(42);
    std::uniform_int_distribution<int> delay(0, 1000);
    std::this_thread::sleep_for
           (std::chrono::milliseconds(delay(dre)));

    return std::cos(x) * std::exp(y);
}
```

Write code to start a task:

```
double x = 0.0; double y = 2.71;

// A. 'Direct' tasks
std::future<double> fut = std::async(compute, x, y);

// Get the shared data
double result = fut.get();
std::cout << "Result: " << result << '\n';
```

b) Rewrite/port the code in order to use a packaged task and delayed execution.
c) Create a queue of packaged tasks, dequeue each task and execute it.

5. (Loop-Level Parallelism with Tasks)

The objective of this exercise is compute *reduction/aggregated* variables relating to numerical array. In order to scope the problem we focus on summing the values of numeric arrays using the so-called *accumulate* function. The main requirements are *speedup improvements* and *serial equivalence* (this means that the multi-threaded solution gives the same results as the single-threaded solution). We start with a single-threaded solution and we incrementally move to multi-threaded solutions based on various parallel design patterns.

Answer the following questions:

a) Create a huge numeric array and sum its elements using `std::accumulate()`. Measure processing time.

b) We create an algorithm to parallelise the code in part a). To this end, we split the index space into two halves; the right half is run in an asynchronous task while the left part constitutes a recursive call to the algorithm for the appropriate index range. If the size of the array is less than 1000 we then side-track parallel processing by calling `std::accumulate()` directly.
c) Measure the speedup of the code in part b).
d) Compare the speedup with that achieved by using OpenMP. Typical code is:

```
#include <omp.h>
omp_set_num_threads(2);
#pragma omp parallel for reduction (+:sumParallel)
for (long i = 0; i < v.size(); ++i)
{
    sumParallel += v[i];
}
```

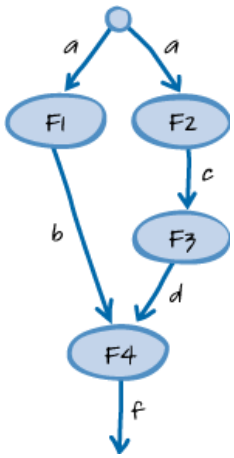
(OpenMP is delivered with Visual Studio).
e) Now use threads instead of tasks in order to perform accumulation. Compare speedup.
f) Write parallel code that use `std::accumulate()` with a binary function as last parameter. Test the algorithm by computing the product of the elements of an array.
g) Compare performance improvements by using the new parallel features in C++. We focus on three STL algorithms (or algorithms of your choice) as applied to std::vector, namely: std::transform, std::sort, std::find.

Answer the following questions:

- For a vector of a given size N (say N = 10'000'000) compute the running time of these algorithms in the sequential and parallel versions.
- Measure the running times for a range of values of N. Can you find the speedup?[CT1]

6. (Task Graphs: Implementation in C++11)

In this exercise we start from a high-level *data dependency graph* for a system to be developed and we map it to an implementation in C++11. The graph is:



The bodies of the functions can be anything you like while the data is also arbitrary.

Answer the following questions:
a) Write single-threaded code for this dependency graph. This will be the baseline code. Measure the processing time.
b) Implement the graph using `std::async` and `std::future`.
c) Implement the graph using promises.
d) Implement the graph using packaged tasks.

Experiment with various data types (built-in types, vectors and matrices) and functions (lightweight/heavyweight). Measure *speedup* and *load balancing* efforts.

3.5 Complexity and STL Containers

Summary and Goals

The objectives in this exercise are:

- Discuss some ADTs (Abstract Data Types), their complexity and implementation.
- Useful utility (a timer class).
- Single-threaded *Command Processor* pattern.
- Synchronised queue and *Producer-Consumer* pattern.
- *Forward lists*.

The experience in this section will be useful in later modules and it dovetails well with the material on multi-threading and C++ concurrent programming.

1. (Priority Queue 101)

A *priority queue* is a collection of elements each of which having a priority value. When an element is chosen for deletion it is always the element with the highest priority. We thus see that a priority queue is not a queue (the latter being FIFO (First In First Out)). We can implement priority queues as:

- A list ordered by priority values (highest priority element at the end of the list).
- A simple unordered list (O (n) complexity in the worst case).
- A heap (supported in STL) (insertion and deletion have logarithmic complexity).

This exercise concerns the priority queue class in STL defined by:

```
template
<
    class T,
    class Container = std::vector<T>,
    class Compare =
           std::less<typename Container::value_type>
> class priority_queue;
```

Its member functions are:

- `push()` insert an element into the priority queue.
- `top()` return the next element in the priority queue.
- `pop()` remove an element from the priority queue.

Answer the following questions:

a) Create a priority queue whose elements are `long long` using the above default template parameters. In particular, perform the following:
   - Push values 66, 22, 44.
   - Print top element and the pop it.
   - Push values 11,22,33.
   - Pop element.
   - Iterate in the priority until it is empty:
     - Print top element.
     - Pop it.

   Inspect the output and convince yourself that is consistent with your expectations.

b) Create a priority queue whose elements are `long long` whose container is `std::deque` and whose comparator is `std::greater<long long>`. Test this case with the data in part a) of the exercise. Inspect the output.

c) Modify the code in part b) to use a lambda function instead of `std::greater<long long>`. Inspect the output.

2. (Simple Single-Threaded *Command Processor* Pattern)

In this exercise we create a mini-application. We generalise and extend it in Module 5. It is a single-threaded *producer-consumer pattern* in which executable commands are inserted into a priority queue. When finished, consumer code executes each command in the priority queue in turn before popping it.

The class in question is:

```
using FunctionType = std::function<double (double)>;

class Command
{
private:
    long ID; // priority of command
    FunctionType algo;

public:
Command(const FunctionType& algorithm, long priority)
          : algo(algorithm), ID(priority){}

    void Execute(double x)
    {
          // Introduce delay to simulate a heavy algorithm
          boost::this_thread::sleep
                (boost::posix_time::millisec(5000));
          std::cout << algo(x) << '\n';
    }

    int priority() const
    {
          return ID;
    }
};
```

Answer the following questions:

a) Create a comparator for `Command` to compare two of its instances.
b) Create several instances of `Command`.
c) Create a priority queue of commands and insert the objects from part b) into it.
d) Execute each command in the priority queue until it is empty.

3. (Forward List 101)

The C++11 `std::forward_list` is a container that supports fast insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is implemented as a singly-linked list and essentially does not have any overhead compared to its implementation in C. Compared to `std::list` this container provides more space efficient storage when bidirectional iteration is not needed.

This exercise entails investigating this new class.

Answer the following questions:

a) Create default list, a list with n elements and a given value, and a list that is built from an *initialiser list* (two forms).
b) Forward lists have no member functions to give their size. Write you own function in terms of `std::distance()`.
c) Create code to insert values after a given position based on value, value and a count, two iterators and initialiser list. The code makes direct calls to *std::forward_list::insert_after*.
d) Create functions to erase values after a given positon as well as in a range of iterators.
e) Create examples to show how *splice* and *merge* functionality work.

4. (Synchronising Queue)

We have already discussed synchronising (thread-safe) FIFO queues. In this exercise we create our own generic synchronising queue having an embedded STL priority queue (both the latter's container type and comparator are generic).

Answer the following questions:

a) Implement this template class, in particular its template parameters and functions for queueing and dequeuing.
b) Add locking mechanisms to functions for synchronisation effects.
c) Add notification mechanisms to allow thread notification.
d) Test the new class.

5. (Producer-Consumer Pattern)

We now generalise exercise 2 to accomodate multiple concurrent producers and consumers of *commands*. First, implement the solution in Boost and then migrate it to C++11.

Answer the following questions:

a) Create a thread group for producers and add commands to it.
b) Create a thread group for consumers and add commands to it.
c) Start the application; we need to know when to stop processing and to this end we define a special command object that can be recognized by consumers to let them know that there is no more input.

(In general, we could use libraries such as Microsoft's PPL and Intel's TBB that have support for *concurrent containers.* These libraries are outside the scope of the current course).

6. (std::chrono)

In this exercise we create an adapter class for certain functionality in the `std::chrono` library. The objective is to create a class to measure how long it takes for an operation to complete.

Answer the following questions:

a) Implement an adapter class with the following interface:

```
class StopWatch
{
public:
    StopWatch();

    void StartStopWatch();
    void StopStopWatch();
    void Reset();

    double GetTime() const;

private:
    StopWatch(const StopWatch &);
    StopWatch & operator=(const StopWatch &);

};
```

b) Test this class.

7. (Compile-Time Fractional Arithmetic 101)

This exercise is based on `std::ratio`, a class template that supports rational arithmetic. It reduces the risk of run-time overflow because a ratio and any ratios resulting from ratio arithmetic are always reduced to the lowest terms.

Several convenient *typedefs* that correspond to the SI ratios are provided by the standard library.

Answer the following questions:

a) Create a number of instances of `std::ratio` and print them.
b) Create two `std::ratio` instances; add, subtract, multiply and divide them.